```
/* symbol table entry */
struct id_entry {
   struct id_entry *i_link;     /* pointer to next entry on hash chain */
   char *i_name;                /* pointer to name in string table */
   int i_type;                  /* type code */
   int i_blevel;                /* block level */
   int i_defined;               /* non-zero if identifier is declared */
   int i_width;                 /* number of words occupied */
   int i_scope;                 /* scope */
   int i_offset;                /* offset in activation frame */
};

/* scopes *** do not rearrange *** */
#define LOCAL 0
#define PARAM 1
#define GLOBAL 2

/* internal types *** do not rearrange *** */
#define T_INT    (1<<0)         /* integer */
#define T_STR    (1<<1)         /* string */
#define T_DOUBLE (1<<2)         /* double */
#define T_PROC   (1<<3)         /* procedure */
#define T_ARRAY  (1<<4)         /* array */
#define T_ADDR   (1<<5)         /* address */
#define T_LBL    (1<<6)         /* label */

/* semantic record */
struct sem_rec {
   int s_place;                 /* triple number */
   int s_mode;                  /* type */
   union {
       struct sem_rec *s_link;  /* used for backpatching */
       struct sem_rec *s_true;  /* true backpatch list */
   } back;
   struct sem_rec *s_false;     /* false backpatch list */
};
```

```
%{
# include <stdio.h>
# include "cc.h"
# include "scan.h"
# include "semutil.h"
# include "sem.h"
# include "sym.h"
%}

%union {
   int inttype;
   char *str_ptr;
   struct sem_rec *rec_ptr;
   struct id_entry *id_ptr;
   }
%token <str_ptr> ID CON STR
%token CHAR ELSE DOUBLE FOR IF INT RESERVED RETURN WHILE DO CONTINUE BREAK GOTO
%right LVAL
%right SET SETOR SETXOR SETAND SETLSH SETRSH SETADD SETSUB SETMUL SETDIV SETMOD
%left  OR
%left  AND
%left  BITOR
%left  BITXOR
%left  BITAND
%left  EQ NE
%left  GT GE LT LE
%left  LSH RSH
%left  ADD SUB
%left  MUL DIV MOD
%right UNARY NOT COM
%type <rec_ptr> lval expr exprs cexpr lblstmt stmt stmts block n func s
%type <rec_ptr> cexpro expro fargs dcls prog
%type <id_ptr> dcl dclr fhead fname args
%type <inttype> type m
%%
prog    : externs               {}
        ;

externs :                       {}
        | externs extern        {}
        ;

extern  : dcl ';'               {}
        | func                  {}
        ;

dcls    :                       {}
        | dcls dcl ';'          {}
        ;

dcl     : type dclr             { $$ = dcl($2, $1, 0); }
        | dcl ',' dclr          { $$ = dcl($3, $1->i_type&~T_ARRAY, 0); }
        ;

dclr    : ID                    { $$ = dclr($1, 0, 1); }
        | ID '[' ']'            { $$ = dclr($1, T_ARRAY, 1); }
        | ID '[' CON ']'        { $$ = dclr($1, T_ARRAY, atoi($3)); }
        ;

type    : CHAR                  { $$ = T_INT; }
        | DOUBLE                { $$ = T_DOUBLE; }
        | INT                   { $$ = T_INT; }
        ;

func    : fhead stmts '}' m     { $$ = ftail($1, $2, $4); }
        ;

fhead   : fname fargs '{' dcls  { $$ = fhead($1); }
        ;
```



```
fname   : type ID               { $$ = fname($1, $2); }
        | ID                    { $$ = fname(T_INT, $1); }
        ;

fargs   : '(' ')'               { enterblock(); }
        | '(' args ')'          { enterblock(); }
        ;

args    : type dclr             { $$ = dcl($2, $1, PARAM); }
        | args ',' type dclr    { $$ = dcl($4, $3, PARAM); }
        ;

s       :                       { startloopscope(); }
        ;

m       :                       { $$ = m(); }
        ;

n       :                       { $$ = n(); }
        ;

block   : '{' stmts '}'         { $$ = $2; }
        ;

stmts   :                       { $$ = 0; }
        | stmts m lblstmt       { $$ = dostmts($1, $2, $3); }
        ;

lblstmt : stmt                  { $$ = $1; }
        | labels stmt           { $$ = $2; }
        ;

