

Appendix B

Source code for the Clang compiler/interpreter

This appendix gives the complete source code for a hand-crafted compiler for the Clang language as developed by the end of Chapter 18.

cln.cpp | misc.h | set.h | srce.h | srce.cpp | report.h | report.cpp | scan.h | scan.cpp | parser.h | parser.cpp | table.h | table.cpp | cgen.h | cgen.cpp | stkmc.h | stkmc.cpp

```
----- cln.cpp -----

// Clang Compiler/Interpreter
// P.D. Terry, Rhodes University, 1996

#include "misc.h"
#include "srce.h"
#include "scan.h"
#include "parser.h"
#include "table.h"
#include "report.h"
#include "stkmc.h"
#include "cgen.h"

#define usage "USAGE: CLN source [listing]\n"

static char SourceName[256], ListName[256], CodeName[256];

TABLE *Table;
CGEN *CGen;
STKMC *Machine;
REPORT *Report;

class clangReport : public REPORT {
public:
    clangReport(SRCE *S) { Srce = S; }
    virtual void error(int errorcode)
    { Srce->reporterror(errorcode); errors = true; }
private:
    SRCE *Srce;
};

class clangSource : public SRCE {
public:
    clangSource(char *sname, char *lname, char *ver, bool lw)
        : SRCE(sname, lname, ver, lw) {};
    virtual void startnewline() { fprintf(1st, "%4d : ", CGen->gettop()); }
};

void main(int argc, char *argv[])
{
    char reply;
    int codelength, initsp;

    // check on correct parameter usage
    if (argc == 1) { printf(usage); exit(1); }
    strcpy(SourceName, argv[1]);
    if (argc > 2) strcpy(ListName, argv[2]);
    else appendextension(SourceName, ".1st", ListName);

    clangSource *Source = new clangSource(SourceName, ListName, STKMC_version, true);
    Report = new clangReport(Source);
    CGen = new CGEN(Report);
    SCAN *Scanner = new SCAN(Source, Report);
    Table = new TABLE(Report);
    PARSER *Parser = new PARSER(CGen, Scanner, Table, Source, Report);
    Machine = new STKMC();

    Parser->parse();
    CGen->getsize(codelength, initsp);

    appendextension(SourceName, ".cod", CodeName);
}
```

```

Machine->listcode(CodeName, codelength);

if (Report->anyerrors())
    printf("Compilation failed\n");
else
{
    printf("Compilation successful\n");
    while (true)
    {
        printf("\nInterpret? (y/n) ");
        do
        {
            scanf("%c", &reply);
            while (toupper(reply) != 'N' && toupper(reply) != 'Y');
            if (toupper(reply) == 'N') break;
            scanf("%*[^\\n]"); getchar();
            Machine->interpret(codelength, initsp);
        }
    }
    delete Source;
    delete Scanner;
    delete Parser;
    delete Table;
    delete Report;
    delete CGen;
    delete Machine;
}

----- misc.h -----

// Various common items

#ifndef MISC_H
#define MISC_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <ctype.h>
#include <limits.h>

#define boolean int
#define bool int
#define true 1
#define false 0
#define TRUE 1
#define FALSE 0
#define maxint INT_MAX

#if __MSDOS__ || MSDOS || WIN32 || __WIN32__
# define pathsep '\\'
#else
# define pathsep '/'
#endif

static void appendextension (char *oldstr, char *ext, char *newstr)
// Changes filename in oldstr from PRIMARY.xxx to PRIMARY.ext in newstr
{
    int i;
    char old[256];
    strcpy(old, oldstr);
    i = strlen(old);
    while ((i > 0) && (old[i-1] != '.') && (old[i-1] != pathsep)) i--;
    if ((i > 0) && (old[i-1] == '.')) old[i-1] = 0;
    if (ext[0] == '.') sprintf(newstr, "%s%s", old, ext);
    else sprintf(newstr, "%s.%s", old, ext);
}

#endif /* MISC_H */

----- set.h -----

// simple set operations
// Thanks to E.P. Wentworth for useful ideas!

#ifndef SET_H
#define SET_H

template <int maxElem>
class Set {
public:
    Set() // Construct { }
    { clear(); }
    // { 0 .. maxElem }

