```c
/*
 * @author Derek Trom
 * @author Elena Corpus
 * io module
 */
#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
 * io.h is responsible for reading/writing to the system and files.
*/

#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 */
```

```c
/*
 * @author Derek Trom
 * @author Elena Corpus
 * Abstract syntax tree magic
 */
#include "ast.h"
#include "io.h"
#include <stdlib.h>

AST *astroot = NULL;

const char *astnodestr[numasms] = {
        "eofasm",

        /* Operators */
        "addasm",
        "multasm",

        /* Scopes */
        "programasm",
        "procedureasm",
        "functionasm",
        "paramasm",
        "statementasm",
        "proccallasm",
        "funccallasm",

        /* Expressions */
        "exprasm",
        "simexprasm",
        "termasm",
        "factorasm",

        /* Boolean operators */
        "relasm",
        "notasm",

        /* Punctuation */
        "assignasm",
        "dotdotasm",

        /* Control flow */
        "ifasm",
        "whileasm",

        /* Variables */
        "idasm",
        "arrayasm",
        "ofasm",
        "charasm",
```

```
51          "stringasm",
52          "integerasm",
53          "realasm",
54          "varasm",
55
56          /* Constants */
57          "valasm",
58          "constasm",
59
60          /* Built-in functions */
61          "chrasm",
62          "ordasm",
63          "readasm",
64          "readlnasm",
65          "writeasm",
66          "writelnasm",
67  };
68
69  AST *AST_initialize(ASTnode node) {
70      AST *ast;
71
72      if (!(ast = malloc(sizeof(*ast)))) {
73          pcerror("Out of memory.\n");
74          return NULL;
75      }
76
77      ast->node = node;
78      ast->name = NULL;
79      ast->sym = eofsym;
80      ast->val.ival = 0;
81      ast->head = NULL;
82      ast->tail = NULL;
83
84      return ast;
85  }
86
87  int AST_addchild(AST *root, AST *child) {
88      ASTchild *cur;
89
90      if (!(cur = malloc(sizeof(*cur)))) {
91          return pcerror("Out of memory.\n");
92      }
93
94      cur->ast = child;
95      cur->next = NULL;
96
97      /* see if we have any children yet, and initialize if
    we don't */
98      if (!root->head) {
99          root->head = cur;
```

```
100         root->tail = cur;
101
102         return 1;
103     }
104
105     /* update the tail */
106     root->tail->next = cur;
107     root->tail = cur;
108     return 1;
109 }
110
111 void AST_cleanup(AST **root) {
112     AST *ast;
113     ASTchild *cur;
114
115     ast = *root;
116
117     /* clean up all our children, left-to-right, starting
    at the deepest child */
118     while ((cur = ast->head)) {
119         AST_cleanup(&(cur->ast));
120         ast->head = cur->next;
121
122         /* cleanup the actual child */
123         cur->next = NULL;
124         free(cur);
125     }
126
127     if (ast->name) free(ast->name);
128     ast->name = NULL;
129     ast->head = NULL;
130     ast->tail = NULL;
131
132     free(ast);
133     *root = NULL;
134 }
135
136 const char *AST_nodestr(ASTnode node) {
137     switch (node) {
138         /* End-of-Tokens */
139         case eofasm:
140             return "eof";
141
142             /* Operators */
143         case addasm:
144             return "add";
145         case multasm:
146             return "mult";
147
148             /* Scopes */
```

```
149         case programasm:
150             return "program";
151         case procedureasm:
152             return "procedure";
153         case functionasm:
154             return "function";
155         case paramasm:
156             return "param";
157         case statementasm:
158             return "statement";
159         case proccallasm:
160             return "proccall";
161         case funccallasm:
162             return "funccall";
163 
164             /* Expressions */
165         case exprasm:
166             return "expr";
167         case simexprasm:
168             return "simexpr";
169         case termasm:
170             return "term";
171         case factorasm:
172             return "factor";
173 
174             /* Boolean operators */
175         case relasm:
176             return "rel";
177         case notasm:
178             return "not";
179 
180             /* Punctuation */
181         case assignasm:
182             return "assign";
183         case dotdotasm:
184             return "dotdot";
185 
186             /* Control flow */
187         case ifasm:
188             return "if";
189         case whileasm:
190             return "while";
191 
192             /* Variables */
193         case idasm:
194             return "id";
195         case arrayasm:
196             return "array";
197         case ofasm:
198             return "of";
```

```
199         case charasm:
200             return "char";
201         case stringasm:
202             return "string";
203         case integerasm:
204             return "integer";
205         case realasm:
206             return "real";
207         case varasm:
208             return "var";
209 
210             /* Constants */
211         case valasm:
212             return "val";
213         case constasm:
214             return "const";
215 
216             /* Built-in functions */
217         case chrasm:
218             return "chr";
219         case ordasm:
220             return "ord";
221         case readasm:
222             return "read";
223         case readlnasm:
224             return "readln";
225         case writeasm:
226             return "write";
227         case writelnasm:
228             return "writeln";
229 
230             /* Number of syms */
231         case numasms:
232             return "numasms";
233 
234         default:
235             return "ERR";
236     }
237 
238     return "ERR";
239 }
240 
241 void AST_print_internal(AST *root, FILE *fp, int depth) {
242     int i = depth;
243     char str[1024];
244     char *c = str;
245     ASTchild *cur = root->head;
246 
247     while (i--) *c++ = '\t';
248 
```

```
249     if (root->name) {
250         if (root->val.ival) snprintf(c, 1023 - depth, "[%s
     name:%s val:Y]\n", AST_nodestr(root->node), root->name);
251         else snprintf(c, 1023 - depth, "[%s name:%s]\n",
    AST_nodestr(root->node), root->name);
252     } else {
253         if (root->val.ival) snprintf(c, 1023 - depth, "[%s
     val:Y]\n", AST_nodestr(root->node));
254         else snprintf(c, 1023 - depth, "[%s]\n",
    AST_nodestr(root->node));
255     }
256 
257     /* print to file */
258     fprintf(fp, "%s", str);
259 
260     /* print children in order */
261     while (cur) {
262         AST_print_internal(cur->ast, fp, depth + 1);
263         cur = cur->next;
264     }
265 }
266 
267 void AST_print(AST *root, FILE *fp) {
268     AST_print_internal(root, fp, 0);
269 }
270 
```

```
 1 /*
 2  * @author Derek Trom
 3  * @author Elena Corpus
 4  * ast.h
 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  extern const char *astnodestr[numasms];
74
75  struct ASTchild;
76  typedef struct AST {
77      ASTnode node;    /* node type */
78      char *name;    /* name in the symbol table */
79      pcsym sym;    /* symbol */
80      symval val;    /* value */
81      struct ASTchild *head;    /* left-most child */
82      struct ASTchild *tail;    /* right-most child */
83  } AST;
84
85  typedef struct ASTchild {
86      AST *ast;    /* value of the child */
87      struct ASTchild *next;    /* next child, left-to-right
     */
88  } ASTchild;
89
90  /* Our global AST */
91  extern AST *astroot;
92
93  /* Initializes an AST for use.
94
95  @param node the type of AST
96  @return memory allocated AST
97  */
98  AST *AST_initialize(ASTnode node);
99
```

```
100 /* Adds a child to the AST, in left-to-right order.
101
102 @param child the AST to add
103 @return 1 on success; 0 otherwise
104 */
105 int AST_addchild(AST *root, AST *child);
106
107 /* Cleans up the memory for a given AST.
108
109 @param root the AST to cleanup
110 */
111 void AST_cleanup(AST **root);
112
113 /* Print an AST tree to the given file.
114
115 @param fp the file pointer
116 */
117 void AST_print(AST *root, FILE *fp);
118
119 #endif /* AST_H */
```

```
 1 /*
 2  * @author Derek Trom
 3  * @author Elena Corpus
 4  * This is the code generation file to creat assembly code.
 5  */
 6 #include "icg.h"
 7 #include "io.h"
 8 #include "symtab.h"
 9 #include <stdio.h>
10 #include <string.h>
11
12 #define EXPECTICG(ASTV, NODEV) if (!expect(ASTV, NODEV))
   return 0
13
14 int pcicg_block(AST *ast, symentry *entry, const char *
   label, ASTchild *params);
15
16 int pcicg_simple_expression(AST *ast, symtype *type, int *t
   );
17
18 int pcicg_statement(AST *ast);
19
20 int pcicg_funccall(AST *ast, symtype *type, int *t);
21
22 FILE *fp;
23 int ifcount, whilecount, forcount;
24
25 int accept(AST *ast, ASTnode node) {
26     return ast && ast->node == node;
27 }
28
29 int expect(AST *ast, ASTnode node) {
30     if (!ast) return pcerror("AST DOESN'T EXIST!\n");
31     if (!accept(ast, node))
32         return pcerror("Unexpected node: %s vs %s\n",
   astnodestr[ast->node], astnodestr[node]);
33     return 1;
34 }
35
36 /* Convert from one type to another, if required. */
37 int pcicg_convert(symtype totype, symtype fromtype, int t
   ) {
38     int error = 0;
39
40     if (totype == integertype) {
41         if (fromtype == realtype) {
42             /* convert from real to integer */
43             fprintf(fp, "cvt.w.s $f%d, $f%d\n", t, t);
44             fprintf(fp, "mfc1 $t%d, $f%d\n", t, t);
45         } else if (fromtype != integertype) {
```

```
46              error = 1;
47          }
48      } else if (totype == realtype) {
49          if (fromtype == integertype) {
50              /* convert from integer to real */
51              fprintf(fp, "mtc1 $f%d, $f%d\n", t, t);
52              fprintf(fp, "cvt.s.w $f%d, $f%d\n", t, t);
53          } else if (fromtype != realtype) {
54              error = 1;
55          }
56      } else if (totype != fromtype) {
57          error = 1;
58      }
59
60      if (error) return pcerror("Unable to convert to type %s
    from %s.\n", symtypestr[totype], symtypestr[fromtype]);
61      return 1;
62  }
63
64  int pcicg_var(AST *ast) {
65      ASTchild *cur;
66      symentry *entry;
67
68      printf("===== ENTERING pcicg_var =====\n");
69
70      /*
71      add our variables to the stack
72      the first variables have the largest offset
73      */
74      cur = ast->head;
75      while (cur) {
76          /* keep track of our sum so we can get the correct
    offset */
77          entry = pclookupsym(cur->ast->name);
78
79          /* initialize to zero */
80          switch (entry->type) {
81              case chartype:
82              case integertype:
83                  fprintf(fp, "sw $0, %d($sp)\n", entry->
    offset);
84                  break;
85              case realtype:
86                  fprintf(fp, "sw.s $0, %d($sp)\n", entry->
    offset);
87                  break;
88              default:
89                  return pcerror("Unhandled var type: %d\n",
    entry->type);
90          }
```

```
 91
 92         cur = cur->next;
 93     }
 94
 95     return 1;
 96 }
 97
 98 int pcicg_const(AST *ast) {
 99     ASTchild *cur;
100     symentry *entry;
101     int i;
102     int len;
103
104     printf("===== ENTERING pcicg_const =====\n");
105
106     /*
107     add our variables to the stack
108     the first variables have the largest offset
109     */
110     cur = ast->head;
111     while (cur) {
112         /* keep track of our sum so we can get the correct
     offset */
113         entry = pclookupsym(cur->ast->name);
114
115         switch (entry->type) {
116             case chartype:
117                 fprintf(fp, "li $t0, %d\n", (int) entry->
    val.cval);
118                 fprintf(fp, "sb $t0, %d($sp)\n", entry->
    offset);
119                 break;
120
121             case integertype:
122                 fprintf(fp, "li $t0, %d\n", entry->val.
    ival);
123                 fprintf(fp, "sw $t0, %d($sp)\n", entry->
    offset);
124                 break;
125
126             case realtype:
127                 fprintf(fp, "li.s $f0, %f\n", entry->val.