labels  : ID ':'                { labeldcl($1); }
        | labels ID ':'         { labeldcl($2); }
        ;

stmt    : expr ';'
                { $$ = 0; }
        | IF '(' cexpr ')' m stmt
                { $$ = doif($3, $5, $6); }
        | IF '(' cexpr ')' m stmt ELSE n m stmt
                { $$ = doifelse($3, $5, $6, $8, $9, $10); }
        | WHILE '(' m cexpr ')' m s stmt
                { $$ = dowhile($3, $4, $6, $8); }
        | DO m s stmt WHILE '(' m cexpr ')' ';'
                { $$ = dodo($2, $4, $7, $8); }
        | FOR '(' expro ';' m cexpro ';' m expro n ')' m s stmt
                { $$ = dofor($5, $6, $8, $10, $12, $14); }
        | CONTINUE ';'
                { $$ = docontinue(); }
        | BREAK ';'
                { $$ = dobreak(); }
        | GOTO ID ';'
                { $$ = dogoto($2); }
        | RETURN ';'
                { $$ = doret((struct sem_rec *) NULL); }
        | RETURN expr ';'
                { $$ = doret($2); }
        | block
                { $$ = $1; }
        | ';'
                { $$ = 0; }
        ;

cexpro  :                       { $$ = node(0, 0, n(), 0); }
        | cexpr                 {}
        ;
```

```
cexpr   : expr EQ expr          { $$ = rel("==", $1, $3); }
        | expr NE expr          { $$ = rel("!=", $1, $3); }
        | expr LE expr          { $$ = rel("<=", $1, $3); }
        | expr GE expr          { $$ = rel(">=", $1, $3); }
        | expr LT expr          { $$ = rel("<",  $1, $3); }
        | expr GT expr          { $$ = rel(">",  $1, $3); }
        | cexpr AND m cexpr     { $$ = ccand($1, $3, $4); }
        | cexpr OR m cexpr      { $$ = ccor($1, $3, $4); }
        | NOT cexpr             { $$ = ccnot($2); }
        | expr                  { $$ = ccexpr($1); }
        ;

exprs   : expr                  { $$ = $1; }
        | exprs ',' expr        { $$ = exprs($1, $3); }
        ;

expro   :                       {}
        | expr                  {}
        ;

expr    : lval SET expr         { $$ = set("",   $1, $3); }
        | lval SETOR expr       { $$ = set("|",  $1, $3); }
        | lval SETXOR expr      { $$ = set("^",  $1, $3); }
        | lval SETAND expr      { $$ = set("&",  $1, $3); }
        | lval SETLSH expr      { $$ = set("<<", $1, $3); }
        | lval SETRSH expr      { $$ = set(">>", $1, $3); }
        | lval SETADD expr      { $$ = set("+",  $1, $3); }
        | lval SETSUB expr      { $$ = set("-",  $1, $3); }
        | lval SETMUL expr      { $$ = set("*",  $1, $3); }
        | lval SETDIV expr      { $$ = set("/",  $1, $3); }
        | lval SETMOD expr      { $$ = set("%",  $1, $3); }
        | expr BITOR expr       { $$ = opb("|",  $1, $3); }
        | expr BITXOR expr      { $$ = opb("^",  $1, $3); }
        | expr BITAND expr      { $$ = opb("&",  $1, $3); }
        | expr LSH expr         { $$ = opb("<<", $1, $3); }
        | expr RSH expr         { $$ = opb(">>", $1, $3); }
        | expr MOD expr         { $$ = opb("%",  $1, $3); }
        | expr ADD expr         { $$ = op2("+",  $1, $3); }
        | expr SUB expr         { $$ = op2("-",  $1, $3); }
        | expr MUL expr         { $$ = op2("*",  $1, $3); }
        | expr DIV expr         { $$ = op2("/",  $1, $3); }
        | BITAND lval %prec UNARY
                                { $$ = $2; }
        | SUB expr %prec UNARY  { $$ = op1("-",      $2); }
        | COM expr              { $$ = op1("~",      $2); }
        | lval %prec LVAL       { $$ = op1("@",      $1); }
        | ID '(' ')'            { $$ = call($1, (struct sem_rec *) NULL); }
        | ID '(' exprs ')'      { $$ = call($1, $3); }
        | '(' expr ')'          { $$ = $2; }
        | CON                   { $$ = con($1); }
        | STR                   { $$ = string($1); }
        ;

lval    : ID                    { $$ = id($1); }
        | ID '[' expr ']'       { $$ = index(id($1), $3); }