```

```

Set(int e1)                                // Construct { e1 }
{ clear(); incl(e1); }

Set(int e1, int e2)                        // Construct { e1, e2 }
{ clear(); incl(e1); incl(e2); }

Set(int e1, int e2, int e3)               // Construct { e1, e2, e3 }
{ clear(); incl(e1); incl(e2); incl(e3); }

Set(int n, int e1[])                      // Construct { e[0] .. e[n-1] }
{ clear(); for (int i = 0; i < n; i++) incl(e1[i]); }

void incl(int e)                          // Include e
{ if (e >= 0 && e <= maxElem) bits[wrđ(e)] |= bitmask(e); }

void excl(int e)                          // Exclude e
{ if (e >= 0 && e <= maxElem) bits[wrđ(e)] &= ~bitmask(e); }

int memb(int e)                           // Test membership for e
{ if (e >= 0 && e <= maxElem) return((bits[wrđ(e)] & bitmask(e)) != 0);
  else return 0;
}

int isempty(void)                        // Test for empty set
{ for (int i = 0; i < length; i++) if (bits[i]) return 0;
  return 1;
}

Set operator + (const Set &s)            // Union with s
{ Set<maxElem> r;
  for (int i = 0; i < length; i++) r.bits[i] = bits[i] | s.bits[i];
  return r;
}

Set operator * (const Set &s)            // Intersection with s
{ Set<maxElem> r;
  for (int i = 0; i < length; i++) r.bits[i] = bits[i] & s.bits[i];
  return r;
}

Set operator - (const Set &s)            // Difference with s
{ Set<maxElem> r;
  for (int i = 0; i < length; i++) r.bits[i] = bits[i] & ~s.bits[i];
  return r;
}

Set operator / (const Set &s)            // Symmetric difference with s
{ Set<maxElem> r;
  for (int i = 0; i < length; i++) r.bits[i] = bits[i] ^ s.bits[i];
  return r;
}

private:
  unsigned char    bits[(maxElem + 8) / 8];
  int              length;
  int wrđ(int i)    { return(i / 8); }
  int bitmask(int i) { return(1 << (i % 8)); }
  void clear()      { length = (maxElem + 8) / 8;
                    for (int i = 0; i < length; i++) bits[i] = 0;
  }
};

#endif /* SET_H */

```

----- srce.h -----

```

// Source handler for various parsers/compilers
// P.D. Terry, Rhodes University, 1996

#ifndef SRCE_H
#define SRCE_H

#include "misc.h"

const int linemax = 129;          // limit on source line length

class SRCE {
public:
  FILE *lst;                     // listing file
  char ch;                       // latest character read

```

```

void nextch(void);
// Returns ch as the next character on this source line, reading a new
// line where necessary.  ch is returned as NUL if src is exhausted.

bool endline(void)      { return (charpos == linelength); }
// Returns true when end of current line has been reached

void listingon(void)     { listing = true; }
// Requests source to be listed as it is read

void listingoff(void)    { listing = false; }
// Requests source not to be listed as it is read

void reporterror(int errorcode);
// Points out error identified by errorcode with suitable message

virtual void startnewline() {}
// called at start of each line

int getline(void)        { return linenumber; }
// returns current line number

SRCE(char *sourcename, char *listname, char *version, bool listwanted);
// Open src and lst files using given names.
// Resets internal state in readiness for starting to scan.
// Notes whether listwanted.  Displays version information on lst file.

~SRCE();
// close src and lst files

private:
FILE *src;                // Source file
int linenumber;           // Current line number
int charpos;              // Character pointer
int minerrpos;            // Last error position
int linelength;           // Line length
char line[linemax + 1];   // Last line read
bool listing;             // true if listing required
};

#endif /*SRCE_H*/

```

```

----- srce.cpp -----

// Source handler for various parsers/compilers
// P.D. Terry, Rhodes University, 1996

#include "srce.h"

void SRCE::nextch(void)
{ if (ch == '\0') return;          // input exhausted
  if (charpos == linelength)       // new line needed
  { linelength = 0; charpos = 0; minerrpos = 0; linenumber++;
    startnewline();
    do
    { ch = getc(src);
      if (ch != '\n' && !feof(src))
      { if (listing) putc(ch, lst);
        if (linelength < linemax) { line[linelength] = ch; linelength++; }
      }
    } while (!(ch == '\n' || feof(src)));
    if (listing) putc('\n', lst);
    if (feof(src))
      line[linelength] = '\0';    // mark end with a nul character
    else
      line[linelength] = ' ';    // mark end with an explicit space
    linelength++;
  }
  ch = line[charpos]; charpos++; // pass back unique character
}

// reporterror has been coded like this (rather than use a static array of
// strings initialized by the array declarator) to allow for easy extension
// in exercises

void SRCE::reporterror(int errorcode)
{ if (charpos > minerrpos) // suppress cascading messages
  { startnewline(); fprintf(lst, "%*c", charpos, '^');
    switch (errorcode)
    { case 1:  fprintf(lst, "Incomplete string\n"); break;
      case 2:  fprintf(lst, "; expected\n"); break;
      case 3:  fprintf(lst, "Invalid start to block\n"); break;
    }
  }
}