    rval);
128                 fprintf(fp, "s.s $f0, %d($sp)\n", entry->
    offset);
129                 break;
130
131             case stringtype:
132                 len = strlen(entry->val.str);
133
```

```
134                 for (i = 0; i < len; ++i) {
135                     fprintf(fp, "li $t0, %d\n", (int) (
    entry->val.str[i]));
136                     fprintf(fp, "sb $t0, %d($sp)\n", i +
    entry->offset);
137                 }
138 
139                 fprintf(fp, "li $t0, 0\n");
140                 fprintf(fp, "sb $t0, %d($sp)\n", i + entry
    ->offset);
141                 break;
142 
143             default:
144                 return pcerror("Unhandled type: %d\n", (
    int) entry->type);
145         }
146 
147         cur = cur->next;
148     }
149 
150     return 1;
151 }
152 
153 int pcicg_function(AST *ast) {
154     ASTchild *cur;
155     ASTchild *params;
156     symentry *entry;
157 
158     printf("===== ENTERING pcicg_function =====\n");
159 
160     if (!(entry = pclookupsym(ast->name)) || entry->type
     != functiontype) {
161         return pcerror("Unable to lookup function.\n");
162     }
163 
164     /* check our parameters */
165     cur = ast->head;
166     EXPECTICG(cur->ast, paramasm);
167     params = cur->ast->head;
168 
169     /* parse the block */
170     if (!pcicg_block(ast, entry, entry->name, params))
    return 0;
171 
172     /* reload in the return address */
173     fprintf(fp, "lw $ra, %d($sp)\n", entry->size);
174 
175     /* pop the stack */
176     fprintf(fp, "addi $sp, $sp, %d\n", entry->size + 4);
177 
```

```
178     /* add a jump back to the caller */
179     fprintf(fp, "jr $ra\n\n");
180 
181     return 1;
182 }
183 
184 int pcicg_procedure(AST *ast) {
185     ASTchild *cur;
186     ASTchild *params;
187     symentry *entry;
188 
189     printf("===== ENTERING pcicg_procedure =====\n");
190 
191     if (!(entry = pclookupsym(ast->name)) || entry->type
     != proceduretype) {
192         return pcerror("Unable to lookup procedure.\n");
193     }
194 
195     /* get our parameters */
196     cur = ast->head;
197     EXPECTICG(cur->ast, paramasm);
198     params = cur->ast->head;
199 
200     /* parse the block */
201     if (!pcicg_block(ast, entry, entry->name, params))
    return 0;
202 
203     /* reload the return address */
204     fprintf(fp, "lw $ra, %d($sp)\n", entry->size);
205 
206     /* pop the stack */
207     fprintf(fp, "addi $sp, $sp, %d\n", entry->size + 4);
208 
209     /* add a jump back to the caller */
210     fprintf(fp, "jr $ra\n\n");
211 
212     return 1;
213 }
214 
215 int pcicg_factor(AST *ast, symtype *type, int *t) {
216     ASTchild *cur;
217     symentry *entry;
218     int offset;
219     int left;
220     symtype forcetype = notype;
221 
222     printf("===== ENTERING pcicg_factor =====\n");
223 
224     cur = ast->head;
225 
```

```
226     /* load id value */
227     if (accept(cur->ast, idasm)) {
228         if (!(entry = pclookupsym_entry(cur->ast->name, &
    offset)))
229             return pcerror("Unable to load entry: %s\n",
    cur->ast->name);
230 
231         switch (entry->type) {
232             case chartype:
233                 fprintf(fp, "li $t%d, 0\n", *t);
234                 fprintf(fp, "lb $t%d, %d($sp)\n", *t,
    offset);
235                 break;
236 
237             case integertype:
238                 fprintf(fp, "lw $t%d, %d($sp)\n", *t,
    offset);
239                 break;
240 
241             case realtype:
242                 fprintf(fp, "li.s $f%d, %d($sp)\n", *t,
    offset);
243                 break;
244 
245             case stringtype:
246                 fprintf(fp, "la $t%d, %d($sp)\n", *t,
    offset);
247                 break;
248 
249             default:
250                 return pcerror("Unhandled type: %s\n",
    symtypestr[entry->type]);
251         }
252 
253         *type = entry->type;
254     }
255 
256         /* ord value */
257     else if (accept(cur->ast, ordasm)) {
258         cur = cur->next;
259         EXPECTICG(cur->ast, exprasm);
260 
261         cur = cur->next;
262         EXPECTICG(cur->ast, simexprasm);
263         forcetype = chartype;
264         if (!pcicg_simple_expression(cur->ast, &forcetype
    , t)) return 0;
265 
266         *type = chartype;
267     }
```

```
268
269         /* chr value */
270     else if (accept(cur->ast, chrasm)) {
271         cur = cur->next;
272         EXPECTICG(cur->ast, exprasm);
273
274         cur = cur->next;
275         EXPECTICG(cur->ast, simexprasm);
276         forcetype = integertype;
277         if (!pcicg_simple_expression(cur->ast, &forcetype
    , t)) return 0;
278
279         /* truncate down to a single byte */
280         fprintf(fp, "li $t%d, 0", (*t) + 1);
281         fprintf(fp, "sb $t%d, $t%d", (*t) + 1, *t);
282         fprintf(fp, "sw $t%d, $t%d", *t, (*t) + 1);
283
284         *type = integertype;
285     }
286
287         /* not value */
288     else if (accept(cur->ast, notasm)) {
289         cur = cur->next;
290         EXPECTICG(cur->ast, factorasm);
291
292         left = *t;
293         *t += 1;
294
295         if (!pcicg_factor(cur->ast, type, t)) return 0;
296         if (!pcicg_convert(integertype, *type, *t)) return
     0;
297
298         fprintf(fp, "addi $t%d, $0, -1", left);
299         fprintf(fp, "xor $t%d, $t%d, $t%d", left, left, *t
    );
300
301         *t = left;
302         *type = integertype;
303     }
304
305         /* expr */
306         /*
307         else if (accept(cur->ast, exprasm)) {
308             cur = cur->next;
309         }
310         */
311
312         /* val */
313     else if (accept(cur->ast, valasm)) {
314         switch (cur->ast->sym) {
```

```
315             case charvalsym:
316                 fprintf(fp, "li $t%d, %d\n", *t, (int) (
    cur->ast->val.cval));
317                 *type = chartype;
318                 break;
319 
320             case integernosym:
321                 fprintf(fp, "li $t%d, %d\n", *t, cur->ast
    ->val.ival);
322                 *type = integertype;
323                 break;
324 
325             case realnosym:
326                 fprintf(fp, "li.s $f%d, %f\n", *t, cur->
    ast->val.rval);
327                 *type = realtype;
328                 break;
329 
330             default:
331                 return pcerror("Unhandled type: %s\n",
    symtypestr[entry->type]);
332         }
333     }
334 
335         /* function call */
336     else if (accept(cur->ast, funccallasm)) {
337         if (!pcicg_funccall(cur->ast, type, t)) return 0;
338     }
339 
340         /* UNKNOWN! */
341     else {
342         return pcerror("Unexpected node: %s\n", astnodestr
    [cur->ast->node]);
343     }
344 
345     /* woot */
346     return 1;
347 }
348 
349 int pcicg_term(AST *ast, symtype *type, int *t) {
350     ASTchild *cur;
351     symtype factortype;
352     pcsym sym;
353     int left;
354 
355     printf("===== ENTERING pcicg_term =====\n");
356 
357     /* grab the first factor */
358     cur = ast->head;
359     EXPECTICG(cur->ast, factorasm);
```

```
360     if (!pcicg_factor(cur->ast, type, t)) return 0;
361 
362     /* update our left-most value for chaining */
363     left = *t;
364     *t += 1;
365 
366     /* go through all our multops */
367     while (cur->next) {
368         cur = cur->next;
369         EXPECTICG(cur->ast, multasm);
370         sym = cur->ast->sym;
371 
372         cur = cur->next;
373         EXPECTICG(cur->ast, factorasm);
374         if (!pcicg_factor(cur->ast, &factortype, t))
    return 0;
375 
376         /* convert and perform the mul function */
377         if (!pcicg_convert(*type, factortype, *t)) return
    0;
378 
379         if (*type == integertype) {
380             if (sym == multsym) {
381                 fprintf(fp, "mul $t%d, $t%d\n", left, *t);
382                 fprintf(fp, "mflo $t%d\n", left);
383             } else if (sym == divsym) {
384                 fprintf(fp, "div $t%d, $t%d\n", left, *t);
385                 fprintf(fp, "mflo $t%d\n", left);
386             } else if (sym == idivsym) {
387                 fprintf(fp, "div $t%d, $t%d\n", left, *t);
388                 fprintf(fp, "mflo $t%d\n", left);
389             } else if (sym == modsym) {
390                 fprintf(fp, "div $t%d, $t%d\n", left, *t);
391                 fprintf(fp, "mfhi $t%d\n", left);
392             } else if (sym == andsym) {
393                 fprintf(fp, "and $t%d, $t%d, $t%d\n", left
    , left, *t);
394             } else {
395                 return pcerror("Unknown multiplication
    symbol: %s\n", symtypestr[sym]);
396             }
397         } else if (*type == realtype) {
398             if (sym == multsym) {
399                 fprintf(fp, "mul.s $f%d, $f%d, $f%d\n",
    left, left, *t);
400             } else if (sym == divsym) {
401                 fprintf(fp, "div.s $f%d, $f%d, $f%d\n",
    left, left, *t);
402             } else if (sym == idivsym) {
403                 fprintf(fp, "div.s $f%d, $f%d, $f%d\n",
```

```
403 left, left, *t);
404             } else if (sym == modsym) {
405                 fprintf(fp, "div $t%d, $t%d\n", left, *t);
406                 fprintf(fp, "mfhi $t%d\n", left);
407             } else {
408                 return pcerror("Unknown multiplication
    symbol: %s\n", symtypestr[sym]);
409             }
410         } else {
411             return pcerror("Cannot multop on type %d\n", *
    type);
412         }
413     }
414 
415     *t = left;
416     return 1;
417 }
418 
419 int pcicg_simple_expression(AST *ast, symtype *type, int *
    t) {
420     ASTchild *cur;
421     pcsym sym;
422     int left;
423     symtype termtype;
424     symtype termtype2;
425 
426     printf("===== ENTERING pcicg_simple_expression =====\n
    ");
427 
428     /* grab the first term */
429     cur = ast->head;
430     EXPECTICG(cur->ast, termasm);
431     if (!pcicg_term(cur->ast, &termtype, t)) return 0;
432 
433     /* if we didn't request a type, assign it to the
    current value's type */
434     if (type == NULL || *type == notype) *type = termtype;
435 
436     /* update our left-most value for chaining */
437     left = *t;
438     *t += 1;
439 
440     /* go through all of our addops */
441     while (cur->next) {
442         cur = cur->next;
443         EXPECTICG(cur->ast, addasm);
444         sym = cur->ast->sym;
445 
446         cur = cur->next;
447         EXPECTICG(cur->ast, termasm);
```

```
448         if (!pcicg_term(cur->ast, &termtype2, t)) return 0
    ;
449
450         /* write out the add or subtract */
451         if (!pcicg_convert(termtype, termtype2, *t))
    return 0;
452         if (termtype == integertype) {
453             if (sym == addsym) fprintf(fp, "add $t%d, $t%d
    , $t%d\n", left, left, *t);
454             else if (sym == minussym) fprintf(fp, "sub $t%
    d, $t%d, $t%d\n", left, left, *t);
455             else if (sym == orsym) fprintf(fp, "or $t%d,
    $t%d, $t%d\n", left, left, *t);
456             else return pcerror("Incompatible addop: %d\n"
    , sym);
457         } else if (termtype == realtype) {
458             if (sym == addsym) fprintf(fp, "add.s $f%d, $f
    %d, $f%d\n", left, left, *t);
459             else if (sym == minussym) fprintf(fp, "sub.s
    $f%d, $f%d, $f%d\n", left, left, *t);
460             else return pcerror("Incompatible addop: %d\n"
    , sym);
461         } else {
462             return pcerror("Cannot addop on type: %s\n",
    symtypestr[termtype]);
463         }
464     }
465
466     /* reupdate the t to reflect our return value (convert
     if needed) */
467     *t = left;
468     return pcicg_convert(*type, termtype, left);
469 }
470
471 int pcicg_assign(AST *ast, int *t) {
472     ASTchild *cur;
473     symentry *entry;
474     int offset;
475     symtype type = notype;
476
477     printf("===== ENTERING pcicg_assign =====\n");
478
479     /* load our entry for storage */
480     cur = ast->head;
481     EXPECTICG(cur->ast, idasm);
482     if (!(entry = pclookupsym_entry(cur->ast->name, &
    offset)))
483         return pcerror("Unable to find variable: %s\n",
    cur->ast->name);
484
```

```
485     /* make sure the entry isn't constant */
486     if (entry->bconst) return pcerror("Cannot alter
    constant variable: %s\n", entry->name);
487
488     /* evaluate the expression */
489     cur = cur->next;
490     EXPECTICG(cur->ast, exprasm);
491     cur = cur->ast->head;
492     EXPECTICG(cur->ast, simexprasm);
493
494     /* if we see function, we assume return */
495     if (entry->type == functiontype) type = entry->
    returntype;
496     else type = entry->type;
497
498     if (!pcicg_simple_expression(cur->ast, &type, t))
    return 0;
499
500     /* actually assign the value */
501     switch (type) {
502         case integertype:
503             if (entry->type == functiontype) fprintf(fp, "
    move $v0, $t%d\n", *t);
504             else fprintf(fp, "sw $t%d, %d($sp)\n", *t,
    offset);
505             break;
506         case realtype:
507             if (entry->type == functiontype) fprintf(fp, "
    move $f10, $t%d\n", *t);
508             else fprintf(fp, "sw.s $f%d, %d($sp)\n", *t,
    offset);
509             break;
510         case chartype:
511             if (entry->type == functiontype) fprintf(fp, "
    move $v0, $t%d\n", *t);
512             else {
513                 fprintf(fp, "sw $0, %d($sp)\n", offset);
514                 fprintf(fp, "sb $t%d, %d($sp)\n", *t,
    offset);
515             }
516             break;
517
518         default:
519             return pcerror("Cannot assign type: %s\n",
    symtypestr[type]);
520     }
521
522     return 1;
523 }
524
```

```
525 int pcicg_if(AST *ast, int *t) {
526     ASTchild *expr;
527     ASTchild *astthen;
528     ASTchild *astelse;
529     int left;
530     pcsym relop;
531     char setcmd[4];
532     char label[20];
533     symtype type = integertype;
534 
535     printf("===== ENTERING pcicg_if =====\n");
536 
537     expr = ast->head;
538     left = *t;
539     *t = *t + 1;
540 
541     /* setup our label */
542     snprintf(label, 20, "Lif%d", ifcount++);
543 
544     /* setup our children */
545     EXPECTICG(expr->ast, exprasm);
546     astthen = expr->next;
547     astelse = astthen->next;
548     expr = expr->ast->head;
549 
550     /* part 1 */
551     EXPECTICG(expr->ast, simexprasm);
552     if (!(pcicg_simple_expression(expr->ast, &type, &left
    ))) return 0;
553     expr = expr->next;
554 
555     /* operator */
556     EXPECTICG(expr->ast, relasm);
557     relop = expr->ast->sym;
558     expr = expr->next;
559 
560     /* part 2 */
561     EXPECTICG(expr->ast, simexprasm);
562     if (!(pcicg_simple_expression(expr->ast, &type, t)))
    return 0;
563 
564     /* comparison */
565     switch (relop) {
566         case ltsym:
567             strcpy(setcmd, "blt");
568             break;
569         case ltesym:
570             strcpy(setcmd, "ble");
571             break;
572         case neqsym:
```

```
573             strcpy(setcmd, "bne");
574             break;
575         case gtsym:
576             strcpy(setcmd, "bgt");
577             break;
578         case gtesym:
579             strcpy(setcmd, "bge");
580             break;
581         case eqsym:
582             strcpy(setcmd, "beq");
583             break;
584         default:
585             return pcerror("Unknown relop: %s\n", pcsymstr
    [relop]);
586     }
587 
588     /* branch to the "then" part if relop holds */
589     fprintf(fp, "\n%s $t%d, $t%d, %s\n", setcmd, left, *t
    , label);
590 
591     /* do the else first for branching purposes */
592     if (astelse)
593         if (accept(astelse->ast, statementasm) && !