        ;
%%
# include <stdio.h>

extern int lineno;

/*
 * main - read a program, and parse it
 */
main(int argc, char *argv)
{

   enterblock();
   initlex();
```



```
   enterblock();
   if (yyparse())
      yyerror("syntax error");
   exit(0);
}

/*
 * yyerror - issue error message
 */
yyerror(char msg[])
{
   fprintf(stderr, " %s. Line %d\n", msg, lineno);
}
```

```
void initlex();
void initrw(int, char *);
int yylex();
void skip();
void comment();
int istype(int);
void putbak(int);
int quote(char []);
```

```
# include <stdio.h>
# include <ctype.h>
# include "y.tab.h"
# include "cc.h"
# include "scan.h"
# include "sym.h"

# define MAXTOK 100     /* maximum token size */

# define LETTER 'a'
# define DIGIT '0'

int lineno = 1;         /* current line number */

/*
 * initlex - initialize lexical analyzer
 */
void initlex()
{
   initrw(SET, "=");
   initrw(SETOR, "|=");
   initrw(SETXOR, "^=");
   initrw(SETAND, "&=");
   initrw(SETLSH, "<<=");
   initrw(SETRSH, ">>=");
   initrw(SETADD, "+=");
   initrw(SETSUB, "-=");
   initrw(SETMUL, "*=");
   initrw(SETDIV, "/=");
   initrw(SETMOD, "%=");
   initrw(OR, "||");
   initrw(AND, "&&");
   initrw(BITOR, "|");
   initrw(BITXOR, "^");
   initrw(BITAND, "&");
   initrw(EQ, "==");
   initrw(NE, "!=");
   initrw(GT, ">");
   initrw(GE, ">=");
   initrw(LT, "<");
   initrw(LE, "<=");
   initrw(LSH, "<<");
   initrw(RSH, ">>");
   initrw(ADD, "+");
   initrw(SUB, "-");
   initrw(MUL, "*");
   initrw(DIV, "/");
   initrw(MOD, "%");
   initrw(NOT, "!");
   initrw(COM, "~");
   initrw(IF, "if");
   initrw(FOR, "for");
   initrw(ELSE, "else");
   initrw(WHILE, "while");
   initrw(DO, "do");
   initrw(RETURN, "return");
   initrw(CONTINUE, "continue");
   initrw(BREAK, "break");
   initrw(GOTO, "goto");
   initrw(CHAR, "char");
   initrw(DOUBLE, "double");
   initrw(INT, "int");
}

/*
 * initrw - initialize a reserved word entry
 */
void initrw(int k, char *s)
{
```



```
   struct id_entry *p;

   p = install(slookup(s), 1);
   p->i_type = k;
   p->i_defined = 1;
}

/*
 * yylex - fetch next token
 */
int yylex()
{
   int c, i, type;
   char lin[MAXTOK];
   struct id_entry *p;

   i = 0;
   type = RESERVED;
   skip();
   switch (istype(c = getchar())) {
      case EOF:
         return (-1);
         break;
      case LETTER:
         putbak(c);
         while ((isalpha(c = getchar()) || isdigit(c) || c == '_') &&
                i < MAXTOK)
            lin[i++] = c;
         putbak(c);
         type = ID;
         break;
      case DIGIT:
         putbak(c);
         while (isdigit(c = getchar()) && i < MAXTOK)
            lin[i++] = c;
         putbak(c);
         type = CON;
         break;
      case '(': case ')': case ',': case '.': case ':':
      case ';': case '?': case '[': case ']': case '{':
      case '}':
         type = lin[i++] = c;
         break;
      case '~':
         lin[i++] = c;
         break;
      case '!': case '%': case '*': case '/': case '^': case '=':
         lin[i++] = c;
         c = getchar();
         if (c == '=')
            lin[i++] = c;
         else
            putbak(c);
         break;
      case '&': case '+': case '-': case '|':
         lin[i++] = c;
         c = getchar();
         if (c == '=' || c == lin[i-1])
            lin[i++] = c;
         else
            putbak(c);
         break;
      case '<': case '>':
         lin[i++] = c;
         c = getchar();
         if (c == lin[i-1]) {
            lin[i++] = c;
            c = getchar();
         }
```

```
         if (c == '=')
            lin[i++] = c;
         else
            putbak(c);
         break;
      case '"':
         i = quote(lin);
         type = STR;
         break;
      default:
        fprintf(stderr, "illegal character: %o\n", c);
        return (yylex());
   }
   lin[i] = 0;
   p = lookup(yylval.str_ptr = slookup(lin), 0);
   if (p != NULL && p->i_blevel == 1)