```

```

case 4:  fprintf(lst, "Invalid declaration sequence\n"); break;
case 5:  fprintf(lst, "Invalid procedure header\n"); break;
case 6:  fprintf(lst, "Identifier expected\n"); break;
case 7:  fprintf(lst, ":= in wrong context\n"); break;
case 8:  fprintf(lst, "Number expected\n"); break;
case 9:  fprintf(lst, "= expected\n"); break;
case 10: fprintf(lst, "]" expected\n"); break;
case 13: fprintf(lst, ", or ) expected\n"); break;
case 14: fprintf(lst, "Invalid factor\n"); break;
case 15: fprintf(lst, "Invalid start to statement\n"); break;
case 17: fprintf(lst, ") expected\n"); break;
case 18: fprintf(lst, "( expected\n"); break;
case 19: fprintf(lst, "Relational operator expected\n"); break;
case 20: fprintf(lst, "Operator expected\n"); break;
case 21: fprintf(lst, ":= expected\n"); break;
case 23: fprintf(lst, "THEN expected\n"); break;
case 24: fprintf(lst, "END expected\n"); break;
case 25: fprintf(lst, "DO expected\n"); break;
case 31: fprintf(lst, ", or ; expected\n"); break;
case 32: fprintf(lst, "Invalid symbol after a statement\n"); break;
case 34: fprintf(lst, "BEGIN expected\n"); break;
case 35: fprintf(lst, "Invalid symbol after block\n"); break;
case 36: fprintf(lst, "PROGRAM expected\n"); break;
case 37: fprintf(lst, ". expected\n"); break;
case 38: fprintf(lst, "COEND expected\n"); break;
case 200: fprintf(lst, "Constant out of range\n"); break;
case 201: fprintf(lst, "Identifier redeclared\n"); break;
case 202: fprintf(lst, "Undeclared identifier\n"); break;
case 203: fprintf(lst, "Unexpected parameters\n"); break;
case 204: fprintf(lst, "Unexpected subscript\n"); break;
case 205: fprintf(lst, "Subscript required\n"); break;
case 206: fprintf(lst, "Invalid class of identifier\n"); break;
case 207: fprintf(lst, "Variable expected\n"); break;
case 208: fprintf(lst, "Too many formal parameters\n"); break;
case 209: fprintf(lst, "Wrong number of parameters\n"); break;
case 210: fprintf(lst, "Invalid assignment\n"); break;
case 211: fprintf(lst, "Cannot read this type of variable\n"); break;
case 212: fprintf(lst, "Program too long\n"); break;
case 213: fprintf(lst, "Too deeply nested\n"); break;
case 214: fprintf(lst, "Invalid parameter\n"); break;
case 215: fprintf(lst, "COBEGIN only allowed in main program\n"); break;
case 216: fprintf(lst, "Too many concurrent processes\n"); break;
case 217: fprintf(lst, "Only global procedure calls allowed here\n"); break;
case 218: fprintf(lst, "Type mismatch\n"); break;
case 219: fprintf(lst, "Unexpected expression\n"); break;
case 220: fprintf(lst, "Missing expression\n"); break;
case 221: fprintf(lst, "Boolean expression required\n"); break;
case 222: fprintf(lst, "Invalid expression\n"); break;
case 223: fprintf(lst, "Index out of range\n"); break;
case 224: fprintf(lst, "Division by zero\n"); break;
default:
    fprintf(lst, "Compiler error\n"); printf("Compiler error\n");
    if (lst != stdout) fclose(lst); exit(1);
}
}
minerrpos = charpos + 1;
}

```

```

SRCE::~SRCE()
{ if (src != NULL) { fclose(src); src = NULL; }
  if (lst != stdout) { fclose(lst); lst = NULL; }
}

SRCE::SRCE(char *sourcename, char *listname, char *version, bool listwanted)
{ src = fopen(sourcename, "r");
  if (src == NULL) { printf("Could not open input file\n"); exit(1); }
  lst = fopen(listname, "w");
  if (lst == NULL) { printf("Could not open listing file\n"); lst = stdout; }
  listing = listwanted;
  if (listing) fprintf(lst, "%s\n", version);
  charpos = 0; linelength = 0; linenumbers = 0; ch = ' ';
}

```

----- report.h -----

```

// Handle reporting of errors when parsing or compiling Clang programs
// P.D. Terry, Rhodes University, 1996

```

```

#ifndef REPORT_H
#define REPORT_H

#include "misc.h"

```

```

class REPORT {
public:
    REPORT () { errors = false; }
    // Initialize error reporter

    virtual void error(int errorcode);
    // Reports on error designated by suitable errorcode number

    bool anyerrors(void) { return errors; }
    // Returns true if any errors have been reported

protected:
    bool errors;
};

#endif /*REPORT_H*/

----- report.cpp -----

// Handle reporting of errors when parsing or compiling Clang programs
// P.D. Terry, Rhodes University, 1996

#include "report.h"

void REPORT::error(int errorcode)
{ printf("Error %d\n", errorcode); errors = true; exit(1); }

----- scan.h -----

// Lexical analyzer for Clang parsers/compilers
// P.D. Terry, Rhodes University, 1996

#ifndef SCAN_H
#define SCAN_H

#include "misc.h"
#include "report.h"
#include "srce.h"

const int lexlenght = 128;
typedef char lexeme[lexlenght + 1];

const int alfaelenght = 10;
typedef char alfa[alfaelenght + 1];

enum SCAN_symtypes {
    SCAN_unknown, SCAN_becomes, SCAN_lbracket, SCAN_times, SCAN_slash, SCAN_plus,
    SCAN_minus, SCAN_eqlsym, SCAN_neqsym, SCAN_lsssym, SCAN_leqsym, SCAN_gtrsym,
    SCAN_geqsym, SCAN_thensym, SCAN_dosym, SCAN_rbracket, SCAN_rparen, SCAN_comma,
    SCAN_lparen, SCAN_number, SCAN_stringsym, SCAN_identifier, SCAN_coendsym,
    SCAN_endsym, SCAN_ifsym, SCAN_whilesym, SCAN_stacksym, SCAN_readsymb,
    SCAN_writesym, SCAN_returnsym, SCAN_cobegsym, SCAN_waitsym, SCAN_signalsym,
    SCAN_semicolon, SCAN_beginsym, SCAN_constsym, SCAN_varsym, SCAN_procsym,
    SCAN_funcsym, SCAN_period, SCAN_progsym, SCAN_eofsym
};

struct SCAN_symbols {
    SCAN_symtypes sym; // symbol type
    int num; // value
    lexeme name; // lexeme
};

class SCAN {
public:
    void getsym(SCAN_symbols &SYM);
    // Obtains the next symbol in the source text

    SCAN(SRCE *S, REPORT *R);
    // Initializes scanner

protected:
    REPORT *Report; // Associated error reporter
    SRCE *Srce; // Associated source handler
    static struct keyword {
        alfa resword;
        SCAN_symtypes ressym;
    } table[]; // Look up table words/symbols
    int keywords; // Actual number of them
    SCAN_symtypes singlesym[256]; // One character symbols
};