    pcicg_statement(astelse->ast)) return 0;
594 
595     /* branch past the "then" part if relop didn't hold (
    or after else) */
596     fprintf(fp, "b %send\n", label);
597 
598     /* make sure we have a thenpart */
599     EXPECTICG(astthen->ast, statementasm);
600     fprintf(fp, "\n%s: ", label);
601     if (!(pcicg_statement(astthen->ast))) return 0;
602 
603     /* print our end label */
604     fprintf(fp, "%send:\n\n", label);
605     *t = left;
606 
607     return 1;
608 }
609 
610 int pcicg_while(AST *ast, int *t) {
611     ASTchild *cur;
612     char setcmd[4];
613     char label[20];
614     symtype type = integertype;
615     pcsym relop;
616     int left;
617 
618     printf("===== ENTERING pcicg_while =====\n");
```

```
619
620     /* setup our label */
621     snprintf(label, 20, "Lwhile%d", whilecount++);
622
623     EXPECTICG(ast, whileasm);
624     cur = ast->head;
625
626     EXPECTICG(cur->ast, exprasm);
627     cur = cur->ast->head;
628
629     left = *t;
630     *t = *t + 1;
631
632     fprintf(fp, "\n%s: ", label);
633
634     /* part 1 */
635     EXPECTICG(cur->ast, simexprasm);
636     if (!(pcicg_simple_expression(cur->ast, &type, &left
    ))) return 0;
637     cur = cur->next;
638
639     /* operator */
640     EXPECTICG(cur->ast, relasm);
641     relop = cur->ast->sym;
642     cur = cur->next;
643
644     /* part 2 */
645     EXPECTICG(cur->ast, simexprasm);
646     if (!(pcicg_simple_expression(cur->ast, &type, t)))
    return 0;
647
648     /* comparison (use opposite here to "break out") */
649     switch (relop) {
650         case ltsym:
651             strcpy(setcmd, "bge");
652             break;
653         case ltesym:
654             strcpy(setcmd, "bgt");
655             break;
656         case neqsym:
657             strcpy(setcmd, "beq");
658             break;
659         case gtsym:
660             strcpy(setcmd, "ble");
661             break;
662         case gtesym:
663             strcpy(setcmd, "blt");
664             break;
665         case eqsym:
666             strcpy(setcmd, "bne");
```

```
667             break;
668         default:
669             return pcerror("Unknown relop: %s\n", pcsymstr
    [relop]);
670     }
671 
672     /* branch past the main part if relop didn't hold (or
    after else) */
673     fprintf(fp, "%s $t%d, $t%d, %send\n", setcmd, left, *t
    , label);
674 
675     /* make sure we have a statement */
676     cur = ast->head->next;
677     EXPECTICG(cur->ast, statementasm);
678     if (!(pcicg_statement(cur->ast))) return 0;
679 
680     /* loop back to the top of the while loop */
681     fprintf(fp, "j %s\n", label);
682 
683     /* print our end label */
684     fprintf(fp, "%send:\n\n", label);
685     *t = left;
686 
687     return 1;
688 }
689 
690 int pcicg_write(AST *ast, int *t) {
691     ASTchild *cur;
692     symtype type = notype;
693 
694     printf("===== ENTERING pcicg_write =====\n");
695 
696     /* error checking */
697     if (!accept(ast, writeasm) && !accept(ast, writelnasm
    )) return 0;
698     cur = ast->head;
699     EXPECTICG(cur->ast, exprasm);
700     cur = cur->ast->head;
701 
702     /* process the statement */
703     EXPECTICG(cur->ast, simexprasm);
704     if (!pcicg_simple_expression(cur->ast, &type, t))
    return 0;
705 
706     /* determine the write based on type */
707     switch (type) {
708         case chartype:
709             fprintf(fp, "li $a0, 0\n");
710             fprintf(fp, "move $a0, $t%d\n", *t);
711             fprintf(fp, "li $v0, 11\n");
```

```
712             break;
713
714         case integertype:
715             fprintf(fp, "move $a0, $t%d\n", *t);
716             fprintf(fp, "li $v0, 1\n");
717             break;
718
719         case realtype:
720             fprintf(fp, "move.s $f12, $f%d\n", *t);
721             fprintf(fp, "li $v0, 2\n");
722             break;
723
724         case stringtype:
725             fprintf(fp, "move $a0, $t%d\n", *t);
726             fprintf(fp, "li $v0, 4\n");
727             break;
728
729         default:
730             return pcerror("Cannot write for type: %s\n",
    symtypestr[type]);
731     }
732
733     /* execute the syscall */
734     fprintf(fp, "syscall\n");
735
736     /* add a new line if ln is used */
737     if (ast->node == writelnasm)
738         fprintf(fp, "li $a0, 10\nli $v0, 11\nsyscall\n");
739
740     return 1;
741 }
742
743 int pcicg_read(AST *ast, int *t) {
744     ASTchild *cur;
745     symentry *entry;
746     symtype type = notype;
747     int offset;
748
749     printf("===== ENTERING pcicg_read =====\n");
750
751     cur = ast->head;
752     EXPECTICG(cur->ast, idasm);
753
754     if (!(entry = pclookupsym_entry(cur->ast->name, &
    offset)))
755         return pcerror("Unable to load entry: %s\n", cur->
    ast->name);
756
757     switch (entry->type) {
758         case integertype:
```

```
759             fprintf(fp, "li $v0, 5\nsyscall\nsw $v0, %d(
    $sp)\n", offset);
760             break;
761 
762         case chartype:
763             fprintf(fp, "li $v0, 12\nsyscall\nsw $v0, %d(
    $sp)\n", offset);
764             break;
765 
766         case realtype:
767             fprintf(fp, "li $v0, 6\nsyscall\nsw.s $f0, %d(
    $sp)\n", offset);
768             break;
769 
770         default:
771             return pcerror("Unable to read type: %s\n",
    symtypestr[entry->type]);
772     }
773 
774     return 1;
775 }
776 
777 int pcicg_proccall(AST *ast, int *t) {
778     ASTchild *cur;
779     ASTchild *pcur;
780     symparam *param;
781     symentry *entry;
782     symtype type = notype;
783     int a = 0;
784 
785     printf("===== ENTERING pcicg_proccall =====\n");
786 
787     EXPECTICG(ast, proccallasm);
788     if (!(entry = pclookupsym(ast->name)))
789         return pcerror("Unable to find procedure: %s\n",
    ast->name);
790 
791     cur = ast->head;
792     param = entry->params;
793     while (cur) {
794         if (!param) return pcerror("[%d] Too many
    parameters - %d\n", entry->lineno, a);
795 
796         EXPECTICG(cur->ast, exprasm);
797         if (!(pcur = cur->ast->head)) return 0;
798         EXPECTICG(pcur->ast, simexprasm);
799 
800         type = param->entry->type;
801         if (!(pcicg_simple_expression(pcur->ast, &type, t
    ))) return 0;
```

```
802
803         /* store the value in the a register for passing
     */
804         switch (type) {
805             case integertype:
806             case chartype:
807                 fprintf(fp, "move $a%d, $t%d\n", a, *t);
808                 break;
809
810             case realtype:
811                 fprintf(fp, "move $f1%d, $f%d\n", a, *t);
812                 break;
813
814             default:
815                 return pcerror("Unsupported param type: %s
    \n", symtypestr[type]);
816         }
817
818         /* increment everything */
819         param = param->next;
820         cur = cur->next;
821         ++a;
822     }
823
824     if (param) return pcerror("[%d] Too few parameters - %
    d\n", entry->lineno, a);
825
826     /* make the function call */
827     fprintf(fp, "jal %s\n", entry->name);
828
829     return 1;
830 }
831
832 int pcicg_funccall(AST *ast, symtype *returntype, int *t
    ) {
833     ASTchild *cur;
834     ASTchild *pcur;
835     symparam *param;
836     symentry *entry;
837     symtype type = notype;
838     int a = 0;
839
840     printf("===== ENTERING pcicg_funccall =====\n");
841
842     EXPECTICG(ast, funccallasm);
843     if (!(entry = pclookupsym(ast->name)))
844         return pcerror("Unable to find function: %s\n",
    ast->name);
845
846     cur = ast->head;
```

```
847     param = entry->params;
848     while (cur) {
849         if (!param) return pcerror("[%d] Too many
    parameters - %d\n", entry->lineno, a);
850 
851         EXPECTICG(cur->ast, exprasm);
852         if (!(pcur = cur->ast->head)) return 0;
853         EXPECTICG(pcur->ast, simexprasm);
854 
855         type = param->entry->type;
856         if (!(pcicg_simple_expression(pcur->ast, &type, t
    ))) return 0;
857 
858         /* store the value in the a register for passing
     */
859         switch (type) {
860             case integertype:
861             case chartype:
862                 fprintf(fp, "move $a%d, $t%d\n", a, *t);
863                 break;
864 
865             case realtype:
866                 fprintf(fp, "move $f1%d, $f%d\n", a, *t);
867                 break;
868 
869             default:
870                 return pcerror("Unsupported param type: %s
    \n", symtypestr[type]);
871         }
872 
873         /* increment everything */
874         param = param->next;
875         cur = cur->next;
876         ++a;
877     }
878 
879     if (param) return pcerror("[%d] Too few parameters - %
    d\n", entry->lineno, a);
880 
881     /* make the function call */
882     fprintf(fp, "jal %s\n", entry->name);
883 
884     /* store the return value in the register */
885     switch (entry->returntype) {
886         case integertype:
887         case chartype:
888             fprintf(fp, "move $t%d, $v0\n", *t);
889             break;
890 
891         case realtype:
```

```
892             fprintf(fp, "move $f%d, $f0\n", *t);
893             break;
894 
895         default:
896             return pcerror("Unreturnable type: %s\n",
    symtypestr[entry->returntype]);
897     }
898 
899     /* convert if needed */
900     if (returntype && *returntype != notype) *returntype
     = entry->returntype;
901 
902     return pcicg_convert(*returntype, entry->returntype, *
    t);
903 }
904 
905 int pcicg_statement(AST *ast) {
906     ASTchild *cur;
907     int t = 0;
908 
909     printf("===== ENTERING pcicg_statement =====\n");
910 
911     cur = ast->head;
912     while (cur) {
913         t = 0;
914         switch (cur->ast->node) {
915             case assignasm:
916                 if (!pcicg_assign(cur->ast, &t)) return 0;
917                 break;
918 
919             case ifasm:
920                 if (!pcicg_if(cur->ast, &t)) return 0;
921                 break;
922 
923             case whileasm:
924                 if (!pcicg_while(cur->ast, &t)) return 0;
925                 break;
926 
927             case writeasm:
928             case writelnasm:
929                 if (!pcicg_write(cur->ast, &t)) return 0;
930                 break;
931 
932             case readasm:
933             case readlnasm:
934                 if (!pcicg_read(cur->ast, &t)) return 0;
935                 break;
936 
937             case proccallasm:
938                 if (!pcicg_proccall(cur->ast, &t)) return
```

```
938 0;
939                 break;
940 
941             case funccallasm:
942                 if (!pcicg_funccall(cur->ast, NULL, &t))
    return 0;
943                 break;
944 
945             default:
946                 pcerror("Unhandled statement type.: %s\n"
    , astnodestr[cur->ast->node]);
947         }
948 
949         cur = cur->next;
950     }
951 
952     return 1;
953 }
954 
955 int pcicg_block(AST *ast, symentry *entry, const char *
    label, ASTchild *params) {
956     ASTchild *cur;
957     int size;
958 
959     printf("===== ENTERING pcicg_block =====\n");
960 
961     /* grab the const and vars first */
962     cur = ast->head;
963 
964     /* entry the new block */
965     if (entry) {
966         if (entry->type != programtype && !
    pcenterscope_nocreate(entry)) return 0;
967 
968         if (entry->type == programtype) size = pcrootsize
    ();
969         else size = entry->size + 4;
970 
971         /* assign our stack and jump to the body */
972         fprintf(fp, "\n%s: addi $sp, $sp, -%d\n", label,
    size);
973 
974         /* store the ra for usage later */
975         if (entry->type != programtype) fprintf(fp, "sw
    $ra, %d($sp)\n", entry->size);
976 
977         /* skip past params */
978         if (accept(cur->ast, paramasm)) cur = cur->next;
979 
980         if (accept(cur->ast, constasm)) {
```

```
 981             if (!pcicg_const(cur->ast)) return 0;
 982             cur = cur->next;
 983             fprintf(fp, "\n");
 984         }
 985         if (accept(cur->ast, varasm)) {
 986             if (!pcicg_var(cur->ast)) return 0;
 987             cur = cur->next;
 988             fprintf(fp, "\n");
 989         }
 990 
 991         /* assign parameters */
 992         if (params) {
 993             int acur = 0;
 994             symentry *pentry;
 995             int offset = 0;
 996 
 997             while (params) {
 998                 if (!params->ast->name) {
 999                     return pcerror("Missing parameter
     name for #%d\n", acur);
1000                 }
1001 
1002                 if (!(pentry = pclookupsym_entry(params->
     ast->name, &offset)))
1003                     return pcerror("Unable to find param
     : %s\n", params->ast->name);
1004 
1005                 /* save onto the stack, based on type */
1006                 switch (pentry->type) {
1007                     case integertype:
1008                         fprintf(fp, "sw $a%d, %d($sp)\n"
     , acur, offset);
1009                         break;
1010                     case realtype:
1011                         fprintf(fp, "sw.s $a%d, %d($sp)\n
     ", acur, offset);
1012                         break;
1013                     case chartype:
1014                         fprintf(fp, "sw $0, %d($sp)\n",
     offset);
1015                         fprintf(fp, "sb $a%d, %d($sp)\n"
     , acur, offset);
1016                         break;
1017                     default:
1018                         return pcerror("Cannot assign
     type: %s\n", symtypestr[pentry->type]);
1019                 }
1020 
1021                 params = params->next;
1022                 ++acur;
```

```
1023            }
1024        }
1025
1026        /* jump to the body */
1027        fprintf(fp, "j %sbody\n\n", label);
1028    }
1029
1030
1031    /* now do the methods */
1032    while (cur) {
1033        if (accept(cur->ast, procedureasm)) {
1034            if (!pcicg_procedure(cur->ast)) return 0;
1035        } else if (accept(cur->ast, functionasm)) {
1036            if (!pcicg_function(cur->ast)) return 0;
1037        } else if (accept(cur->ast, statementasm)) {
1038            break;
1039        } else {
1040            return pcerror("Unexpected node: %s\n",
    astnodestr[cur->ast->node]);
1041        }
1042
1043        cur = cur->next;
1044    }
1045
1046    /* now do the statement */
1047    EXPECTICG(cur->ast, statementasm);
1048
1049    /* process our body */
1050    if (entry) fprintf(fp, "%sbody: ", label);
1051    if (!pcicg_statement(cur->ast)) return 0;
1052
1053    /* exit the scope */
1054    if (entry && entry->type != programtype) return
    pcleavescope();
1055    return 1;
1056 }
1057
1058 int pcicg_program(AST *ast) {
1059    symentry *entry;
1060
1061    printf("===== ENTERING pcicg_program =====\n");
1062
1063    EXPECTICG(ast, programasm);
1064    fprintf(fp, ".data\n\n.text\n");
1065
1066    /* grab the entry for this block */
1067    if (!(entry = pclookupsym(ast->name))) return pcerror
    ("Unable to find program.\n");
1068    if (!pcicg_block(ast, entry, "main", NULL)) return 0;
1069
```

```
1070     /* print the program end syscall */
1071     fprintf(fp, "li $v0, 10\nsyscall");
1072     return 1;
1073 }
1074
1075 int pcicg_start(FILE *fpo) {
1076     printf("===== ENTERING pcicg_start =====\n");
1077
1078     if (!fpo) {
1079         printf("\nFILE REQUIRED FOR ICG!\n");
1080         return 0;
1081     }
1082
1083     /* setup the globals */
1084     fp = fpo;
1085     ifcount = whilecount = forcount = 0;
1086
1087     return pcicg_program(astroot);
1088 }
1089
```

```
 1 /*
 2  * @author Derek Trom
 3  * @author Elena Corpus
 4  * compiler.h is the main entry point of the program and is
    responsible
 5  * for handling user input and running the other portions
   of the compiler,
 6  * such as the scanner and parser.