      type = p->i_type;
   return (type);
}

/*
 * skip - eat blanks, comments
 */
void skip()
{
   int c1, c2;

   c1 = getchar();
   for (;;)
      if (isspace(c1)) {
         if (c1 == '\n')
            lineno++;
         c1 = getchar();
      }
      else if (c1 == '/' && (c2 = getchar()) == '*') {
         comment();
         c1 = getchar();
      }
      else if (c1 == '/') {
         putbak(c2);
         break;
      }
      else
         break;
   putbak(c1);
}

/*
 * comment - skip over comment
 */
void comment()
{
   int c1, c2;

   for (;;) {
      c1 = getchar();
      if (c1 == EOF)
         break;
      else if (c1 == '*' && (c2 = getchar()) == '/')
         break;
      else if (c1 == '*')
         putbak(c2);
      else if (c1 == '\n')
         lineno++;
   }
}

/*
 * istype - classify character
```



```
 */
int istype(int c)
{
   if (isalpha(c))
      return(LETTER);
   else if (isdigit(c))
      return(DIGIT);
   else
      return(c);
}

/*
 * putbak - push character back onto input
 */
void putbak(int c)
{
   if (c != EOF)
      ungetc(c, stdin);
}

/*
 * quote - get quoted string
 */
int quote(char lin[])
{
   int c, i, j, peek;

   i = j = 0;

   /* copy quote */
   lin[i++] = '"';

   /* get rest of string */
   c = getchar();
   for (;;) {
      if (c == '"') {
         lin[i++] = c;
         break;
      }

      /* supply a missing quote if needed */
      if (c == EOF) {
         fprintf(stderr, "missing quote\n");
         lin[i++] = '"';
         break;
      }

      /* handle escaped characters */
      if (c == '\\') {
         peek = getchar();

         /* ignore escaped newline */
         if (peek == '\n') {
            c = getchar();
            lineno++;
            continue;
         }
         else
            putbak(peek);
      }

      /* count characters inside string */
      j++;

      /* copy next char */
      lin[i] = c;

      /* get the next character */
      c = getchar();
```

```
      /* copy escaped char */
      if (lin[i++] == '\\' && c != EOF) {
         lin[i++] = c;
         c = getchar();
      }
   }

   /* terminate the string */
   lin[i++] = '\0';
   return (i);
}
```

```
int currtrip();
struct id_entry *dcl(struct id_entry *, int, int);
struct id_entry *dclr(char *, int, int);
struct sem_rec *merge(struct sem_rec *, struct sem_rec *);
int nexttrip();
struct sem_rec *node(int, int, struct sem_rec *, struct sem_rec *);
int tsize(int);
```

```
# include <stdlib.h>
# include <stdio.h>
# include "cc.h"
# include "sem.h"
# include "sym.h"
# define MAXARGS 50
# define MAXLOCS 50

int ntmp = 0;                /* last triple number         */
int formalnum;               /* number of formal arguments */
char formaltypes[MAXARGS];   /* types of formal arguments  */
int localnum;                /* number of local variables  */
char localtypes[MAXLOCS];    /* types of local variables   */
int localwidths[MAXLOCS];    /* widths of local variables  */

extern struct sem_rec **top;

/*
 * currtrip - returns the current triple number
 */
int currtrip()
{
   return ntmp;
}

/*
 * dcl - adjust the offset or allocate space for a global
 */
struct id_entry *dcl(struct id_entry *p, int type, int scope)
{
   extern int level;

   p->i_type += type;
   if (scope != 0)
      p->i_scope = scope;
   else if (p->i_width > 0 && level == 2)
      p->i_scope = GLOBAL;
   else
      p->i_scope = LOCAL;
   if (level > 2 && p->i_scope == PARAM) {
      p->i_offset = formalnum;
      if (p->i_type == T_DOUBLE)
         formaltypes[formalnum++] = 'f';
      else
         formaltypes[formalnum++] = 'i';
      if (formalnum > MAXARGS) {
         fprintf(stderr, "too many arguments\n");
         exit(1);
      }
   }
   else if (level > 2 && p->i_scope != PARAM) {
      p->i_offset = localnum;
      localwidths[localnum] = p->i_width;
      if (p->i_type & T_DOUBLE)