```

```
#endif /*SCAN_H*/
```

```
----- scan.cpp -----
```

```
// Lexical analyzer for Clang parsers/compiler  
// P.D. Terry, Rhodes University, 1996
```

```
#include "scan.h"
```

```
SCAN::keyword SCAN::table[] = { // this must be in alphabetic order  
{ "BEGIN", SCAN_beginsym },  
{ "COBEGIN", SCAN_cobegsym },  
{ "COEND", SCAN_coendsym },  
{ "CONST", SCAN_constsym },  
{ "DO", SCAN_dosym },  
{ "END", SCAN_endsym },  
{ "FUNCTION", SCAN_funcsym },  
{ "IF", SCAN_ifsym },  
{ "PROCEDURE", SCAN_procsym },  
{ "PROGRAM", SCAN_progsym },  
{ "READ", SCAN_readsym },  
{ "RETURN", SCAN_returnsym },  
{ "SIGNAL", SCAN_signalsym },  
{ "STACKDUMP", SCAN_stacksym },  
{ "THEN", SCAN_thensym },  
{ "VAR", SCAN_varsym },  
{ "WAIT", SCAN_waitsym },  
{ "WHILE", SCAN_whilesym },  
{ "WRITE", SCAN_writesym },  
};
```

```
void SCAN::getsym(SCAN_symbols &SYM)  
{ int length; // index into SYM.Name  
  int digit; // value of digit character  
  int l, r, look; // for binary search  
  bool overflow; // in numeric conversion  
  bool endstring; // in string analysis  
  
  while (Srce->ch == ' ') Srce->nextch(); // Ignore spaces between tokens  
  SYM.name[0] = Srce->ch; SYM.name[1] = '\0'; SYM.num = 0; length = 0;  
  SYM.sym = singlesym[Srce->ch]; // Initial assumption  
  if (isalpha(Srce->ch)) // Identifier or reserved word  
  { while (isalnum(Srce->ch))  
    { if (length < lexlength) { SYM.name[length] = toupper(Srce->ch); length++; }  
      Srce->nextch();  
    }  
    SYM.name[length] = '\0'; // Terminate string properly  
    l = 0; r = keywords - 1;  
    do // Binary search  
    { look = (l + r) / 2;  
      if (strcmp(SYM.name, table[look].resword) <= 0) r = look - 1;  
      if (strcmp(SYM.name, table[look].resword) >= 0) l = look + 1;  
    } while (l <= r);  
    if (l - 1 > r)  
      SYM.sym = table[look].ressym;  
    else  
      SYM.sym = SCAN_identifier;  
  }  
  else if (isdigit(Srce->ch)) // Numeric literal  
  { SYM.sym = SCAN_number;  
    overflow = false;  
    while (isdigit(Srce->ch))  
    { digit = Srce->ch - '0';  
      // Check imminent overflow  
      if (SYM.num <= (maxint - digit) / 10)  
        SYM.num = SYM.num * 10 + digit;  
      else  
      { overflow = true;  
        if (length < lexlength) { SYM.name[length] = toupper(Srce->ch); length++; }  
        Srce->nextch();  
      }  
    }  
    if (overflow) Report->error(200);  
    SYM.name[length] = '\0'; // Terminate string properly  
  }  
  else switch (Srce->ch)  
  { case '::':  
    { Srce->nextch();  
      if (Srce->ch == '=')  
      { SYM.sym = SCAN_becomes; strcpy(SYM.name, "::="); Srce->nextch(); }  
      // else SYM.sym := SCAN_unknown; SYM.name := ":"  
      break;  
    }  
  }
```

```

case '<':
    Srce->nextch();
    if (Srce->ch == '=')
        { SYM.sym = SCAN_leqsym; strcpy(SYM.name, "<="); Srce->nextch(); }
    else if (Srce->ch == '>')
        { SYM.sym = SCAN_neqsym; strcpy(SYM.name, "<>"); Srce->nextch(); }
    // else SYM.sym := SCAN_lsssym; SYM.name := "<"
    break;

case '>':
    Srce->nextch();
    if (Srce->ch == '=')
        { SYM.sym = SCAN_geqsym; strcpy(SYM.name, ">="); Srce->nextch(); }
    // else SYM.sym := SCAN_gtrsym; SYM.name := ">"
    break;

case '\\': // String literal
    Srce->nextch();
    SYM.sym = SCAN_stringsym;
    endstring = false;
    do
    { if (Srce->ch == '\\')
        { Srce->nextch(); endstring = (Srce->ch != '\\'); }
        if (!endstring)
        { if (length < lexlength) { SYM.name[length] = Srce->ch; length++; }
            Srce->nextch();
        }
    } while (!(endstring || Srce->endline()));
    if (!endstring) Report->error(1);
    SYM.name[length] = '\\0'; // Terminate string properly
    break;

case '\\0':
    SYM.sym = SCAN_eofsym; break;

default:
    // implementation defined symbols - SYM.sym := singlesym[Srce->ch]
    Srce->nextch(); break;
}
}

SCAN::SCAN(SRCE *S, REPORT *R)
{ Srce = S; Report = R;
  // Define one char symbols
  for (int i = 0; i <= 255; i++) singlesym[i] = SCAN_unknown;
  singlesym['+'] = SCAN_plus;      singlesym['-'] = SCAN_minus;
  singlesym['*'] = SCAN_times;     singlesym['/'] = SCAN_slash;
  singlesym['('] = SCAN_lparen;    singlesym[')'] = SCAN_rparen;
  singlesym['['] = SCAN_lbracket;  singlesym[']'] = SCAN_rbracket;
  singlesym['='] = SCAN_eqsym;     singlesym[';'] = SCAN_semicolon;
  singlesym[','] = SCAN_comma;     singlesym['.'] = SCAN_period;
  singlesym['<'] = SCAN_lsssym;    singlesym['>'] = SCAN_gtrsym;
  keywords = sizeof(table) / sizeof(keyword);
  Srce->nextch();
}