 7 */
 8
 9 #ifndef ICG_H
10 #define ICG_H
11
12 #include "ast.h"
13
14 int pcicg_start(FILE *fpo);
15
16 #endif /* ICG_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 icg.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 run:    $(PROJNAME)
51      ./$(PROJNAME) tests/goodTest.pas
52
```

```
 1 .data
 2
 3 .text
 4
 5 main: addi $sp, $sp, -108
 6 li $t0, 72
 7 sb $t0, 0($sp)
 8 li $t0, 111
 9 sb $t0, 1($sp)
10 li $t0, 119
11 sb $t0, 2($sp)
12 li $t0, 32
13 sb $t0, 3($sp)
14 li $t0, 109
15 sb $t0, 4($sp)
16 li $t0, 97
17 sb $t0, 5($sp)
18 li $t0, 110
19 sb $t0, 6($sp)
20 li $t0, 121
21 sb $t0, 7($sp)
22 li $t0, 32
23 sb $t0, 8($sp)
24 li $t0, 70
25 sb $t0, 9($sp)
26 li $t0, 105
27 sb $t0, 10($sp)
28 li $t0, 98
29 sb $t0, 11($sp)
30 li $t0, 111
31 sb $t0, 12($sp)
32 li $t0, 110
33 sb $t0, 13($sp)
34 li $t0, 97
35 sb $t0, 14($sp)
36 li $t0, 99
37 sb $t0, 15($sp)
38 li $t0, 99
39 sb $t0, 16($sp)
40 li $t0, 105
41 sb $t0, 17($sp)
42 li $t0, 32
43 sb $t0, 18($sp)
44 li $t0, 110
45 sb $t0, 19($sp)
46 li $t0, 117
47 sb $t0, 20($sp)
48 li $t0, 109
49 sb $t0, 21($sp)
50 li $t0, 98
```

```
 51 sb $t0, 22($sp)
 52 li $t0, 101
 53 sb $t0, 23($sp)
 54 li $t0, 114
 55 sb $t0, 24($sp)
 56 li $t0, 115
 57 sb $t0, 25($sp)
 58 li $t0, 63
 59 sb $t0, 26($sp)
 60 li $t0, 32
 61 sb $t0, 27($sp)
 62 li $t0, 0
 63 sb $t0, 28($sp)
 64 li $t0, 73
 65 sb $t0, 32($sp)
 66 li $t0, 110
 67 sb $t0, 33($sp)
 68 li $t0, 108
 69 sb $t0, 34($sp)
 70 li $t0, 105
 71 sb $t0, 35($sp)
 72 li $t0, 110
 73 sb $t0, 36($sp)
 74 li $t0, 101
 75 sb $t0, 37($sp)
 76 li $t0, 32
 77 sb $t0, 38($sp)
 78 li $t0, 87
 79 sb $t0, 39($sp)
 80 li $t0, 104
 81 sb $t0, 40($sp)
 82 li $t0, 105
 83 sb $t0, 41($sp)
 84 li $t0, 108
 85 sb $t0, 42($sp)
 86 li $t0, 101
 87 sb $t0, 43($sp)
 88 li $t0, 45
 89 sb $t0, 44($sp)
 90 li $t0, 76
 91 sb $t0, 45($sp)
 92 li $t0, 111
 93 sb $t0, 46($sp)
 94 li $t0, 111
 95 sb $t0, 47($sp)
 96 li $t0, 112
 97 sb $t0, 48($sp)
 98 li $t0, 0
 99 sb $t0, 49($sp)
100 li $t0, 70
```

```
101 sb $t0, 52($sp)
102 li $t0, 117
103 sb $t0, 53($sp)
104 li $t0, 110
105 sb $t0, 54($sp)
106 li $t0, 99
107 sb $t0, 55($sp)
108 li $t0, 116
109 sb $t0, 56($sp)
110 li $t0, 105
111 sb $t0, 57($sp)
112 li $t0, 111
113 sb $t0, 58($sp)
114 li $t0, 110
115 sb $t0, 59($sp)
116 li $t0, 97
117 sb $t0, 60($sp)
118 li $t0, 108
119 sb $t0, 61($sp)
120 li $t0, 32
121 sb $t0, 62($sp)
122 li $t0, 87
123 sb $t0, 63($sp)
124 li $t0, 104
125 sb $t0, 64($sp)
126 li $t0, 105
127 sb $t0, 65($sp)
128 li $t0, 108
129 sb $t0, 66($sp)
130 li $t0, 101
131 sb $t0, 67($sp)
132 li $t0, 45
133 sb $t0, 68($sp)
134 li $t0, 76
135 sb $t0, 69($sp)
136 li $t0, 111
137 sb $t0, 70($sp)
138 li $t0, 111
139 sb $t0, 71($sp)
140 li $t0, 112
141 sb $t0, 72($sp)
142 li $t0, 0
143 sb $t0, 73($sp)
144 li $t0, 82
145 sb $t0, 76($sp)
146 li $t0, 101
147 sb $t0, 77($sp)
148 li $t0, 99
149 sb $t0, 78($sp)
150 li $t0, 117
```

```
151 sb $t0, 79($sp)
152 li $t0, 114
153 sb $t0, 80($sp)
154 li $t0, 115
155 sb $t0, 81($sp)
156 li $t0, 105
157 sb $t0, 82($sp)
158 li $t0, 118
159 sb $t0, 83($sp)
160 li $t0, 101
161 sb $t0, 84($sp)
162 li $t0, 0
163 sb $t0, 85($sp)
164
165 sw $0, 88($sp)
166 sw $0, 92($sp)
167 sw $0, 96($sp)
168 sw $0, 100($sp)
169 sw $0, 104($sp)
170
171 j mainbody
172
173
174 recursivefibonacci: addi $sp, $sp, -16
175 sw $ra, 12($sp)
176 sw $a0, 0($sp)
177 sw $a1, 4($sp)
178 sw $a2, 8($sp)
179 j recursivefibonaccibody
180
181 recursivefibonaccibody: lw $t0, 0($sp)
182 li $t1, 0
183
184 bgt $t0, $t1, Lif0
185 b Lif0end
186
187 Lif0: li $t0, 32
188 li $a0, 0
189 move $a0, $t0
190 li $v0, 11
191 syscall
192 lw $t0, 4($sp)
193 li $t1, 0
194
195 beq $t0, $t1, Lif1
196 lw $t0, 8($sp)
197 li $t1, 0
198
199 beq $t0, $t1, Lif2
200 lw $t0, 4($sp)
```

```
201 lw $t1, 8($sp)
202 add $t0, $t0, $t1
203 move $a0, $t0
204 li $v0, 1
205 syscall
206 lw $t0, 0($sp)
207 li $t1, 1
208 sub $t0, $t0, $t1
209 move $a0, $t0
210 lw $t0, 8($sp)
211 move $a1, $t0
212 lw $t0, 4($sp)
213 lw $t1, 8($sp)
214 add $t0, $t0, $t1
215 move $a2, $t0
216 jal recursivefibonacci
217 b Lif2end
218
219 Lif2: li $t0, 1
220 move $a0, $t0
221 li $v0, 1
222 syscall
223 lw $t0, 0($sp)
224 li $t1, 1
225 sub $t0, $t0, $t1
226 move $a0, $t0
227 li $t0, 1
228 move $a1, $t0
229 li $t0, 1
230 move $a2, $t0
231 jal recursivefibonacci
232 Lif2end:
233
234 b Lif1end
235
236 Lif1: li $t0, 1
237 move $a0, $t0
238 li $v0, 1
239 syscall
240 lw $t0, 0($sp)
241 li $t1, 1
242 sub $t0, $t0, $t1
243 move $a0, $t0
244 li $t0, 1
245 move $a1, $t0
246 li $t0, 0
247 move $a2, $t0
248 jal recursivefibonacci
249 Lif1end:
250
```

```
Lif0end:

lw $ra, 12($sp)
addi $sp, $sp, 16
jr $ra


nextfibonacci: addi $sp, $sp, -12
sw $ra, 8($sp)
sw $a0, 0($sp)
sw $a1, 4($sp)
j nextfibonaccibody

nextfibonaccibody: lw $t0, 0($sp)
li $t1, 0

beq $t0, $t1, Lif3
lw $t0, 4($sp)
li $t1, 0

beq $t0, $t1, Lif4
lw $t0, 0($sp)
lw $t1, 4($sp)
add $t0, $t0, $t1
move $v0, $t0
b Lif4end

Lif4: li $t0, 1
move $v0, $t0
Lif4end:

b Lif3end

Lif3: li $t0, 1
move $v0, $t0
Lif3end:

lw $ra, 8($sp)
addi $sp, $sp, 12
jr $ra

mainbody: la $t0, 0($sp)
move $a0, $t0
li $v0, 4
syscall
li $v0, 5
syscall
sw $v0, 88($sp)
li $t0, 32
li $a0, 0
```

```
move $a0, $t0
li $v0, 11
syscall
li $a0, 10
li $v0, 11
syscall
la $t0, 32($sp)
move $a0, $t0
li $v0, 4
syscall
li $a0, 10
li $v0, 11
syscall
li $t0, 0
sw $t0, 92($sp)

Lwhile0: lw $t0, 92($sp)
lw $t1, 88($sp)
bge $t0, $t1, Lwhile0end
lw $t0, 92($sp)
li $t1, 1

ble $t0, $t1, Lif5
lw $t0, 104($sp)
sw $t0, 96($sp)
lw $t0, 100($sp)
lw $t1, 104($sp)
add $t0, $t0, $t1
sw $t0, 104($sp)
lw $t0, 96($sp)
sw $t0, 100($sp)
li $t0, 32
li $a0, 0
move $a0, $t0
li $v0, 11
syscall
lw $t0, 104($sp)
move $a0, $t0
li $v0, 1
syscall
b Lif5end

Lif5: li $t0, 32
li $a0, 0
move $a0, $t0
li $v0, 11
syscall
li $t0, 1
move $a0, $t0
li $v0, 1
```

```
351 syscall
352 li $t0, 1
353 sw $t0, 100($sp)
354 li $t0, 1
355 sw $t0, 104($sp)
356 Lif5end:
357
358 lw $t0, 92($sp)
359 li $t1, 1
360 add $t0, $t0, $t1
361 sw $t0, 92($sp)
362 j Lwhile0
363 Lwhile0end:
364
365 li $t0, 32
366 li $a0, 0
367 move $a0, $t0
368 li $v0, 11
369 syscall
370 li $a0, 10
371 li $v0, 11
372 syscall
373 li $t0, 32
374 li $a0, 0
375 move $a0, $t0
376 li $v0, 11
377 syscall
378 li $a0, 10
379 li $v0, 11
380 syscall
381 la $t0, 52($sp)
382 move $a0, $t0
383 li $v0, 4
384 syscall
385 li $a0, 10
386 li $v0, 11
387 syscall
388 li $t0, 0
389 sw $t0, 92($sp)
390 li $t0, 0
391 sw $t0, 100($sp)
392 li $t0, 0
393 sw $t0, 104($sp)
394
395 Lwhile1: lw $t0, 92($sp)
396 lw $t1, 88($sp)
397 bge $t0, $t1, Lwhile1end
398 lw $t0, 100($sp)
399 move $a0, $t0
400 lw $t0, 104($sp)
```

```
401 move $a1, $t0
402 jal nextfibonacci
403 move $t0, $v0
404 sw $t0, 96($sp)
405 li $t0, 32
406 li $a0, 0
407 move $a0, $t0
408 li $v0, 11
409 syscall
410 lw $t0, 96($sp)
411 move $a0, $t0
412 li $v0, 1
413 syscall
414 lw $t0, 104($sp)
415 sw $t0, 100($sp)
416 lw $t0, 96($sp)
417 sw $t0, 104($sp)
418 lw $t0, 92($sp)
419 li $t1, 1
420 add $t0, $t0, $t1
421 sw $t0, 92($sp)
422 j Lwhile1
423 Lwhile1end:
424
425 li $t0, 32
426 li $a0, 0
427 move $a0, $t0
428 li $v0, 11
429 syscall
430 li $a0, 10
431 li $v0, 11
432 syscall
433 li $t0, 32
434 li $a0, 0
435 move $a0, $t0
436 li $v0, 11
437 syscall
438 li $a0, 10
439 li $v0, 11
440 syscall
441 la $t0, 76($sp)
442 move $a0, $t0
443 li $v0, 4
444 syscall
445 li $a0, 10
446 li $v0, 11
447 syscall
448 lw $t0, 88($sp)
449 move $a0, $t0
450 li $t0, 0
```

```
451 move $a1, $t0
452 li $t0, 0
453 move $a2, $t0
454 jal recursivefibonacci
455 li $v0, 10
456 syscall
457
```

```c
/*
 * @author Derek Trom
 * @author Elena Corpus
 * parser 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) {
```

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

```
 97 }
 98
 99 /* Forces two specific symbols to be found in sequence.
100
101 @param sym1 the first symbol to match against
102 @param sym2 the second symbol to match against
103 @return 1 on success; 0 otherwise
104 */
105 int
106 pcp_expect2(pcsym sym1, pcsym sym2) {
107     pcsym tmpsym = token->sym;
108
109     if (pcp_accept(sym1)) {
110         if (pcp_accept(sym2)) {
111             return 1;
112         }
113
114         return pcerror("[%u] pcp_expect: Unexpected end to
     symbol sequence: %s %s vs %s %s\n",
115                        token->lineno, pcsymstr[tmpsym],
    pcsymstr[token->sym], pcsymstr[sym1], pcsymstr[sym2]);
116     }
117
118     return pcerror("[%u] pcp_expect: Unexpected start to
    symbol sequence: %s %s vs %s %s\n",
119                    token->lineno, pcsymstr[tmpsym],
    pcsymstr[nexttoken->sym], pcsymstr[sym1], pcsymstr[sym2]);
120 }
121
122 /* Converts a symbol to a symtype used to store in the
    symbol table.
123
124 @param sym the symbol to check for type
125 @param type return value of the type
126 @return 1 on success; 0 otherwise
127 */
128 int
129 sym_to_type(pcsym sym, symtype *type) {
130     if (sym == integersym) *type = integertype;
131     else if (sym == realsym) *type = realtype;
132     else if (sym == stringsym) *type = stringtype;
133     else if (sym == charsym) *type = chartype;
134     else return pcerror("Unknown type. Arrays and custom
    types not yet supported.\n");
135
136     return 1;
137 }
138
139 /* Adds a pctoken to the end of our list.
140
```

```
141 @param tail the tail of our list
142 @param token the token to add
143 @return the new tail pointer
144 */
145 pctokenlist *
146 tokenlist_add(pctokenlist *tail, pctoken *token) {
147     pctokenlist *next;
148 
149     if (!(next = malloc(sizeof(*next)))) {
150         pcerror("Out of memory.\n");
151         return NULL;
152     }
153     next->token = token;
154     next->next = NULL;
155 
156     tail->next = next;
157     return next;
158 }
159 
160 pctoken *
161 pcp_const_no_id(symtype *type) {
162     if (token->sym == integernosym) *type = integertype;
163     else if (token->sym == realnosym) *type = realtype;
164     else if (token->sym == charvalsym) *type = chartype;
165     else if (token->sym == stringvalsym) *type =
    stringtype;
166     else return NULL;
167 
168     return token;
169 }
170 
171 /* Ensures the the ord() function is called correctly.
172 
173 @return 1 on success; 0 otherwise
174 */
175 int
176 pcp_ord(AST *ast) {
177     AST *astord;
178 
179     printf("## ENTERING pcp_ord ##\n");
180 
181     EXPECT(lparensym);
182 
183     astord = AST_initialize(ordasm);
184     AST_addchild(ast, astord);
185 
186     if (!pcp_expression(astord)) return 0;
187 
188     EXPECT(rparensym);
189 
```

```
190     return 1;
191 }
192 
193 /* Ensures the the chr() function is called correctly.