         localtypes[localnum++] = 'f';
      else
         localtypes[localnum++] = 'i';
      if (localnum > MAXLOCS) {
         fprintf(stderr, "too many locals\n");
         exit(1);
      }
   }
   else if (p->i_width > 0 && level == 2)
      printf("%d.\talloc %s %d\n", nexttrip(), p->i_name,
             p->i_width * tsize(p->i_type&~T_ARRAY));
   return (p);
}

/*
```



```
 * dclr - insert attributes for a declaration
 */
struct id_entry *dclr(char *name, int type, int width)
{
   struct id_entry *p;
   extern int level;
   char msg[80];

   if ((p = lookup(name, 0)) == NULL || p->i_blevel != level)
      p = install(name, -1);
   else  {
      sprintf(msg, "identifier %s previously declared", name);
      yyerror(msg);
      return (p);
   }
   p->i_defined = 1;
   p->i_type = type;
   p->i_width = width;
   return (p);
}

/*
 * merge - merge backpatch lists p1 and p2
 */
struct sem_rec *merge(struct sem_rec *p1, struct sem_rec *p2)
{
   struct sem_rec *p;

   if (p1 == NULL)
      return (p2);
   if (p2 == NULL)
      return (p1);
   for (p = p1; p->back.s_link; p = p->back.s_link)
      ;
   p->back.s_link = p2;
   return (p1);
}

/*
 * nexttrip - increments the triple number and returns it
 */
int nexttrip()
{
   return ++ntmp;
}


/*
 * node - allocate a semantic node with fields a, b, c, d
 */
struct sem_rec *node(int a, int b, struct sem_rec *c, struct sem_rec *d)
{
   struct sem_rec *t;

   /* allocate space */
   t = (struct sem_rec *) alloc(sizeof(struct sem_rec));

   /* save semantic record */
   save_rec(t);

   /* fill in the fields */
   t->s_place = a;
   t->s_mode = b;
   t->back.s_link = c;
   t->s_false = d;
   return (t);
}

/*
```

```
 * tsize - return size of type
 */
int tsize(int type)
{
   if (type == T_INT)
      return(4);
   else if (type == T_DOUBLE)
      return(8);
   else
      return(0);
}
```

```
void dump(int, FILE *);
void new_block();
void exit_block();
void enterblock();
struct id_entry *install(char *, int);
void leaveblock();
struct id_entry *lookup(char *, int);
void sdump(FILE *);
char *slookup(char []);
int hash(char *);
char *alloc(unsigned);
void save_rec(struct sem_rec *);
```

```
/* symbol table management */

# include <stdlib.h>
# include <stdio.h>
# include <malloc.h>
# include <string.h>
# include "cc.h"
# include "sym.h"

# define STABSIZE 119                    /* hash table size for strings */
# define ITABSIZE 37                     /* hash table size for identifiers */
# define MAXSTK   1000

struct sem_rec *stk[MAXSTK];             /* stack of ptrs to semantic recs */
struct sem_rec **top = stk;              /* stack pointer */
struct sem_rec **prevtop = NULL;         /* previous top */

int numrecs = 0;                         /* number of semantic recs */

int level = 0;                           /* current block level */

struct s_chain {
   char *s_ptr;                          /* string pointer */
   struct s_chain *s_next;               /* next in chain */
   } *str_table[STABSIZE] = {0};         /* string hash table */

struct id_entry *id_table[ITABSIZE] = {0}; /* identifier hash table */

/*
 *  dump - dump identifiers with block level >= blev to f
 */
void dump(int blev, FILE *f)
{
   struct id_entry **i, *p;

   fprintf(f, "Dumping identifier table\n");
   for (i = id_table; i < &id_table[ITABSIZE]; i++)
      for (p = *i; p; p = p->i_link)
         if (p->i_blevel >= blev)
            fprintf(f, "%s\t%d\t%d\t%d\n", p->i_name, p->i_blevel,
             p->i_type, p->i_defined);
}

/*
 * new_block - save previous stack top and mark a new one
 */
void new_block()
{
   save_rec((struct sem_rec *) prevtop);
   prevtop = top - 1;
}

/*
 * exit_block - exit block, free up semantic records
 */
void exit_block()
{
   for (top--; top > prevtop;) {
      numrecs--;
      free((char *) *top--);
   }
   prevtop = (struct sem_rec **) *top;