```

----- parser.h -----

```

// Parser for Clang source
// P.D. Terry, Rhodes University, 1996

#ifndef PARSER_H
#define PARSER_H

#include "scan.h"
#include "report.h"
#include "table.h"
#include "srce.h"
#include "set.h"
#include "cgen.h"

typedef Set<SCAN_eofsym> symset;

class PARSER {
public:
    PARSER(CGEN *C, SCAN *L, TABLE *T, SRCE *S, REPORT *R);
    // Initializes parser

    void parse(void);
    // Parses and compiles the source code

```



```

private:
    symset RelOpSyms, FirstDeclaration, FirstBlock, FirstFactor,
        FirstExpression, FirstStatement, EmptySet;
    SCAN_symbols SYM;
    REPORT *Report;
    SCAN *Scanner;
    TABLE *Table;
    SRCE *Srce;
    CGEN *CGen;
    bool debugging;
    int blocklevel;
    TABLE_idclasses blockclass;
    void GetSym(void);
    void accept(SCAN_symtypes expected, int errorcode);
    void test(symset allowed, symset beacons, int errorcode);
    CGEN_operators op(SCAN_symtypes s);
    void OneConst(void);
    void ConstDeclarations(void);
    void OneVar(int &framesize);
    void VarDeclarations(int &framesize);
    void OneFormal(TABLE_entries &procentry, TABLE_index &parindex);
    void FormalParameters(TABLE_entries &procentry);
    void ProcDeclaration(symset followers);
    void Designator(TABLE_entries entry, symset followers, int errorcode);
    void ReferenceParameter(void);
    void OneActual(symset followers, TABLE_entries procentry, int &actual);
    void ActualParameters(TABLE_entries procentry, symset followers);
    void Variable(symset followers, int errorcode);
    void Factor(symset followers);
    void Term(symset followers);
    void Expression(symset followers);
    void Condition(symset followers);
    void CompoundStatement(symset followers);
    void Assignment(symset followers, TABLE_entries entry);
    void ProcedureCall(symset followers, TABLE_entries entry);
    void IfStatement(symset followers);
    void WhileStatement(symset followers);
    void ReturnStatement(symset followers);
    void WriteElement(symset followers);
    void WriteStatement(symset followers);
    void ReadStatement(symset followers);
    void ProcessCall(symset followers, int &processes);
    void CobeginStatement(symset followers);
    void SemaphoreStatement(symset followers);
    void Statement(symset followers);
    void Block(symset followers, int blklevel, TABLE_idclasses blkclass,
        int initialframesize);
    void ClangProgram(void);
};

#endif /*PARSER_H*/

```

----- parser.cpp -----

```

// Parser for Clang level 4 - incorporates error recovery and code generation
// Includes procedures, functions, parameters, arrays, concurrency.
// Display machine.
// P.D. Terry, Rhodes University, 1996

```

```

#include "misc.h"
#include "parser.h"
#include "report.h"
#include "table.h"
#include "scan.h"
#include "srce.h"
#include "set.h"