194 
195 @return 1 on success; 0 otherwise
196 */
197 int
198 pcp_chr(AST *ast) {
199     AST *astchr;
200 
201     printf("## ENTERING pcp_chr ##\n");
202 
203     EXPECT(lparensym);
204 
205     astchr = AST_initialize(chrasm);
206     AST_addchild(ast, astchr);
207 
208     if (!pcp_expression(astchr)) return 0;
209 
210     EXPECT(rparensym);
211 
212     return 1;
213 }
214 
215 int
216 pcp_factor(AST *ast) {
217     AST *astfactor;
218     AST *astother;
219     pctoken *ltoken;
220 
221     printf("## ENTERED pcp_factor ##\n");
222 
223     astfactor = AST_initialize(factorasm);
224     AST_addchild(ast, astfactor);
225 
226     if (pcp_accept(notsym)) {
227         astother = AST_initialize(notasm);
228         AST_addchild(astfactor, astother);
229         return pcp_factor(astfactor);
230     }
231 
232     if (pcp_accept(idsym)) {
233         ltoken = lasttoken;
234         if (pcp_accept(lparensym)) {
235             return pcp_application(astfactor, ltoken);
236         }
237         if (pcp_accept(lbracksym)) {
238             return pcerror("Arrays not yet supported.\n");
239         } else {
```

```
240             astother = AST_initialize(idasm);
241             astother->name = strdup(ltoken->val.id);
242             AST_addchild(astfactor, astother);
243         }
244 
245         return 1;
246     }
247 
248     if (pcp_accept(ordsym)) {
249         return pcp_ord(astfactor);
250     }
251 
252     if (pcp_accept(chrsym)) {
253         return pcp_chr(astfactor);
254     }
255 
256     if (pcp_accept(lparensym)) {
257         if (!pcp_expression(astfactor)) return 0;
258 
259         pcp_expect(rparensym);
260 
261         return 1;
262     }
263 
264     /* see if we have an inline 'constant' value */
265     if (token->sym == integernosym || token->sym ==
    realnosym || token->sym == stringvalsym ||
266         token->sym == charvalsym) {
267         astother = AST_initialize(valasm);
268         astother->sym = token->sym;
269 
270         if (token->sym == stringvalsym) {
271             astother->val.str = strdup(token->val.str);
272         } else {
273             astother->val = token->val;
274         }
275 
276         AST_addchild(astfactor, astother);
277         NEXTTOKEN();
278         return 1;
279     }
280 
281     /* failed all our branches */
282     return 0;
283 }
284 
285 int
286 pcp_term(AST *ast) {
287     AST *astterm;
288     AST *astmult;
```

```
289
290     printf("## ENTERED pcp_term ##\n");
291
292     astterm = AST_initialize(termasm);
293     AST_addchild(ast, astterm);
294
295     if (!pcp_factor(astterm)) return 0;
296
297     /* keep doing all the multiplicitive arithmetic */
298     while (pcp_accept(multsym) || pcp_accept(idivsym) ||
    pcp_accept(divsym) || pcp_accept(andsym)) {
299         astmult = AST_initialize(multasm);
300         astmult->sym = lasttoken->sym;
301         AST_addchild(astterm, astmult);
302
303         if (!pcp_factor(astterm)) return 0;
304     }
305
306     return 1;
307 }
308
309 int
310 pcp_simple_expression(AST *ast) {
311     AST *astsimexpr;
312     AST *astaddasm;
313
314     printf("## ENTERED pcp_simple_expression ##\n");
315
316     astsimexpr = AST_initialize(simexprasm);
317     AST_addchild(ast, astsimexpr);
318
319     if (!pcp_term(astsimexpr)) return 0;
320
321     /* keep doing all the additional arithemetic */
322     while (pcp_accept(addsym) || pcp_accept(minussym) ||
    pcp_accept(orsym)) {
323         astaddasm = AST_initialize(addasm);
324         astaddasm->sym = lasttoken->sym;
325         AST_addchild(astsimexpr, astaddasm);
326
327         if (!pcp_term(astsimexpr)) return 0;
328     }
329
330     /* skip next token */
331     /*NEXTTOKEN();*/
332     return 1;
333 }
334
335 int
336 pcp_expression(AST *ast) {
```

```
337     AST *astexpr;
338     AST *astrel;
339 
340     printf("## ENTERED pcp_expression ##\n");
341 
342     astexpr = AST_initialize(exprasm);
343     AST_addchild(ast, astexpr);
344 
345     if (!pcp_simple_expression(astexpr)) return 0;
346 
347     /* see if this is relational */
348     if (pcp_accept(eqsym) || pcp_accept(neqsym) ||
    pcp_accept(ltsym) || pcp_accept(ltesym) || pcp_accept(
    gtesym) ||
349         pcp_accept(gtsym)) {
350         astrel = AST_initialize(relasm);
351         astrel->sym = lasttoken->sym;
352         AST_addchild(astexpr, astrel);
353         return pcp_simple_expression(astexpr);
354     }
355 
356     return 1;
357 }
358 
359 /*
360 int
361 pcp_for(AST *ast) {
362     AST *astfor;
363 
364     printf("## ENTERED pcp_for ##\n");
365 
366     astfor = AST_initialize(forasm);
367     AST_addchild(ast, astfor);
368 
369     if (!pcp_expression(astwhile)) return 0;
370 
371     EXPECT(dosym);
372 
373     return pcp_statement_part(astwhile);
374     return pcp_statement_part();
375 }*/
376 
377 int
378 pcp_while(AST *ast) {
379     AST *astwhile;
380 
381     printf("## ENTERED pcp_while ##\n");
382 
383     astwhile = AST_initialize(whileasm);
384     AST_addchild(ast, astwhile);
```

```
385
386     if (!pcp_expression(astwhile)) return 0;
387
388     EXPECT(dosym);
389
390     return pcp_statement_part(astwhile);
391 }
392
393 int
394 pcp_if(AST *ast) {
395     AST *astif;
396
397     printf("## ENTERED pcp_if ##\n");
398
399     astif = AST_initialize(ifasm);
400     AST_addchild(ast, astif);
401
402     if (!pcp_expression(astif)) return 0;
403
404     EXPECT(thensym);
405
406     if (!pcp_statement_part(astif)) return 0;
407
408     if (pcp_accept(elsesym)) {
409         return pcp_statement_part(astif);
410     }
411
412     return 1;
413 }
414
415 int
416 pcp_write(AST *ast, ASTnode nodetype) {
417     AST *astwrite;
418
419     printf("## ENTERED pcp_write ##\n");
420
421     EXPECT(lparensym);
422
423     astwrite = AST_initialize(nodetype);
424     AST_addchild(ast, astwrite);
425
426     if (!pcp_expression(astwrite)) return 0;
427
428     while (pcp_accept(commasym)) {
429         if (!pcp_expression(astwrite)) return 0;
430     }
431
432     EXPECT(rparensym);
433     return 1;
434 }
```

```
435
436 int
437 pcp_read(AST *ast) {
438     AST *astread;
439     AST *astcur;
440
441     printf("## ENTERED pcp_read ##\n");
442
443     EXPECT(lparensym);
444     EXPECT(idsym);
445
446     astread = AST_initialize(readasm);
447     AST_addchild(ast, astread);
448
449     astcur = AST_initialize(idasm);
450     astcur->name = strdup(lasttoken->val.id);
451     AST_addchild(astread, astcur);
452
453     while (pcp_accept(commasym)) {
454         EXPECT(idsym);
455         astcur = AST_initialize(idasm);
456         astcur->name = strdup(lasttoken->val.id);
457         AST_addchild(astread, astcur);
458     }
459
460     EXPECT(rparensym);
461     return 1;
462 }
463
464 int
465 pcp_application(AST *ast, pctoken *ltoken) {
466     AST *astfunccall;
467     symentry *entry = pclookupsym(ltoken->val.id);
468     int params = 1;
469
470     printf("## ENTERED pcp_application ##\n");
471
472     if (!entry || entry->type != functiontype) {
473         return pcerror("Undefined ID or unexpected type.\n
    ");
474     }
475
476     astfunccall = AST_initialize(funccallasm);
477     astfunccall->name = strdup(ltoken->val.id);
478     AST_addchild(ast, astfunccall);
479
480     if (!pcp_expression(astfunccall)) return 0;
481
482     while (pcp_accept(commasym)) {
483         if (!pcp_expression(astfunccall)) return 0;
```

```
484         ++params;
485     }
486 
487     EXPECT(rparensym);
488     return 1;
489 }
490 
491 int
492 pcp_procedure_call(AST *ast, pctoken *ltoken) {
493     AST *astproccall;
494     symentry *entry = pclookupsym(ltoken->val.id);
495     int params = 1;
496 
497     printf("## ENTERED pcp_procedure_call ##\n");
498 
499     if (!entry || entry->type != proceduretype) {
500         return pcerror("Undefined ID or unexpected type.\n
    ");
501     }
502 
503     astproccall = AST_initialize(proccallasm);
504     astproccall->name = strdup(ltoken->val.id);
505     AST_addchild(ast, astproccall);
506 
507     if (!pcp_expression(astproccall)) return 0;
508 
509     while (pcp_accept(commasym)) {
510         if (!pcp_expression(astproccall)) return 0;
511         ++params;
512     }
513 
514     EXPECT(rparensym);
515     return 1;
516 }
517 
518 int
519 pcp_procedure_call_or_application(AST *ast, pctoken *
    ltoken) {
520     return pcp_procedure_call(ast, ltoken) ||
    pcp_application(ast, ltoken);
521 }
522 
523 int
524 pcp_assign(AST *ast, pctoken *ltoken) {
525     AST *astassign;
526     AST *astlval;
527     symentry *entry = pclookupsym(ltoken->val.id);
528 
529     printf("## ENTERED pcp_assign ##\n");
530 
```

```
531     if (!entry) return pcerror("Undefined ID.\n");
532     if (entry->type != functiontype && entry->type !=
    integertype && entry->type != realtype &&
533         entry->type != chartype && entry->type !=
    stringtype)
534         return pcerror("Unexpected type: %d\n", entry->
    type);
535 
536     astassign = AST_initialize(assignasm);
537     astassign->name = strdup(ltoken->val.id);
538     AST_addchild(ast, astassign);
539 
540     astlval = AST_initialize(idasm);
541     astlval->name = strdup(ltoken->val.id);
542     AST_addchild(astassign, astlval);
543 
544     return pcp_expression(astassign);
545 }
546 
547 int
548 pcp_statement(AST *ast) {
549     int success = 0;
550 
551     printf("## ENTERED pcp_statement ##\n");
552 
553     /* procedure/function call or assignment */
554     if (pcp_accept(idsym)) {
555         pctoken *oldtoken = lasttoken;
556 
557         if (pcp_accept(lparensym)) success =
    pcp_procedure_call_or_application(ast, oldtoken);
558         else if (pcp_accept(assignsym)) success =
    pcp_assign(ast, oldtoken);
559         else if (pcp_accept(lbracksym)) return pcerror("
    Arrays not yet supported.\n");
560         else return pcerror("Unexpected symbol.\n");
561     } else if (pcp_accept(readsym) || pcp_accept(readlnsym
    )) {
562         success = pcp_read(ast);
563     } else if (pcp_accept(writesym)) {
564         success = pcp_write(ast, writeasm);
565     } else if (pcp_accept(writelnsym)) {
566         success = pcp_write(ast, writelnasm);
567     } else if (pcp_accept(ifsym)) {
568         success = pcp_if(ast);
569     } else if (pcp_accept(whilesym)) {
570         success = pcp_while(ast);
571     } /*else if (pcp_accept(forsym)) {
572         success = pcp_for();
573     }*/ else if (pcp_accept(beginsym)) {
```

```
574         success = pcp_statement_part(ast);
575     } else {
576         return pcerror("Unexpected statement.\n");
577     }
578 
579     if (!success) return 0;
580 
581     /*NEXTTOKEN();*/
582     EXPECT(semicolonsym);
583 
584     return 1;
585 }
586 
587 int
588 pcp_statement_part(AST *ast) {
589     AST *aststatement;
590 
591     printf("## ENTERING pcp_statement_part ##\n");
592 
593     EXPECT(beginsym);
594 
595     aststatement = AST_initialize(statementasm);
596     AST_addchild(ast, aststatement);
597 
598     /* go through all the statements until end */
599     while (!pcp_accept(endsym)) {
600         if (!pcp_statement(aststatement)) return 0;
601     }
602 
603     return 1;
604 }
605 
606 int
607 pcp_formal_parameters(AST *ast) {
608     pctokenlist tokens = {NULL, NULL};
609     pctokenlist *tail = NULL;
610     pctokenlist *cur;
611     pctoken *val;
612     symtype type;
613     AST *astparam;
614     AST *astcur;
615 
616     printf("## ENTERING pcp_formal_parameters ##\n");
617 
618     EXPECT(idsym);
619     tokens.token = lasttoken;
620     tail = &tokens;
621 
622     astparam = AST_initialize(paramasm);
623     AST_addchild(ast, astparam);
```

```
624
625     while (pcp_accept(commasym)) {
626         EXPECT(idsym);
627         ADDTOKEN(tail, lasttoken);
628     }
629
630     EXPECT(colonsym);
631
632     val = token;
633     if (!sym_to_type(val->sym, &type)) return 0;
634     NEXTTOKEN();
635
636     /* add all the id's to the symbol table */
637     cur = &tokens;
638     while (cur != NULL) {
639         if (!pcaddparam(cur->token->val.id, type, cur->
    token->lineno)) return 0;
640
641         astcur = AST_initialize(idasm);
642         astcur->name = strdup(cur->token->val.id);
643         AST_addchild(astparam, astcur);
644
645         tail = cur;
646         cur = cur->next;
647         if (tail != &tokens) free(tail);
648     }
649
650     return 1;
651 }
652
653 int
654 pcp_function_declaration(AST *ast) {
655     symtype type;
656     AST *astfunc;
657     symentry *entry;
658
659     printf("## ENTERING pcp_function_declaration ##\n");
660
661     /* enter our new scope for the function */
662     EXPECT(idsym);
663     if (!(entry = pcenterscope(lasttoken->val.id,
    functiontype, lasttoken->lineno))) return 0;
664
665     astfunc = AST_initialize(functionasm);
666     astfunc->name = strdup(lasttoken->val.id);
667     AST_addchild(ast, astfunc);
668
669     EXPECT(lparensym);
670     if (!pcp_formal_parameters(astfunc)) return 0;
671
```

```
672     EXPECT(rparensym);
673     EXPECT(colonsym);
674 
675     if (!sym_to_type(token->sym, &type)) return 0;
676 
677     /* update our return type */
678     entry->returntype = type;
679 
680     NEXTTOKEN();
681     EXPECT(semicolonsym);
682 
683     if (!pcp_block(astfunc)) return 0;
684 
685     /* leave the function scope */
686     return pcleavescope();
687 }
688 
689 int
690 pcp_procedure_declaration(AST *ast) {
691     AST *astproc;
692 
693     printf("## ENTERING pcp_procedure_declaration ##\n");
694 
695     /* create our new scope for the new variables */
696     EXPECT(idsym);
697     if (!pcenterscope(lasttoken->val.id, proceduretype,
    lasttoken->lineno)) return 0;
698 
699     astproc = AST_initialize(procedureasm);
700     astproc->name = strdup(lasttoken->val.id);
701     AST_addchild(ast, astproc);
702 
703     EXPECT(lparensym);
704     if (!pcp_formal_parameters(astproc)) return 0;
705 
706     EXPECT(rparensym);
707     EXPECT(semicolonsym);
708 
709     if (!pcp_block(astproc)) return 0;
710 
711     /* leave the procedure scope */
712     return pcleavescope();
713 }
714 
715 int
716 pcp_procedure_and_function_definition_part(AST *ast) {
717     printf("## ENTERING
    pcp_procedure_and_function_definition_part ##\n");
718 
719     if (pcp_accept(proceduresym)) {
```

```
720         pcp_procedure_declaration(ast);
721     } else if (pcp_accept(functionsym)) {
722         pcp_function_declaration(ast);
723     } else return 1;
724 
725     /*NEXTTOKEN();*/
726     EXPECT(semicolonsym);
727 
728     return pcp_procedure_and_function_definition_part(ast
    );
729 }
730 
731 int
732 pcp_variable_definition(AST *ast) {
733     pctokenlist tokens = {NULL, NULL};
734     pctokenlist *tail = NULL;
735     pctokenlist *cur;
736     pctoken *val;
737     symtype type;
738     AST *astcur;
739 
740     printf("## ENTERING pcp_variable_definition ##\n");
741 
742     EXPECT(idsym);
743     tokens.token = lasttoken;
744     tail = &tokens;
745 
746     while (pcp_accept(commasym)) {
747         /* update the linked list */
748         EXPECT(idsym);
749         ADDTOKEN(tail, lasttoken);
750     }
751 
752     EXPECT(colonsym);
753 
754     val = token;
755     if (!sym_to_type(val->sym, &type)) return 0;
756     NEXTTOKEN();
757 
758     /* add all the id's to the symbol table */
759     cur = &tokens;
760     while (cur != NULL) {
761         if (!pcaddsym(cur->token->val.id, type, (symval) 0
    , 0, cur->token->lineno)) return 0;
762 
763         /* add to the parse tree */
764         astcur = AST_initialize(idasm);
765         astcur->name = strdup(cur->token->val.id);
766         AST_addchild(ast, astcur);
767 
```

```
768         tail = cur;
769         cur = cur->next;
770         if (tail != &tokens) free(tail);
771     }
772 
773     EXPECT(semicolonsym);
774     if (token->sym == idsym) return
    pcp_variable_definition(ast);
775 
776     return 1;
777 }
778 
779 int
780 pcp_variable_definition_part(AST *ast) {
781     AST *astvar;
782     printf("## ENTERING pcp_variable_definition_part ##\n"
    );
783 
784     if (!pcp_accept(varsym)) return 1;
785 
786     astvar = AST_initialize(varasm);
787     AST_addchild(ast, astvar);
788 
789     return pcp_variable_definition(astvar);
790 }
791 
792 int
793 pcp_constant_definition(AST *ast) {
794     pctokenlist tokens = {NULL, NULL};
795     pctokenlist *tail = NULL;
796     pctokenlist *cur;
797     pctoken *val;
798     symtype type;
799     AST *astcur;
800 
801     printf("## ENTERING pcp_constant_definition ##\n");
802 
803     EXPECT(idsym);
804     tokens.token = lasttoken;
805     tail = &tokens;
806 
807     while (pcp_accept(commasym)) {
808         /* update the linked list */
809         EXPECT(idsym);
810         ADDTOKEN(tail, lasttoken);
811     }
812 
813     EXPECT(eqsym);
814 
815     if (!(val = pcp_const_no_id(&type))) return 0;
```

```
816     NEXTTOKEN();
817
818     /* add all the id's to the symbol table */
819     cur = &tokens;
820     while (cur != NULL) {
821         /* add to the symbol table */
822         if (!pcaddsym(cur->token->val.id, type, val->val,
    1, cur->token->lineno)) return 0;
823
824         /* add to the parse tree */
825         astcur = AST_initialize(idasm);
826         astcur->name = strdup(cur->token->val.id);
827         AST_addchild(ast, astcur);
828
829         /* free and go to the next */
830         tail = cur;
831         cur = cur->next;
832         if (tail != &tokens) free(tail);
833     }
834
835     EXPECT(semicolonsym);
836     if (token->sym == idsym) return
    pcp_constant_definition(ast);
837
838     return 1;
839 }
840
841 int
842 pcp_constant_definition_part(AST *ast) {
843     AST *astconst;
844
845     printf("## ENTERING pcp_constant_definition_part ##\n"
    );
846
847     if (!pcp_accept(constsym)) return 1;
848
849     astconst = AST_initialize(constasm);
850     AST_addchild(ast, astconst);
851
852     return pcp_constant_definition(astconst);
853 }
854
855 int
856 pcp_block(AST *ast) {
857     if (!pcp_constant_definition_part(ast)) return 0;
858     /*if (!pcp_type_definition_part()) return 0;*/
859     if (!pcp_variable_definition_part(ast)) return 0;
860     if (!pcp_procedure_and_function_definition_part(ast))
    return 0;
861     if (!pcp_statement_part(ast)) return 0;
```

```
862
863     return 1;
864 }
865
866 int
867 pcp_program() {
868     EXPECT(programsym);
869
870     /* add to our symbol table */
871     EXPECT(idsym);
872     if (!pcaddsym(lasttoken->val.id, programtype, (symval
    ) 0, 0, lasttoken->lineno)) return 0;
873
874     /* add to our tree */
875     astroot = AST_initialize(programasm);
876     astroot->name = strdup(lasttoken->val.id);
877
878     EXPECT(semicolonsym);
879
880     if (!pcp_block(astroot)) return 0;
881
882     /*NEXTTOKEN();*/
883     if (token->sym != dotsym) {
884         return pcerror("[%u] pcp_expect: Unexpected symbol
    : %s vs %s\n", token->lineno, token->val, pcsymstr[dotsym
    ]);
885     }
886
887     if (pcgettoken(fp) != NULL) {
888         pcerror("Expected end-of-file, but there is still
    content.");
889         return 0;
890     }
891
892     return 1;
893 }
894
895 int
896 pcp_start() {
897     return pcp_program();
898 }
899
900 int
901 pcparse(FILE *ifp) {
902     fp = ifp;
903     lasttoken = token = nexttoken = pcgettoken(fp);
904     NEXTTOKEN();
905
906     return pcp_start();
907 }
```

```
1 #ifndef PARSER_H
2 #define PARSER_H
3
4 #include "scanner.h"
5
6 /* Parsers our input file */
7 int pcparse(FILE *fp);
8
9 #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
 * symbol table generator.