}

/*
 * enterblock - enter a new block
 */
void enterblock()
{
```



```
   new_block();
   level++;
}

/*
 * install - install name with block level blev, return ptr
 */
struct id_entry *install(char *name, int blev)
{
   struct id_entry *ip, **q;

   if (blev < 0)
      blev = level;

   /* allocate space */
   ip = (struct id_entry *) alloc(sizeof(struct id_entry));

   /* set fields of symbol table */
   ip->i_name = name;
   ip->i_blevel = blev;
   for (q = &id_table[hash(name)%ITABSIZE]; *q; q = &((*q)->i_link))
      if (blev >= (*q)->i_blevel)
         break;
   ip->i_link = *q;
   *q = ip;
   return (ip);
}

/*
 * leaveblock - exit a block
 */
void leaveblock()
{
   struct id_entry **i, *p, *tmp;

   if (level > 0) {
      for (i = id_table; i < &id_table[ITABSIZE]; i++) {
         for (p = *i; p; p = tmp)
            if (p->i_blevel < level)
               break;
            else {
               tmp = p->i_link;
               cfree(p);
            }
         *i = p;
      }
      level--;
   }
   exit_block();
}

/*
 * lookup - lookup name, return ptr; use default scope if blev == 0
 */
struct id_entry *lookup(char *name, int blev)
{
   struct id_entry *p;

   for (p = id_table[hash(name)%ITABSIZE]; p; p = p->i_link)
      if (name == p->i_name && (blev == 0 || blev == p->i_blevel))
         return (p);
   return (NULL);
}

/*
 * sdump - dump string table to f
 */
void sdump(FILE *f)
{
```

```
   struct s_chain **s, *p;

   fprintf(f, "Dumping string table\n");
   for (s = str_table; s < &str_table[STABSIZE]; s++)
      for (p = *s; p; p = p->s_next)
         fprintf(f, "%s\n", p->s_ptr);
}

/*
 * slookup - lookup str in string table, install if necessary, return ptr
 */
char *slookup(char str[])
{
   struct s_chain *p;
   int i, k;

   for (k = i = 0; i < 5; i++)  /* simple hash function */
      if (str[i])
         k += str[i];
      else
         break;

   k %= STABSIZE;
   for (p = str_table[k]; p; p = p->s_next)
      if (strcmp(str, p->s_ptr) == 0)
         return (p->s_ptr);
   p = (struct s_chain *) alloc(sizeof(struct s_chain));
   p->s_next = str_table[k];
   str_table[k] = p;
   p->s_ptr = (char *) alloc((unsigned) strlen(str) + 1);
   p->s_ptr = strcpy(p->s_ptr, str);
   return (p->s_ptr);
}

/*
 * hash - hash name, turn address into hash number
 */
int hash(char *s)
{
   return((int ) s);
}

/*
 * alloc - alloc space
 */
char *alloc(unsigned n)
{
   char *p;

   if ((p = calloc(1, n)) == NULL) {
      yyerror("csem: out of space");
      exit (1);
   }
   return (p);
}

/*
 * save_rec - save a semantic record so it can be reclaimed later
 */
void save_rec(struct sem_rec *s)
{
   /* save on stack so can reclaim */
   if (numrecs++ > MAXSTK) {
      fprintf(stderr, "too many semantic records\n");
      exit(1);
   }
   *top++ = s;
}
```

```
LIB=/home/faculty/whalley/asg5
CFLAGS= -I$(LIB) -I. -c -g
CC=gcc

csem: sym.o scan.o sem.o semutil.o cgram.o
        $(CC) -g -o csem sym.o scan.o sem.o semutil.o cgram.o

sym.o: $(LIB)/sym.c $(LIB)/cc.h $(LIB)/sym.h
        $(CC) $(CFLAGS) $(LIB)/sym.c

scan.o: $(LIB)/scan.c $(LIB)/cc.h $(LIB)/scan.h $(LIB)/sym.h y.tab.h
        $(CC) $(CFLAGS) $(LIB)/scan.c

sem.o: sem.c $(LIB)/cc.h $(LIB)/sem.h $(LIB)/semutil.h $(LIB)/sym.h
        $(CC) $(CFLAGS) sem.c

semutil.o: $(LIB)/semutil.c $(LIB)/cc.h $(LIB)/sem.h $(LIB)/sym.h
        $(CC) $(CFLAGS) $(LIB)/semutil.c

cgram.o: cgram.c $(LIB)/cc.h $(LIB)/scan.h $(LIB)/sem.h $(LIB)/semutil.h $(LIB)/
sym.h
        $(CC) $(CFLAGS) cgram.c

y.tab.h cgram.c: $(LIB)/cgram.y
        yacc -vd $(LIB)/cgram.y
        mv y.tab.c cgram.c
```

```
void backpatch(struct sem_rec *, int);
struct sem_rec *call(char *, struct sem_rec *);
struct sem_rec *cast(struct sem_rec *, int);
struct sem_rec *ccand(struct sem_rec *, int, struct sem_rec *);
struct sem_rec *ccexpr(struct sem_rec *);
struct sem_rec *ccnot(struct sem_rec *);