```

```

static int relopsyms [] = {SCAN_eqlsym, SCAN_neqsym, SCAN_gtrsym,
    SCAN_geqsym, SCAN_lsssym, SCAN_leqsym};

static int firstDeclaration [] = {SCAN_constsym, SCAN_varsym, SCAN_procsym,
    SCAN_funcsym};

static int firstBlock [] = {SCAN_constsym, SCAN_varsym, SCAN_procsym,
    SCAN_funcsym, SCAN_beginsym};

static int firstFactor [] = {SCAN_identifier, SCAN_number, SCAN_lparen,
    SCAN_stringsym};

static int firstExpression [] = {SCAN_identifier, SCAN_number, SCAN_lparen,
    SCAN_plus, SCAN_minus};

```

```

static int firstStatement [] = {SCAN_identifier, SCAN_beginsym, SCAN_ifsym,
                                SCAN_whilesym, SCAN_returnsym,
                                SCAN_writesym, SCAN_readsym, SCAN_stacksym,
                                SCAN_cobegsym, SCAN_waitsym, SCAN_signalsym};

PARSER::PARSER(CGEN *C, SCAN *L, TABLE *T, SRCE *S, REPORT *R) :
    RelOpSyms(sizeof(relopsyms)/sizeof(int), relopsyms),
    FirstDeclaration(sizeof(firstDeclaration)/sizeof(int), firstDeclaration),
    FirstBlock(sizeof(firstBlock)/sizeof(int), firstBlock),
    FirstFactor(sizeof(firstFactor)/sizeof(int), firstFactor),
    FirstExpression(sizeof(firstExpression)/sizeof(int), firstExpression),
    FirstStatement(sizeof(firstStatement)/sizeof(int), firstStatement),
    EmptySet()
{ CGen = C, Scanner = L; Report = R; Table = T; Srce = S; }

void PARSER::GetSym(void)
{ Scanner->getsym(SYM); }

void PARSER::accept(SCAN_symtypes expected, int errorcode)
{ if (SYM.sym == expected) GetSym(); else Report->error(errorcode); }

void PARSER::test(symset allowed, symset beacons, int errorcode)
// Test whether current Sym is Allowed, and recover if not
{ if (allowed.memb(SYM.sym)) return;
  Report->error(errorcode);
  symset stopset = allowed + beacons;
  while (!stopset.memb(SYM.sym)) GetSym();
}

CGEN_operators PARSER::op(SCAN_symtypes s)
// Map symbol to corresponding code operator
{ switch (s)
  { case SCAN_plus:   return CGEN_opadd;
    case SCAN_minus:  return CGEN_opsub;
    case SCAN_times:  return CGEN_opmul;
    case SCAN_slash:  return CGEN_opdvd;
    case SCAN_eqlsym: return CGEN_opeql;
    case SCAN_neqsym: return CGEN_opneq;
    case SCAN_gtrsym: return CGEN_opgtr;
    case SCAN_geqsym: return CGEN_opgeq;
    case SCAN_lsssym: return CGEN_oplss;
    case SCAN_leqsym: return CGEN_opleq;
  }
}

// ++++++ Declaration Part ++++++

void PARSER::OneConst(void)
// OneConst = ConstIdentifier "=" Number ";" .
{ TABLE_entries constentry;
  TABLE_index constindex;
  if (SYM.sym != SCAN_identifier) { Report->error(6); return; }
  sprintf(constentry.name, "%.*s", TABLE_alfalength, SYM.name);
  constentry.idclass = TABLE_consts;
  GetSym();
  if (SYM.sym == SCAN_becomes || SYM.sym == SCAN_eqlsym)
  { if (SYM.sym == SCAN_becomes) Report->error(7);
    GetSym();
    if (SYM.sym != SCAN_number)
    { constentry.c.value = 0; Report->error(8); }
    else
    { constentry.c.value = SYM.num; GetSym(); }
  }
  else Report->error(9);
  Table->enter(constentry, constindex);
  accept(SCAN_semicolon, 2);
}

void PARSER::ConstDeclarations(void)
// ConstDeclarations = "CONST" OneConst { OneConst } .
{ GetSym();
  OneConst();
  while (SYM.sym == SCAN_identifier) OneConst();
}

void PARSER::OneVar(int &framesize)
// OneVar = VarIdentifier [ UpperBound ] .
// UpperBound = "[" Number "]" .
{ TABLE_entries vareentry;
  TABLE_index varindex;
  if (SYM.sym != SCAN_identifier) { Report->error(6); return; }
  vareentry.idclass = TABLE_vars;
  vareentry.v.size = 1;
}