 */
#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 */

symentry *rootentry = NULL;    /* our root entry */

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, int *offset) {
    symentry *entry;
    symtab *tab;
```

```
47
48      if (!current) return NULL;
49
50      /* go through the current scope looking for the lexeme
    */
51      entry = current->entries;
52      while (entry) {
53          if (strcmp(entry->name, name) == 0) {
54              if (offset) *offset = entry->offset;
55              return entry;
56          }
57
58          entry = entry->next;
59      }
60
61      /* check the see if we're referencing our own name */
62      entry = current->block;
63      if (strcmp(entry->name, name) == 0) {
64          if (offset) *offset = entry->offset;
65          return entry;
66      }
67
68      /* go up to the parent, if needed */
69      if (current_scope_only) return NULL;
70
71      if (offset) *offset = current->block->size;
72      tab = current->parent;
73      while (tab) {
74          entry = tab->entries;
75          while (entry) {
76              if (strcmp(entry->name, name) == 0) {
77                  if (offset) *offset += entry->offset;
78                  return entry;
79              }
80
81              entry = entry->next;
82          }
83
84          /* keep going up */
85          if (offset) *offset += current->block->size;
86          tab = tab->parent;
87      }
88
89      /* never found */
90      return NULL;
91  }
92
93  int pcintializesymtab() {
94      symval val;
95
```

```
 96      /* create a main entry */
 97      if (!(rootentry = malloc(sizeof(*rootentry)))) return
    0;
 98      rootentry->name = strdup("main");
 99      rootentry->type = programtype;
100      rootentry->val = (symval) 0;
101      rootentry->bconst = 1;
102      rootentry->lineno = 0;
103      rootentry->tab = NULL;
104      rootentry->params = NULL;
105      rootentry->returntype = notype;
106      rootentry->size = 0;
107      rootentry->offset = 0;
108
109      /* create or root table and set it current */
110      if (!(root = malloc(sizeof(*root)))) return 0;
111
112      current = root;
113      current->parent = NULL;
114      current->entries = NULL;
115      current->block = rootentry;
116
117      val.ival = 0;
118
119      /* this is expanded from scanner.c -> pcgetkeyword
    () */
120      /* TODO: make a pckeywords struct array */
121
122      pcaddsym("div", keywordtype, (symval) "div", 1, 0);
123      pcaddsym("mod", keywordtype, (symval) "mod", 1, 0);
124
125      pcaddsym("program", keywordtype, (symval) "program", 1
    , 0);
126      pcaddsym("procedure", keywordtype, (symval) "procedure
    ", 1, 0);
127      pcaddsym("function", keywordtype, (symval) "function"
    , 1, 0);
128      pcaddsym("begin", keywordtype, (symval) "begin", 1, 0
    );
129      pcaddsym("end", keywordtype, (symval) "end", 1, 0);
130
131      pcaddsym("and", keywordtype, (symval) "and", 1, 0);
132      pcaddsym("or", keywordtype, (symval) "or", 1, 0);
133      pcaddsym("not", keywordtype, (symval) "not", 1, 0);
134
135      pcaddsym("if", keywordtype, (symval) "if", 1, 0);
136      pcaddsym("else", keywordtype, (symval) "else", 1, 0);
137      pcaddsym("then", keywordtype, (symval) "then", 1, 0);
138      pcaddsym("do", keywordtype, (symval) "do", 1, 0);
139      pcaddsym("while", keywordtype, (symval) "while", 1, 0
```

```
139 );
140
141     pcaddsym("array", keywordtype, (symval) "array", 1, 0
    );
142     pcaddsym("of", keywordtype, (symval) "of", 1, 0);
143     pcaddsym("char", keywordtype, (symval) "char", 1, 0);
144     pcaddsym("string", keywordtype, (symval) "string", 1,
    0);
145     pcaddsym("integer", keywordtype, (symval) "integer", 1
    , 0);
146     pcaddsym("real", keywordtype, (symval) "real", 1, 0);
147     pcaddsym("var", keywordtype, (symval) "var", 1, 0);
148     pcaddsym("const", keywordtype, (symval) "const", 1, 0
    );
149
150     pcaddsym("chr", keywordtype, (symval) "chr", 1, 0);
151     pcaddsym("ord", keywordtype, (symval) "ord", 1, 0);
152     pcaddsym("read", keywordtype, (symval) "read", 1, 0);
153     pcaddsym("readln", keywordtype, (symval) "readln", 1,
    0);
154     pcaddsym("write", keywordtype, (symval) "write", 1, 0
    );
155     pcaddsym("writeln", keywordtype, (symval) "writeln", 1
    , 0);
156
157     return 1;
158 }
159
160 void pcprintsymtabnode(symtab *node, unsigned depth) {
161     symentry *entry;
162     char tabs[21], *c;
163     int i;
164
165     if (!node) return;
166
167     /* setup our tabs */
168     c = tabs;
169     for (i = 0; (i < depth && i < 20); ++i) {
170         *c++ = '\t';
171     }
172     *c = '\0';
173
174     /* print self first */
175     entry = node->entries;
176     while (entry) {
177         switch (entry->type) {
178             case integertype:
179                 printf("%s%s (%s @ %d // %d) : %d : %d :\n
    ", tabs, entry->name, symtypestr[entry->type], entry->
    offset,
```

```
180                         entry->size, entry->lineno, entry->
    val.ival);
181                 break;
182             case realtype:
183                 printf("%s%s (%s @ %d // %d) : %d : %f\n"
    , tabs, entry->name, symtypestr[entry->type], entry->
    offset,
184                         entry->size, entry->lineno, entry->
    val.rval);
185                 break;
186             case chartype:
187                 printf("%s%s (%s @ %d // %d) : %d : %c\n"
    , tabs, entry->name, symtypestr[entry->type], entry->
    offset,
188                         entry->size, entry->lineno, entry->
    val.cval);
189                 break;
190             case stringtype:
191                 printf("%s%s (%s @ %d // %d) : %d : %s\n"
    , tabs, entry->name, symtypestr[entry->type], entry->
    offset,
192                         entry->size, entry->lineno, entry->
    val.str);
193                 break;
194             default:
195                 printf("%s%s (%s @ %d // %d) : %d : NULL\n
    ", tabs, entry->name, symtypestr[entry->type], entry->
    offset,
196                         entry->size, entry->lineno);
197         }
198 
199         /* print the symbol table for the child, if it
    exists */
200         if (entry->tab) pcprintsymtabnode(entry->tab,
    depth + 1);
201 
202         /* go to the next entry */
203         entry = entry->next;
204     }
205 }
206 
207 void pcprintsymtab() {
208     printf("\n===== SYMBOL TABLE @ %d =====\n", root->
    block->size);
209     pcprintsymtabnode(root, 0);
210 }
211 
212 void pccleanupsymtabnode(symtab **tab) {
213     symentry *cur;
214 
```

```
215     cur = (*tab)->entries;
216     while (cur) {
217         /* cleanup children fist */
218         if (cur->tab) {
219             pccleanupsymtabnode(&(cur->tab));
220             cur->tab = NULL;
221         }
222 
223         /* cleanup name */
224         free((void *) (cur->name));
225         cur->name = NULL;
226 
227         /* next */
228         cur = cur->next;
229     }
230 
231     /* destroy our symtab */
232     free(*tab);
233     (*tab) = NULL;
234 }
235 
236 void pccleanupsymtab() {
237     pccleanupsymtabnode(&root);
238 }
239 
240 symentry *pcaddsym(const char *name, symtype type, symval
    val, int bconst, unsigned lineno) {
241     symentry *entry;
242 
243     /* make sure we have a root */
244     if (!current) {
245         pcerror("{%d} ERR: No symbol table defined.\n");
246         return NULL;
247     }
248 
249     /* make sure it doesn't yet exist in this scope */
250     if (pclookupsym_internal(name, 1, NULL)) {
251         pcerror("{%d} ERR: %s already exists in symbol
    table.\n", lineno, name);
252         return NULL;
253     }
254 
255     /* populate our entry */
256     if (!(entry = malloc(sizeof(*entry)))) return 0;
257     entry->name = strdup(name);
258     entry->type = type;
259     entry->val = val;
260     entry->bconst = bconst;
261     entry->lineno = lineno;
262     entry->tab = NULL;
```

```
263     entry->params = NULL;
264     entry->returntype = notype;
265 
266     /* stack memory information */
267     switch (type) {
268         case stringtype:
269             entry->size = strlen(val.str) + 1;
270 
271             /* make sure we're padded to 4 */
272             if (entry->size % 4) {
273                 entry->size = entry->size + (4 - (entry->
    size % 4));
274             }
275 
276             /* update our offset */
277             entry->offset = current->block->size;
278             break;
279 
280         case blocktype:
281         case functiontype:
282         case proceduretype:
283         case programtype:
284         case keywordtype:
285             /* block types have no size or offset */
286             entry->size = 0;
287             entry->offset = 0;
288             break;
289 
290         default:
291             /* update our offset */
292             entry->size = 4;
293             entry->offset = current->block->size;
294     }
295 
296     /* update our current stack size */
297     if (type != keywordtype) current->block->size += entry
    ->size;
298 
299     /* add to the head of the entries */
300     entry->next = current->entries;
301     current->entries = entry;
302     return entry;
303 }
304 
305 symentry *pcaddparam(const char *name, symtype type,
    unsigned lineno) {
306     symentry *entry;
307     symentry *func;
308     symparam *param;
309     symparam *cur;
```

```
310
311     /* make sure we're in a function/procedure */
312     if (!(func = current->block) || (func->type !=
    proceduretype && func->type != functiontype)) {
313         pcerror("{%d} ERR: Unable to determine function/
    procedure.\n", lineno);
314         return NULL;
315     }
316
317     /* add this to the symbol table */
318     if (!(entry = pcaddsym(name, type, (symval) 0, 0,
    lineno))) return NULL;
319
320     /* update the params with the new param */
321     param = malloc(sizeof(*param));
322     param->entry = entry;
323     param->next = NULL;
324
325     /* add to the root if there isn't one here yet */
326     if (!(func->params)) {
327         func->params = param;
328         return entry;
329     }
330
331     /* add to the tail */
332     cur = func->params;
333     while (cur->next) cur = cur->next;
334     cur->next = param;
335
336     return entry;
337 }
338
339 symentry *pclookupsym_entry(const char *name, int *offset
    ) {
340     *offset = 0;
341     return pclookupsym_internal(name, 0, offset);
342 }
343
344 symentry *pclookupsym(const char *name) {
345     return pclookupsym_internal(name, 0, NULL);
346 }
347
348 int pcenterscope_nocreate(symentry *entry) {
349     if (!entry || !entry->tab) return pcerror("Unable to
    enter scope!\n");
350
351     current = entry->tab;
352     return 1;
353 }
354
```

```
355 symentry *pcenterscope(const char *name, symtype type,
    unsigned lineno) {
356     symentry *entry;
357 
358     if (!(entry = pcaddsym(name, type, (symval) 0, 0,
    lineno))) return NULL;
359 
360     /* create our table and make it the current, while
    updating it's parent */
361     if (!(entry->tab = malloc(sizeof(*(entry->tab)))))
    return NULL;
362     entry->tab->parent = current;
363     entry->tab->entries = NULL;
364     entry->tab->block = entry;
365 
366     if (!pcenterscope_nocreate(entry)) return 0;
367     return entry;
368 }
369 
370 int pcleavescope() {
371     /* can't leave if we're top dog */
372     if (current == root) return 0;
373 
374     /* go up to our parent */
375     current = current->parent;
376     return 1;
377 }
378 
379 int pcrootsize() {
380     if (root) return root->block->size;
381     return 0;
382 }
```

```
 1 /*
 2  * @author Derek Trom
 3  * @author Elena Corpus
 4  * symtab.h is responsible for storing our tokens so that
   they can be accessed later.
 5  * Things to focus on are names, values, and scope.
   Initially, the symtable is only
 6  * filled with keywords.
 7 */
 8
 9
10 #ifndef SYMTAB_H
11 #define SYMTAB_H
12
13 #include "tokens.h"
14
15 /*
16 Symtype holds information about the type of an entry in the
    symbol table.
17 */
18 typedef enum symtype {
19     /* Keywords */
20     keywordtype = 0,
21
22     /* Variable types */
23     chartype,
24     stringtype,
25     integertype,
26     realtype,
27
28     /* Block types */
29     programtype,
30     proceduretype,
31     functiontype,
32     blocktype,
33
34     /* total count of types */
35     notype,
36     numsymtypes
37 } symtype;
38
39 /* Parameters for functions and procedures. */
40 struct symentry;
41 typedef struct symparam {
42     struct symentry *entry;
43     struct symparam *next;
44 } symparam;
45
46 /*
47 Array of string representation for each symtype.
```

```
48 KEEP UP TO DATE WITH symtype enum.
49 */
50 extern const char *symtypestr[numsymtypes];
51
52 /*
53 Symentry holds links for our table.
54 */
55 struct symtab;
56 typedef struct symentry {
57     const char *name;    /* name of the lexeme */
58     symtype type;    /* type of the lexeme */
59     symval val;    /* value of the lexeme */
60     unsigned lineno;    /* line the lexeme was declared on
    */
61     int bconst;    /* whether or not it's constant */
62     unsigned size;    /* the size of the entry */
63     int offset;    /* stack offset */
64
65     struct symtab *tab;        /* symbol table for this
   entry (procedures and functions) */
66     symtype returntype; /* return type for functions */
67     struct symparam *params;    /* paramaters */
68
69     /* link to the next entry */
70     struct symentry *next;
71 } symentry;
72
73 struct symtab {
74     struct symtab *parent;    /* parent symtab */
75     symentry *block;        /* the block entry that starts
   this */
76     symentry *entries;    /* linked list of entries */
77 };
78
79 typedef struct symtab symtab;
80
81 /*
82 Initializes the symbol table with keywords.
83
84 @return 1 on success; 0 otherwise
85 */
86 int pcintializesymtab();
87
88 /*
89 Prints the symbol table.
90 */
91 void pcprintsymtab();
92
93 /*
94 Cleans up the symbol table.
```

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

```
141 @param type the type of the lexeme
142 @param lineno the line the lexeme is declared on
143 @return 1 on success; 0 otherwise
144 */
145 symentry *pcenterscope(const char *name, symtype type,
    unsigned lineno);
146
147 /*
148 Enters the scope of the given element without creating an
    entry.
149 */
150 int pcenterscope_nocreate(symentry *entry);
151
152 /*
153 Leaves the current scope, returning to the parent scope.
154
155 @return 1 on success; 0 otherwise
156 */
157 int pcleavescope();
158
159 int pcrootsize();
160
161
162 #endif /* SYMTAB_H */
```

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

```c
/*
 * @author Derek Trom
 * @author Elena Corpus
 * tokens.h
 */

#ifndef TOKENS_H
#define TOKENS_H

/*
All of our possible sym value types.
*/
typedef union symval {
    int ival;
    double rval;
    char cval;
    char *id;
    char *str;
} symval;

/*
All of our possible tokens.
KEEP UP TO DATE WITH pcsymstr.