struct sem_rec *ccor(struct sem_rec *, int, struct sem_rec *);
struct sem_rec *con(char *);
struct sem_rec *dobreak();
struct sem_rec *docontinue();
struct sem_rec *dodo(int, struct sem_rec *, int, struct sem_rec *);
struct sem_rec *dogoto(char *);
struct sem_rec *dofor(int, struct sem_rec *, int, struct sem_rec *,
                      int, struct sem_rec *);
struct sem_rec *doif(struct sem_rec *, int, struct sem_rec *);
struct sem_rec *doifelse(struct sem_rec *, int, struct sem_rec *,
                         struct sem_rec *, int, struct sem_rec *);
struct sem_rec *doret(struct sem_rec *);
struct sem_rec *dostmts(struct sem_rec *, int, struct sem_rec *);
struct sem_rec *dowhile(int m1, struct sem_rec *, int, struct sem_rec *);
void endloopscope(struct sem_rec *);
struct sem_rec *exprs(struct sem_rec *, struct sem_rec *);
struct id_entry *fhead(struct id_entry *);
struct id_entry *fname(int, char *);
struct sem_rec *ftail(struct id_entry *, struct sem_rec *, int);
struct sem_rec *gen(char *, struct sem_rec *, struct sem_rec *, int);
struct sem_rec *id(char *);
struct sem_rec *index(struct sem_rec *, struct sem_rec *);
void labeldcl(char *);
int m();
struct sem_rec *n();
struct sem_rec *op1(char *, struct sem_rec *);
struct sem_rec *op2(char *, struct sem_rec *, struct sem_rec *);
struct sem_rec *opb(char *, struct sem_rec *, struct sem_rec *);
struct sem_rec *rel(char *, struct sem_rec *, struct sem_rec *);
struct sem_rec *set(char *, struct sem_rec *, struct sem_rec *);
void startloopscope();
struct sem_rec *string(char *);
int tsize(int);
```

```
# include <stdio.h>
# include "cc.h"
# include "semutil.h"
# include "sem.h"
# include "sym.h"

/*
 * backpatch - backpatch list of triples starting at p with k
 */
void backpatch(struct sem_rec *p, int k)
{
   fprintf(stderr, "sem: backpatch not implemented\n");
}

/*
 * call - procedure invocation
 */
struct sem_rec *call(char *f, struct sem_rec *args)
{
   fprintf(stderr, "sem: call not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * ccand - logical and
 */
struct sem_rec *ccand(struct sem_rec *e1, int m, struct sem_rec *e2)
{
   fprintf(stderr, "sem: ccand not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * ccexpr - convert arithmetic expression to logical expression
 */
struct sem_rec *ccexpr(struct sem_rec *e)
{
   fprintf(stderr, "sem: ccexpr not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * ccnot - logical not
 */
struct sem_rec *ccnot(struct sem_rec *e)
{
   fprintf(stderr, "sem: ccnot not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * ccor - logical or
 */
struct sem_rec *ccor(struct sem_rec *e1, int m, struct sem_rec *e2)
{
   fprintf(stderr, "sem: ccor not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * con - constant reference in an expression
 */
struct sem_rec *con(char *x)
{
   fprintf(stderr, "sem: con not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
```



```
 * dobreak - break statement
 */
struct sem_rec *dobreak()
{
   fprintf(stderr, "sem: dobreak not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * docontinue - continue statement
 */
struct sem_rec *docontinue()
{
   fprintf(stderr, "sem: docontinue not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * dodo - do statement
 */
struct sem_rec *dodo(int m1, struct sem_rec *s, int m2, struct sem_rec *e)
{
   fprintf(stderr, "sem: dodo not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * dofor - for statement
 */
struct sem_rec *dofor(int m1, struct sem_rec *e2, int m2, struct sem_rec *n,
                      int m3, struct sem_rec *s)
{
   fprintf(stderr, "sem: dofor not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * dogoto - goto statement
 */
struct sem_rec *dogoto(char *id)
{
   fprintf(stderr, "sem: dogoto not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * doif - one-arm if statement
 */