```

```

varentry.v.scalar = true;
varentry.v.ref = false;
varentry.v.offset = framesize + 1;
sprintf(varentry.name, "%.*s", TABLE_alfalength, SYM.name);
GetSym();
if (SYM.sym == SCAN_lbracket)
{ // array declaration
  GetSym();
  varentry.v.scalar = false;
  if (SYM.sym == SCAN_identifier || SYM.sym == SCAN_number)
  { if (SYM.sym == SCAN_identifier)
    Report->error(8);
    else
      varentry.v.size = SYM.num + 1;
    GetSym();
  }
  else
    Report->error(8);
  accept(SCAN_rbracket, 10);
}
Table->enter(varentry, varindex);
framesize += varentry.v.size;
}

void PARSE::VarDeclarations(int &framesize)
// VarDeclarations = "VAR" OneVar { "," OneVar } ";" .
{ GetSym();
  OneVar(framesize);
  while (SYM.sym == SCAN_comma || SYM.sym == SCAN_identifier)
  { accept(SCAN_comma, 31); OneVar(framesize); }
  accept(SCAN_semicolon, 2);
}

void PARSE::OneFormal(TABLE_entries &procentry, TABLE_index &parindex)
// OneFormal := ParIdentifier [ "[" "]" ] .
{ TABLE_entries parentry;
  if (SYM.sym != SCAN_identifier) { Report->error(6); return; }
  parentry.idclass = TABLE_vars;
  parentry.v.size = 1;
  parentry.v.scalar = true;
  parentry.v.ref = false;
  parentry.v.offset = procentry.p.paramsize + CGEN_headersize + 1;
  sprintf(parentry.name, "%.*s", TABLE_alfalength, SYM.name);
  GetSym();
  if (SYM.sym == SCAN_lbracket)
  { parentry.v.size = 2;
    parentry.v.scalar = false;
    parentry.v.ref = true;
    GetSym();
    accept(SCAN_rbracket, 10);
  }
  Table->enter(parentry, parindex);
  procentry.p.paramsize += parentry.v.size;
  procentry.p.params++;
}

void PARSE::FormalParameters(TABLE_entries &procentry)
// FormalParameters = "(" OneFormal { "," OneFormal } ")" .
{ TABLE_index p;
  GetSym();
  OneFormal(procentry, procentry.p.firstparam);
  while (SYM.sym == SCAN_comma || SYM.sym == SCAN_identifier)
  { accept(SCAN_comma, 13); OneFormal(procentry, p); }
  accept(SCAN_rparen, 17);
}

void PARSE::ProcDeclaration(symset followers)
// ProcDeclaration = ( "PROCEDURE" ProcIdentifier | "FUNCTION" FuncIdentifier )
// [ FormalParameters ] ";"
// Block ";" .
{ TABLE_entries procentry;
  TABLE_index procindex;
  if (SYM.sym == SCAN_funcsym)
    procentry.idclass = TABLE_funcs;
  else
    procentry.idclass = TABLE_procs;
  GetSym();
  if (SYM.sym != SCAN_identifier)
  { Report->error(6); *procentry.name = '\0'; }
  else
  { sprintf(procentry.name, "%.*s", TABLE_alfalength, SYM.name);
    GetSym();
  }
}

```

```

procentry.p.params = 0;
procentry.p.paramsize = 0;
procentry.p.firstparam = NULL;
CGen->storelabel(procentry.p.entrypoint);
Table->enter(procentry, procindex);
Table->openscope();
if (SYM.sym == SCAN_lparen)
{ FormalParameters(procentry);
  Table->update(procentry, procindex);
}
test(symset(SCAN_semicolon), followers, 5);
accept(SCAN_semicolon, 2);
Block(symset(SCAN_semicolon) + followers, procentry.level + 1,
      procentry.idclass, procentry.p.paramsize + CGEN_headersize);
accept(SCAN_semicolon, 2);
}

// +++++ Expressions and Designators+++++

void PARSE::Designator(TABLE_entries entry, symset followers, int errorcode)
// Designator = VarIdentifier [ "[" Expression "]" ] .
{ bool isVariable = (entry.idclass == TABLE_vars);
  if (isVariable)
    CGen->stackaddress(entry.level, entry.v.offset, entry.v.ref);
  else
    Report->error(errorcode);
  GetSym();
  if (SYM.sym == SCAN_lbracket)
  { // subscript
    if (isVariable && entry.v.scalar) Report->error(204);
    GetSym();
    Expression(symset(SCAN_rbracket) + followers);
    if (isVariable)
    { if (entry.v.ref)
      { CGen->stackaddress(entry.level, entry.v.offset + 1, entry.v.ref);
        else
          CGen->stackconstant(entry.v.size);
        CGen->subscript();
      }
    }
    accept(SCAN_rbracket, 10);
  }
  else if (isVariable && !entry.v.scalar) Report->error(205);
}

void PARSE::Variable(symset followers, int errorcode)
// Variable = VarDesignator .
// VarDesignator = Designator .
{ TABLE_entries entry;
  bool found;
  if (SYM.sym != SCAN_identifier) { Report->error(6); return; }
  Table->search(SYM.name, entry, found);
  if (!found) Report->error(202);
  Designator(entry, followers, errorcode);
}

void PARSE::ReferenceParameter(void)
{ TABLE_entries entry;
  bool found;
  if (SYM.sym != SCAN_identifier)
    Report->error(214);
  else
  { Table->search(SYM.name, entry, found);
    if (!found)
      Report->error(202);
    else if (entry.idclass != TABLE_vars || entry.v.scalar)
      Report->error(214);
    else
    { CGen->stackaddress(entry.level, entry.v.offset, entry.v.ref);
      // now pass size of array as next parameter
      if (entry.v.ref)
        CGen->stackaddress(entry.level, entry.v.offset + 1, entry.v.ref);
      else
        CGen->stackconstant(entry.v.size);
    }
    GetSym();
  }
}

void PARSE::OneActual(symset followers, TABLE_entries procentry, int &actual)
{ actual++;
  if (Table->isrefparam(procentry, actual))
    ReferenceParameter();
  else