*/
typedef enum {
    /* End-of-Tokens */
    eofsym = 0,

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

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

    /* Boolean operators */
    andsym,
    orsym,
    notsym,
    ltsym,
    ltesym,
    neqsym,
```

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

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

```
 1 [program name:fibonacci]
 2     [const]
 3         [id name:question]
 4         [id name:msginline]
 5         [id name:msgfunction]
 6         [id name:msgrecursive]
 7     [var]
 8         [id name:n]
 9         [id name:cur]
10         [id name:temp]
11         [id name:f1]
12         [id name:f2]
13     [procedure name:recursivefibonacci]
14         [param]
15             [id name:n]
16             [id name:f1]
17             [id name:f2]
18         [statement]
19             [if]
20                 [expr]
21                     [simexpr]
22                         [term]
23                             [factor]
24                                 [id name:n]
25                     [rel]
26                     [simexpr]
27                         [term]
28                             [factor]
29                                 [val]
30                 [statement]
31                     [write]
32                         [expr]
33                             [simexpr]
34                                 [term]
35                                     [factor]
36                                         [val val:Y]
37                     [if]
38                         [expr]
39                             [simexpr]
40                                 [term]
41                                     [factor]
42                                         [id name:f1]
43                             [rel]
44                             [simexpr]
45                                 [term]
46                                     [factor]
47                                         [val]
48                         [statement]
49                             [write]
50                                 [expr]
```

```
51                                          [simexpr]
52                                              [term]
53                                                  [factor]
54                                                      [val val:Y
   ]
55                                  [proccall name:
   recursivefibonacci]
56                                      [expr]
57                                          [simexpr]
58                                              [term]
59                                                  [factor]
60                                                      [id name:n
   ]
61                                              [add]
62                                              [term]
63                                                  [factor]
64                                                      [val val:Y
   ]
65                                      [expr]
66                                          [simexpr]
67                                              [term]
68                                                  [factor]
69                                                      [val val:Y
   ]
70                                      [expr]
71                                          [simexpr]
72                                              [term]
73                                                  [factor]
74                                                      [val]
75                              [statement]
76                                  [if]
77                                      [expr]
78                                          [simexpr]
79                                              [term]
80                                                  [factor]
81                                                      [id name:
   f2]
82                                          [rel]
83                                          [simexpr]
84                                              [term]
85                                                  [factor]
86                                                      [val]
87                                      [statement]
88                                          [write]
89                                              [expr]
90                                                  [simexpr]
91                                                      [term]
92                                                          [
   factor]
93                                                                  [
```

```
 93 val val:Y]
 94                                        [proccall name:
    recursivefibonacci]
 95                                            [expr]
 96                                                [simexpr]
 97                                                    [term]
 98                                                        [
    factor]
 99                                                            [
    id name:n]
100                                                    [add]
101                                                    [term]
102                                                        [
    factor]
103                                                            [
    val val:Y]
104                                            [expr]
105                                                [simexpr]
106                                                    [term]
107                                                        [
    factor]
108                                                            [
    val val:Y]
109                                            [expr]
110                                                [simexpr]
111                                                    [term]
112                                                        [
    factor]
113                                                            [
    val val:Y]
114                                    [statement]
115                                        [write]
116                                            [expr]
117                                                [simexpr]
118                                                    [term]
119                                                        [
    factor]
120                                                            [
    id name:f1]
121                                                    [add]
122                                                    [term]
123                                                        [
    factor]
124                                                            [
    id name:f2]
125                                        [proccall name:
    recursivefibonacci]
126                                            [expr]
127                                                [simexpr]
128                                                    [term]
```

```
129                                                                 [
    factor]
130                                                                     [
    id name:n]
131                                                             [add]
132                                                             [term]
133                                                                 [
    factor]
134                                                                     [
    val val:Y]
135                                                 [expr]
136                                                     [simexpr]
137                                                         [term]
138                                                             [
    factor]
139                                                                     [
    id name:f2]
140                                                 [expr]
141                                                     [simexpr]
142                                                         [term]
143                                                             [
    factor]
144                                                                     [
    id name:f1]
145                                                             [add]
146                                                             [term]
147                                                                 [
    factor]
148                                                                     [
    id name:f2]
149     [function name:nextfibonacci]
150         [param]
151             [id name:f1]
152             [id name:f2]
153         [statement]
154             [if]
155                 [expr]
156                     [simexpr]
157                         [term]
158                             [factor]
159                                 [id name:f1]
160                     [rel]
161                     [simexpr]
162                         [term]
163                             [factor]
164                                 [val]
165                 [statement]
166                     [assign name:nextfibonacci]
167                         [id name:nextfibonacci]
168                         [expr]
```

```
                                    [simexpr]
                                        [term]
                                            [factor]
                                                [val val:Y]
                    [statement]
                        [if]
                            [expr]
                                [simexpr]
                                    [term]
                                        [factor]
                                            [id name:f2]
                                [rel]
                                [simexpr]
                                    [term]
                                        [factor]
                                            [val]
                            [statement]
                                [assign name:nextfibonacci]
                                    [id name:nextfibonacci]
                                    [expr]
                                        [simexpr]
                                            [term]
                                                [factor]
                                                    [val val:Y
    ]
                            [statement]
                                [assign name:nextfibonacci]
                                    [id name:nextfibonacci]
                                    [expr]
                                        [simexpr]
                                            [term]
                                                [factor]
                                                    [id name:
    f1]
                                            [add]
                                            [term]
                                                [factor]
                                                    [id name:
    f2]
        [statement]
            [write]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [id name:question]
            [read]
                [id name:n]
            [writeln]
                [expr]
```

```
                    [simexpr]
                        [term]
                            [factor]
                                [val val:Y]
            [writeln]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [id name:msginline]
            [assign name:cur]
                [id name:cur]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [val]
            [while]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [id name:cur]
                    [rel]
                    [simexpr]
                        [term]
                            [factor]
                                [id name:n]
                [statement]
                    [if]
                        [expr]
                            [simexpr]
                                [term]
                                    [factor]
                                        [id name:cur]
                            [rel]
                            [simexpr]
                                [term]
                                    [factor]
                                        [val val:Y]
                        [statement]
                            [write]
                                [expr]
                                    [simexpr]
                                        [term]
                                            [factor]
                                                [val val:Y]
                            [write]
                                [expr]
                                    [simexpr]
```

```
                                        [term]
                                            [factor]
                                                [val val:Y]
                                [assign name:f1]
                                    [id name:f1]
                                    [expr]
                                        [simexpr]
                                            [term]
                                                [factor]
                                                    [val val:Y]
                                [assign name:f2]
                                    [id name:f2]
                                    [expr]
                                        [simexpr]
                                            [term]
                                                [factor]
                                                    [val val:Y]
                            [statement]
                                [assign name:temp]
                                    [id name:temp]
                                    [expr]
                                        [simexpr]
                                            [term]
                                                [factor]
                                                    [id name:f2]
                                [assign name:f2]
                                    [id name:f2]
                                    [expr]
                                        [simexpr]
                                            [term]
                                                [factor]
                                                    [id name:f1]
                                            [add]
                                            [term]
                                                [factor]
                                                    [id name:f2]
                                [assign name:f1]
                                    [id name:f1]
                                    [expr]
                                        [simexpr]
                                            [term]
                                                [factor]
                                                    [id name:temp]
                                [write]
                                    [expr]
                                        [simexpr]
                                            [term]
                                                [factor]
                                                    [val val:Y]
                                [write]
```

```
                                [expr]
                                    [simexpr]
                                        [term]
                                            [factor]
                                                [id name:f2]
                    [assign name:cur]
                        [id name:cur]
                        [expr]
                            [simexpr]
                                [term]
                                    [factor]
                                        [id name:cur]
                                [add]
                                [term]
                                    [factor]
                                        [val val:Y]
            [writeln]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [val val:Y]
            [writeln]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [val val:Y]
            [writeln]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [id name:msgfunction]
            [assign name:cur]
                [id name:cur]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [val]
            [assign name:f1]
                [id name:f1]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [val]
            [assign name:f2]
                [id name:f2]
```

```
366             [expr]
367                 [simexpr]
368                     [term]
369                         [factor]
370                             [val]
371         [while]
372             [expr]
373                 [simexpr]
374                     [term]
375                         [factor]
376                             [id name:cur]
377                 [rel]
378                 [simexpr]
379                     [term]
380                         [factor]
381                             [id name:n]
382             [statement]
383                 [assign name:temp]
384                     [id name:temp]
385                     [expr]
386                         [simexpr]
387                             [term]
388                                 [factor]
389                                     [funccall name:
    nextfibonacci]
390                                         [expr]
391                                             [simexpr]
392                                                 [term]
393                                                     [
    factor]
394                                                         [
    id name:f1]
395                                         [expr]
396                                             [simexpr]
397                                                 [term]
398                                                     [
    factor]
399                                                         [
    id name:f2]
400                 [write]
401                     [expr]
402                         [simexpr]
403                             [term]
404                                 [factor]
405                                     [val val:Y]
406                 [write]
407                     [expr]
408                         [simexpr]
409                             [term]
410                                 [factor]
```

```
411                                     [id name:temp]
412                 [assign name:f1]
413                     [id name:f1]
414                     [expr]
415                         [simexpr]
416                             [term]
417                                 [factor]
418                                     [id name:f2]
419                 [assign name:f2]
420                     [id name:f2]
421                     [expr]
422                         [simexpr]
423                             [term]
424                                 [factor]
425                                     [id name:temp]
426                 [assign name:cur]
427                     [id name:cur]
428                     [expr]
429                         [simexpr]
430                             [term]
431                                 [factor]
432                                     [id name:cur]
433                             [add]
434                             [term]
435                                 [factor]
436                                     [val val:Y]
437         [writeln]
438             [expr]
439                 [simexpr]
440                     [term]
441                         [factor]
442                             [val val:Y]
443         [writeln]
444             [expr]
445                 [simexpr]
446                     [term]
447                         [factor]
448                             [val val:Y]
449         [writeln]
450             [expr]
451                 [simexpr]
452                     [term]
453                         [factor]
454                             [id name:msgrecursive]
455         [proccall name:recursivefibonacci]
456             [expr]
457                 [simexpr]
458                     [term]
459                         [factor]
460                             [id name:n]
```

```
461                 [expr]
462                     [simexpr]
463                         [term]
464                             [factor]
465                                 [val]
466                 [expr]
467                     [simexpr]
468                         [term]
469                             [factor]
470                                 [val]
471
```

```c
/*
 * @author Derek Trom
 * @author Elena Corpus
 * scanner/lexer that looks for syntax errors.
 */
#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;
    }
}

```

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

```
 94     );
 95 }
 96
 97 /*
 98 Determines if the character is a random character (i.e.
    not a end of token,
 99 not alpha, and not a number).
100 */
101 int
102 pcisrandom(char c) {
103     return (!pcisendoftoken(c) && !isalpha(c) && !isdigit(
    c));
104 }
105
106 void
107 pcresetline() {
108     /* print the line */
109     *lineptr = '\0';
110     printf("[%d] %s\n", pclineno, line);
111
112     /* reset the lineptr and size */
113     lineptr = line;
114     linesize = 0;
115 }
116
117 /*
118 Skips whitespace.
119 */
120 void
121 pcskipwhitespace(char *cur, char *next, FILE *fp) {
122     int dontprint = 0;
123     while (isspace(*cur)) {
124         /* see if we have a new line */
125         if (*cur == '\n') {
126             /* don't print multiple blank lines */
127             if (!dontprint) {
128                 pcresetline();
129                 dontprint = 1;
130             } else {
131                 lineptr = line;
132                 linesize = 0;
133                 *lineptr = '\0';
134             }
135
136             /* increment our line counter */
137             ++pclineno;
138         }
139
140         pcgetnextc(cur, next, fp);
141     }
```

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

```
190
191     /* found a keyword; sym has been updated */
192     return 1;
193 }
194
195 pctoken *
196 pcgettoken(FILE *fp) {
197     char cur, next,        /* current and next characters
    in the FILE */
198     *b, buf[255];    /* buffer filled while grabbing
    characters */