struct sem_rec *doif(struct sem_rec *e, int m, struct sem_rec *s)
{

   fprintf(stderr, "sem: doif not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * doifelse - if then else statement
 */
struct sem_rec *doifelse(struct sem_rec *e, int m1, struct sem_rec *s1,
                         struct sem_rec *n, int m2, struct sem_rec *s2)
{
   fprintf(stderr, "sem: doifelse not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * doret - return statement
 */
struct sem_rec *doret(struct sem_rec *e)
```

```
{
   fprintf(stderr, "sem: doret not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * dostmts - statement list
 */
struct sem_rec *dostmts(struct sem_rec *sl, int m, struct sem_rec *s)
{

   fprintf(stderr, "sem: dostmts not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * dowhile - while statement
 */
struct sem_rec *dowhile(int m1, struct sem_rec *e, int m2, struct sem_rec *s)
{
   fprintf(stderr, "sem: dowhile not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * endloopscope - end the scope for a loop
 */
void endloopscope(struct sem_rec *p)
{
   fprintf(stderr, "sem: endloopscope not implemented\n");
}

/*
 * exprs - form a list of expressions
 */
struct sem_rec *exprs(struct sem_rec *l, struct sem_rec *e)
{
   fprintf(stderr, "sem: exprs not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * fhead - beginning of function body
 */
struct id_entry *fhead(struct id_entry *p)
{
   fprintf(stderr, "sem: fhead not implemented\n");
   return ((struct id_entry *) NULL);
}

/*
 * fname - function declaration
 */
struct id_entry *fname(int t, char *id)
{
   fprintf(stderr, "sem: fname not implemented\n");
   return ((struct id_entry *) NULL);
}

/*
 * ftail - end of function body
 */
struct sem_rec *ftail(struct id_entry *p, struct sem_rec *s, int m)
{
   fprintf(stderr, "sem: ftail not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
```



```
 * id - variable reference
 */
struct sem_rec *id(char *x)
{
   fprintf(stderr, "sem: id not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * index - subscript
 */
struct sem_rec *index(struct sem_rec *x, struct sem_rec *i)
{
   fprintf(stderr, "sem: index not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * labeldcl - process a label declaration
 */
void labeldcl(char *id)
{
   fprintf(stderr, "sem: labeldcl not implemented\n");
}

/*
 * m - generate label and return next triple number
 */
int m()
{
   fprintf(stderr, "sem: m not implemented\n");
   return (0);
}

/*
 * n - generate goto and return backpatch pointer
 */
struct sem_rec *n()
{
   fprintf(stderr, "sem: n not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * op1 - unary operators
 */
struct sem_rec *op1(char *op, struct sem_rec *x)
{
   fprintf(stderr, "sem: op1 not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * op2 - arithmetic operators
 */
struct sem_rec *op2(char *op, struct sem_rec *x, struct sem_rec *y)
{
   fprintf(stderr, "sem: op2 not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * opb - bitwise operators
 */
struct sem_rec *opb(char *op, struct sem_rec *x, struct sem_rec *y)
{
   fprintf(stderr, "sem: opb not implemented\n");
   return ((struct sem_rec *) NULL);
}
```

```
/*
 * rel - relational operators
 */
struct sem_rec *rel(char *op, struct sem_rec *x, struct sem_rec *y)
{
   fprintf(stderr, "sem: rel not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * set - assignment operators
 */
struct sem_rec *set(char *op, struct sem_rec *x, struct sem_rec *y)
{
   fprintf(stderr, "sem: set not implemented\n");
   return ((struct sem_rec *) NULL);
}

/*
 * startloopscope - start the scope for a loop
 */
void startloopscope()
{
   fprintf(stderr, "sem: startloopscope not implemented\n");
}

/*
 * string - generate code for a string
 */
struct sem_rec *string(char *s)
{
   fprintf(stderr, "sem: string not implemented\n");
   return ((struct sem_rec *) NULL);
}
```