199     symval val;            /* value of the token */
200     pcsym sym;            /* sym of the token */
201
202     /* skip whitespace */
203     next = pcgetc(fp);
204     pcgetnextc(&cur, &next, fp);
205     pcskipwhitespace(&cur, &next, fp);
206
207     /* end-of-file? */
208     if (cur == EOF) {
209         pcresetline();
210         return NULL;
211     }
212
213     /* initialize our variables */
214     b = buf;
215     *b = '\0';
216     sym = eofsym;
217     val.ival = 0;
218
219     /* skip over single-line comments */
220     if (cur == '/') {
221         if (next == '/') {
222             /* consume up to the end of line */
223             while (next != '\n' && next != EOF) {
224                 pcgetnextc(&cur, &next, fp);
225             }
226
227             /* put the \n token back and get the next
    token */
228             PCGETTOKEN_RECURSE(next, fp);
229         }
230     }
231
232     /* skip over multi-line comments */
233     if (cur == '(' || cur == '{') {
234         char end1, end2;
235
236         /* determine our ending 2-char sequence */
```

```
237         if (cur == '(' && next == '*') {
238             end1 = '*';
239             end2 = ')';
240         } else if (cur == '{') {
241             end1 = '}';
242             end2 = 0;
243         } else {
244             end1 = 0;
245             end2 = 0;
246         }
247 
248         /* only skip if we have an ending sequence */
249         if (end1) {
250             /* store our starting lineno, since it will
    likely change */
251             unsigned startinglineno = pclineno;
252 
253             while (1) {
254                 /* match the first part */
255                 if (cur == end1) {
256                     /* only 1 to match, so leave our next
     */
257                     if (!end2) {
258                         break;
259                     }
260 
261                     /* 2 to match, so grab the next value
     */
262                     if (end2 && next == end2) {
263                         pcgetnextc(&cur, &next, fp);
264                         break;
265                     }
266                 }
267 
268                 /* warn if we hit the end of file without
    terminating */
269                 if (cur == EOF) {
270                     if (end2) {
271                         pcerror("{%d} ERR: Multiline
    comment missing termintors: %c%c", startinglineno, end1,
    end2);
272                     } else {
273                         pcerror("{%d} ERR: Multiline
    comment missing termintor: %c", startinglineno, end1);
274                     }
275 
276                     return NULL;
277                 }
278 
279                 /* add to our linecount on \n */
```

```
280                 if (cur == '\n') {
281                     ++pclineno;
282                 }
283 
284                 /* keep skipping characters */
285                 pcgetnextc(&cur, &next, fp);
286             }
287 
288             /* put the next token back and get the next
    token */
289             PCGETTOKEN_RECURSE(next, fp);
290         }
291     }
292 
293     /* check the terminators */
294     if (pcistermintor(cur)) {
295         switch (cur) {
296             case '(':
297                 sym = lparensym;
298                 break;
299             case ')':
300                 sym = rparensym;
301                 break;
302             case '[':
303                 sym = lbracksym;
304                 break;
305             case ']':
306                 sym = rbracksym;
307                 break;
308             case ';':
309                 sym = semicolonsym;
310                 break;
311             case ',':
312                 sym = commasym;
313                 break;
314             case '.':
315                 if (next == '.') {
316                     sym = dotdotsym;
317                     pcgetnextc(&cur, &next, fp);
318                 } else {
319                     sym = dotsym;
320                 }
321                 break;
322             case ':':
323                 if (next == '=') {
324                     sym = assignsym;
325                     pcgetnextc(&cur, &next, fp);
326                 } else {
327                     sym = colonsym;
328                 }
```

```
329                 break;
330             case '=':
331                 sym = eqsym;
332                 break;
333             case '<':
334                 if (next == '=') {
335                     sym = ltesym;
336                     pcgetnextc(&cur, &next, fp);
337                 } else if (next == '>') {
338                     sym = neqsym;
339                     pcgetnextc(&cur, &next, fp);
340                 } else {
341                     sym = ltsym;
342                 }
343                 break;
344             case '>':
345                 if (next == '=') {
346                     sym = gtesym;
347                     pcgetnextc(&cur, &next, fp);
348                 } else {
349                     sym = gtsym;
350                 }
351                 break;
352             case '+':
353                 sym = addsym;
354                 break;
355             case '-':
356                 sym = minussym;
357                 break;
358             case '*':
359                 sym = multsym;
360                 break;
361             case '/':
362                 sym = divsym;
363                 break;
364         }
365     }
366         /* now check for a number */
367     else if (isdigit(cur)) {
368         *b++ = cur;
369 
370         /* keep adding digits until the next isn't a digit
    */
371         while (isdigit(next)) {
372             pcappendnext(&b, &cur, &next, fp);
373         }
374 
375         /* see if we have a dot and shift to real digit */
376         if (next == '.') {
377             pcappendnext(&b, &cur, &next, fp);
```

```
378
379             /* keep adding digits until the next isn't a
    digit */
380             while (isdigit(next)) {
381                 pcappendnext(&b, &cur, &next, fp);
382             }
383
384             /* see if we have a e or E and shift to
    scientific */
385             if (next == 'e' || next == 'E') {
386                 pcappendnext(&b, &cur, &next, fp);
387
388                 /* check for +/- */
389                 if (next == '+' || next == '-') {
390                     pcappendnext(&b, &cur, &next, fp);
391                 }
392
393                 /* keep adding digits */
394                 while (isdigit(next)) {
395                     pcappendnext(&b, &cur, &next, fp);
396                 }
397
398                 /* if we don't have a terminal/whitespace
    now, ill formed real number */
399                 if (!pcisendoftoken(next)) {
400                     /* keep consuming until we do hit a
    space or terminator */
401                     while (!pcisendoftoken(next)) {
402                         pcappendnext(&b, &cur, &next, fp);
403                     }
404
405                     /* print the error */
406                     *b = '\0';
407                     pcerror("{%d} ERR: Ill formed real
    number: %s\n", pclineno, buf);
408                     ++pcscanerrors;
409
410                     /* go to the next token */
411                     PCGETTOKEN_RECURSE(next, fp);
412                 }
413             }
414                 /* if we don't have a terminal /
    whitespace / random char, ill formed real number */
415             else if (!pcisendoftoken(next)) {
416                 /* keep consuming until we do hit a space
    or terminator */
417                 while (!pcisendoftoken(next)) {
418                     pcappendnext(&b, &cur, &next, fp);
419                 }
420
```

```
421                 /* print the error */
422                 *b = '\0';
423                 pcerror("{%d} ERR: Ill formed real number
    : %s\n", pclineno, buf);
424                 ++pcscanerrors;
425 
426                 /* go to the next token */
427                 PCGETTOKEN_RECURSE(next, fp);
428             }
429 
430             /* we have a legitimate real number! calculate
     and create our token */
431             *b = '\0';
432             val.rval = atof(buf);
433             sym = realnosym;
434         }
435 
436             /* if we don't have a end of token, ill formed
     integer number */
437         else if (!pcisendoftoken(next)) {
438             /* keep consuming until we do hit a space or
    terminator */
439             while (!pcisendoftoken(next)) {
440                 pcappendnext(&b, &cur, &next, fp);
441             }
442 
443             /* print the error */
444             *b = '\0';
445             pcerror("{%d} ERR: Ill formed integer number
    or id: %s\n", pclineno, buf);
446             ++pcscanerrors;
447 
448             /* go to the next token */
449             PCGETTOKEN_RECURSE(next, fp);
450         }
451 
452             /* we have a good integer! */
453         else {
454             *b = '\0';
455             val.ival = atoi(buf);
456             sym = integernosym;
457         }
458     }
459 
460         /* now check for strings and characters */
461     else if (cur == '\'') {
462         int scount = 0;
463 
464         /* add values to the buffer until we hit a \n or
     ' or EOF */
```

```
465         while (next != '\n' && next != '\'' && next != EOF
    ) {
466             pcgetnextc(&cur, &next, fp);
467             *b++ = cur;
468             ++scount;
469         }
470 
471         /* if we hit a new line or EOF, then we have an
    ill-formed string */
472         if (next == '\n' || next == EOF) {
473             /* print the error */
474             *b = '\0';
475             pcerror("{%d} ERR: No closing ': %s\n",
    pclineno, buf);
476             ++pcscanerrors;
477 
478             /* go to the next token */
479             PCGETTOKEN_RECURSE(next, fp);
480         }
481 
482         /* warn about empty strings */
483         *b = '\0';
484         if (!scount) {
485             pcerror("{%d} WARN: Empty string/character
    found.\n", pclineno);
486             ++pcscanwarnings;
487         }
488 
489         /* prepare our character if 1 value */
490         if (scount == 1) {
491             sym = charvalsym;
492             val.cval = *buf;
493         }
494             /* otherwise, it's a string */
495         else {
496             sym = stringvalsym;
497             val.str = strdup(buf);
498         }
499 
500         /* we consume another from the stream, so the tick
     doesn't go back in */
501         pcgetnextc(&cur, &next, fp);
502     }
503 
504         /* now check for keywords and id's */
505     else if (isalpha(cur)) {
506         *b++ = cur;
507 
508         /* consume letters and numbers */
509         while (isalpha(next) || isdigit(next)) {
```

```
510             pcappendnext(&b, &cur, &next, fp);
511         }
512 
513         /* make sure we have an end of token */
514         if (!pcisendoftoken(next)) {
515             while (!pcisendoftoken(next)) {
516                 pcappendnext(&b, &cur, &next, fp);
517             }
518 
519             /* print the error */
520             *b = '\0';
521             pcerror("{%d} ERR: Ill formed keyword or id: %
    s\n", pclineno, buf);
522             ++pcscanerrors;
523 
524             /* go to the next token */
525             PCGETTOKEN_RECURSE(next, fp);
526         }
527 
528         /* determine what kind of symbol we have */
529         *b = '\0';
530         strtolower(buf);
531         if (!pcgetkeyword(buf, &sym)) {
532             sym = idsym;
533             val.id = strdup(buf);
534         }
535     }
536 
537         /* unknown character */
538     else {
539         pcerror("{%d} ERR: Unknown character: %c\n",
    pclineno, cur);
540         ++pcscanerrors;
541 
542         /* get the next token */
543         PCGETTOKEN_RECURSE(next, fp);
544     }
545 
546     /* unget our next value (so it's our current in next
    call) */
547     if (next != EOF) pcungetc(next, fp);
548 
549     /* generate and return our token */
550     return pcnewtoken(sym, val, pclineno);
551 }
```

```
 1 /*
 2  * @author Derek Trom
 3  * @author Elena Corpus
 4  * scanner header file
 5  */
 6
 7 #ifndef SCANNER_H
 8 #define SCANNER_H
 9
10 #include "tokens.h"
11 #include <stdio.h>
12
13 extern unsigned pclineno;
14 extern int pcscanerrors;
15 extern int pcscanwarnings;
16
17 /*
18 Gets the next token from the stream, or NULL if consumed.
19
20 @param fp the FILE pointer
21 @return a malloc'd next token, or NULL if consumed
22 */
23 pctoken *pcgettoken(FILE *fp);
24
25 #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 %%
```

```
## EDITORS
*~*
*.bak
*.swp
*.tmp
*.log

## WINDOWS
Thumbs.db
Desktop.ini

## MAC
.DS_Store

## C
*.o

## Flex / Bison
*.output
lex.yy.c
*.tab.c
*.tab.h

## Programs
mini-pascal-compiler
yy-mini-pascal-compiler
asftp.txt
*.s

```

```c
/*
 * @author Derek Trom
 * @author Elena Corpus
 * This is the main driver program
 */
#include "compiler.h"

#ifndef YYCOMPILE

# include "tokens.h"
# include "scanner.h"
# include "parser.h"
# include "symtab.h"
# include "io.h"
# include "ast.h"
# include "icg.h"

#endif /* YYCOMPILE */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#ifdef YYCOMPILE
extern FILE *yyin;
extern int yylex(void);
#else
extern int pcscanerrors;
#endif /* YYCOMPILE */

void
usage(const char *progname) {
    printf("Usage: %s filename\n  filename\tPascal file to compile\n", progname);
}

char *
strtolower(char *s) {
    char *c = s;
    for (; *c; ++c) {
        *c = tolower(*c);
    }

    return s;
}

int
main(int argc, char **argv) {
    FILE *fp;
    char *filename;
```

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

```
 99      /* save our AST tree */
100      FILE *astfp;
101      if ((astfp = fopen("astfp.txt", "w"))) {
102          AST_print(astroot, astfp);
103          printf("\nSaved astfp.txt\n");
104      }
105
106      /* save our output */
107      FILE *output;
108      if (!(output = fopen("output.s", "w"))) {
109          printf("\nUNABLE TO SAVE\n");
110          pccleanupsymtab();
111          return EXIT_FAILURE;
112      }
113
114      /* print our symbol table */
115      pcprintsymtab();
116
117      pcicg_start(output);
118      AST_cleanup(&astroot);
119
120      /* print our symbol table */
121      pccleanupsymtab();
122  #endif /* YYCOMPILE */
123
124      return EXIT_SUCCESS;
125  }
```

```
 1 /*
 2  * @author Derek Trom
 3  * @author Elena Corpus
 4  * compiler.h is the main entry point of the program and is
    responsible
 5  * for handling user input and running the other portions
   of the compiler,
 6  * such as the scanner and parser.
 7  */
 8
 9
10
11
12 #ifndef COMPILER_H
13 #define COMPILER_H
14
15 /* MACRO for debug printing */
16 #ifdef DEBUG
17 # define DEBUG_PRINTF(x) printf x
18 #else
19 # define DEBUG_PRINTF(x) do {} while (0)
20 #endif
21
22 /*
23 Transforms a string to a lower-case alternative.
24 Assumes that the string is NUL-terminated.
25
26 @param s the string to turn to lowercase
27 @return pointer to the front of s
28 */
29 char *strtolower(char *s);
30
31 #endif /* COMPILER_H */
```

```
 1 start   : program./output.s
 2 program : PROGRAM ID (ID,ID) SEMICOLON block PERIOD
 3 block   : constant-definition-part
 4           /*type-definition-part*/
 5           variable-declaration-part
 6           procedure-and-function-declaration-part
 7           statement-part
 8
 9 constant-definition-part      : CONST constant-definition
    | ε
10 constant-definition           : ID constant-definition-
   variable EQUALS constant-no-id SEMICOLON constant-
   definition-recursive
11 constant-definition-variable  : COMMA ID constant-
   definition-variable | ε
12 constant-definition-recursive : constant-definition | ε
13
14 type-definition-part      : TYPE type-definition | ε
15 type-definition           : ID EQUALS type type-definition-
   recursive
16 type-definition-recursive : SEMICOLON type-definition | ε
17
18 variable-definition-part      : VAR variable-declaration |
   ε
19 variable-definition           : ID variable-definition-
   variable COLON type SEMICOLON variable-definition-recursive
    | ε
20 variable-definition-variable  : COMMA ID variable-
   definition-variable | ε
21 variable-definition-recursive : variable-definition | ε
22
23 type            : simple-type /*| array-type*/
24 array-type      : ARRAY LBRACK index-type RBRACK OF simple-
   type
25 index-type      : ID | index-range
26 index-constant  : sign INTVAL | CHARVAL | sign constant-
   name
27 index-range     : index-constant DOTDOT index-constant
28 simple-type     : STRING | INTEGER | REAL | CHAR
29 constant-name   : ID
30 sign            : ADD | MINUS | ε
31
32 procedure-and-function-definition-part : procedure-
   declaration SEMICOLON | function-declaration SEMICOLON | ε
33 procedure-declaration : PROCEDURE ID
34                         LPAREN formal-parameters RPAREN
   SEMICOLON
35                         block
36                         procedure-and-function-definition-
   part
```

```
37
38 formal-parameters           : ID formal-parameters-variable
    COLON type
39 formal-parameters-variable  : COMMA ID formal-parameters-
   variable | ε
40
41 function-declaration  : FUNCTION ID
42                         LPAREN formal-parameters RPAREN
43                         COLON type SEMICOLON
44                         block
45                         procedure-and-function-declaration-
   part
46
47 statement-part      : compound-statement
48 compount-statement  : BEGIN statement statement-recursive
   SEMICOLON END
49 statement           : simple-statement | structured-
   statement
50 statement-recursive : SEMICOLON statement statement-
   recursive | ε
51 simple-statement    : assignment-statement | procedure-
   statement | application | read-statement | write-statement
52
53 assignment-statement      : variable ASSIGN expression
54 procedure-statement       : ID
55 application               : ID LPAREN expression
   application-recursive RPAREN
56 application-recursive     : COMMA expression application-
   recursive | ε
57 read-statement            : READ read-statement-part |
   READLN read-statement-part
58 read-statement-part       : LPAREN ID read-statement-
   recursive RPAREN
59 read-statement-recursive  : COMMA ID read-statement-
   recursive | ε
60 write-statement           : WRITE write-statement-part |
   WRITELN write-statement-part
61 write-statement-part      : LPAREN expression write-
   statement-recursive RPAREN
62 write-statement-recursive : COMMA expression write-
   statement-recursive | ε
63
64 structured-statement  : compound-statement | if-statement
    | while-statement | for-statement
65 if-statement          : IF expression THEN statement if-
   statement-else
66 if-statement-else     : ELSE statement | ε
67 while-statement       : WHILE expression DO statement
68 for-statement         : FOR ID ASSIGN expression for-
   statement-to expression DO statement
```

```
 69 for-statement-to      : TO | DOWNTO
 70
 71 expression            : simple-expression expression-
    relational
 72 simple-expression     : sign term expression-add
 73 expression-add        : add-term term expression-add | ε
 74 add-term              : ADD | MINUS | OR
 75
 76 term                  : factor term-mult
 77 term-mult             : mult-term factor term-mult | ε
 78 mult-term             : MULT | IDIV | DIV | AND
 79 factor                : application | variable | constant
     | NOT factor
 80
 81 expression-relational : relational-operator simple-
    expression | ε
 82 relational-operator   : EQ | NEQ | LT | LTE | GTE | GT
 83
 84 variable              : ID | ID LBRACK expression RBRACK
 85 paramenter-identifier : ID
 86
 87 constant              : constant-no-id | sign constant-
    identifier
 88 constant-no-id        : constant-number | sign constant-
    identifier | CHARVAL | STRINGVAL
 89 constant-number       : sign INTEGERNO | sign REALNO
 90 constant-identifier   : ID
```

```
 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 program Fibonacci;
 2 const
 3   question = 'How many Fibonacci numbers? ';
 4   msgInline = 'Inline While-Loop';
 5   msgFunction = 'Functional While-Loop';
 6   msgRecursive = 'Recursive';
 7
 8 var
 9   n, cur, temp, f1, f2 : integer;
10
11 (* Recursive Fibonacci printing. *)
12 procedure recursiveFibonacci (n, f1, f2 : integer);
13 begin
14   if n > 0 then begin
15     write(' ');
16
17     if f1 = 0 then begin
18       write(1);
19       recursiveFibonacci(n - 1, 1, 0);
20     end
21     else begin
22       if f2 = 0 then begin
23         write(1);
24         recursiveFibonacci(n - 1, 1, 1);
25       end else begin
26         write(f1 + f2);
27         recursiveFibonacci(n - 1, f2, f1 + f2);
28       end;
29     end;
30   end;
31 end;
32
33 (* Gets the next Fibonacci value and returns it. *)
34 function nextFibonacci (f1, f2 : integer) : integer;
35 begin
36   if f1 = 0 then begin
37     nextFibonacci := 1;
38   end
39   else begin
40     if f2 = 0 then begin
41       nextFibonacci := 1;
42     end else begin
43       nextFibonacci := f1 + f2;
44     end;
45   end;
46 end;
47
48 begin
49   // get user input
50   write(question);
```

```
 51   read(n);
 52   writeln(' ');
 53   writeln(msgInline);
 54
 55   // while loop test with inline computation
 56   cur := 0;
 57   while cur < n do begin
 58     if cur <= 1 then begin
 59       write(' ');
 60       write(1);
 61       f1 := 1;
 62       f2 := 1;
 63     end else begin
 64       // get the new value into f2 and old f2 into f1
 65       temp := f2;
 66       f2 := f1 + f2;
 67       f1 := temp;
 68
 69       write(' ');
 70       write(f2);
 71     end;
 72
 73     // increment current
 74     cur := cur + 1;
 75   end;
 76
 77   writeln(' ');
 78   writeln(' ');
 79   writeln(msgFunction);
 80
 81   // for-loop test with functional computation
 82   cur := 0;
 83   f1 := 0;
 84   f2 := 0;
 85   while cur < n do begin
 86     // calculate via our function
 87     temp := nextFibonacci(f1, f2);
 88     write(' ');
 89     write(temp);
 90
 91     // swap the values
 92     f1 := f2;
 93     f2 := temp;
 94
 95     cur := cur + 1;
 96   end;
 97
 98   // RECURSIVE
 99   writeln(' ');
100   writeln(' ');
```

```
101   writeln(msgRecursive);
102   recursiveFibonacci(n, 0, 0);
103 end.
```

```
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.
```