```

```

    Expression(symset(SCAN_comma, SCAN_rparen) + followers);
test(symset(SCAN_comma, SCAN_rparen), followers - symset(SCAN_identifier), 13);
}

void PARSER::ActualParameters(TABLE_entries procentry, symset followers)
// ActualParameters = [ "(" Expression { "," Expression } ")" ] .
{
    int actual = 0;
    if (SYM.sym == SCAN_lparen)
    {
        GetSym();
        OneActual(followers, procentry, actual);
        while ((SYM.sym == SCAN_comma) || FirstExpression.memb(SYM.sym))
        {
            accept(SCAN_comma, 13); OneActual(followers, procentry, actual);
        }
        accept(SCAN_rparen, 17);
    }
    if (actual != procentry.p.params) Report->error(209);
}

void PARSER::Factor(symset followers)
// Factor = ValDesignator | ConstIdentifier | Number
//           | FuncIdentifier ActualParameters | "(" Expression ")" .
// ValDesignator = Designator .
{
    TABLE_entries entry;
    bool found;
    test(FirstFactor, followers, 14);
    switch (SYM.sym)
    {
        case SCAN_identifier:
            Table->search(SYM.name, entry, found);
            if (!found) Report->error(202);
            switch (entry.idclass)
            {
                case TABLE_consts:
                    CGen->stackconstant(entry.c.value); GetSym(); break;
                case TABLE_funcs:
                    GetSym();
                    CGen->markstack();
                    ActualParameters(entry, followers);
                    CGen->call(entry.level, entry.p.entrypoint);
                    break;
                default:
                    Designator(entry, followers, 206); CGen->dereference(); break;
            }
            break;

        case SCAN_number:
            CGen->stackconstant(SYM.num);
            GetSym();
            break;

        case SCAN_lparen:
            GetSym();
            Expression(symset(SCAN_rparen) + followers);
            accept(SCAN_rparen, 17);
            break;

        case SCAN_stringsym:
            Report->error(14);
            GetSym();
            break;

        default:
            Report->error(14);
            break;
    }
}

void PARSER::Term(symset followers)
// Term = Factor { ( "*" | "/" ) Factor } .
{
    SCAN_syntypes opsym;
    Factor(symset(SCAN_times, SCAN_slash) + followers);
    while (SYM.sym == SCAN_times || SYM.sym == SCAN_slash || FirstFactor.memb(SYM.sym))
    {
        if (SYM.sym == SCAN_times || SYM.sym == SCAN_slash)
        {
            opsym = SYM.sym; GetSym();
        }
        else
        {
            opsym = SCAN_times; Report->error(20);
        }
        Factor(symset(SCAN_times, SCAN_slash) + followers);
        CGen->binaryintegerop(op(opsym));
    }
}

void PARSER::Expression(symset followers)
// Expression = [ "+" | "-" ] Term { ( "+" | "-" ) Term } .
{
    SCAN_syntypes opsym;
    if (SYM.sym == SCAN_plus)
    {
        GetSym();
    }
}

```

```

    Term(symset(SCAN_plus, SCAN_minus) + followers);
}
else if (SYM.sym == SCAN_minus)
{
    GetSym();
    Term(symset(SCAN_plus, SCAN_minus) + followers);
    CGen->negateinteger();
}
else
    Term(symset(SCAN_plus, SCAN_minus) + followers);
while (SYM.sym == SCAN_plus || SYM.sym == SCAN_minus)
{
    opsym = SYM.sym; GetSym();
    Term(symset(SCAN_plus, SCAN_minus) + followers);
    CGen->binaryintegerop(op(opsym));
}
}

void PARSER::Condition(symset followers)
// Condition = Expression Relop Expression .
{
    SCAN_symtypes opsym;
    symset stopset = RelOpSyms + followers;
    Expression(stopset);
    if (!RelOpSyms.memb(SYM.sym)) { Report->error(19); return; }
    opsym = SYM.sym; GetSym();
    Expression(followers); CGen->comparison(op(opsym));
}

// ++++++ Statement Part ++++++

void PARSER::CompoundStatement(symset followers)
// CompoundStatement = "BEGIN" Statement { ";" Statement } "END" .
{
    accept(SCAN_beginsym, 34);
    Statement(symset(SCAN_semicolon, SCAN_endsym) + followers);
    while (SYM.sym == SCAN_semicolon || FirstStatement.memb(SYM.sym))
    {
        accept(SCAN_semicolon, 2);
        Statement(symset(SCAN_semicolon, SCAN_endsym) + followers);
    }
    accept(SCAN_endsym, 24);
}

void PARSER::Assignment(symset followers, TABLE_entries entry)
// Assignment = VarDesignator ":"=" Expression .
// VarDesignator = Designator .
{
    Designator(entry, symset(SCAN_becomes, SCAN_eqlsym) + followers, 210);
    if (SYM.sym == SCAN_becomes)
        GetSym();
    else
    {
        Report->error(21); if (SYM.sym == SCAN_eqlsym) GetSym(); }
    Expression(followers);
    CGen->assign();
}

void PARSER::ProcedureCall(symset followers, TABLE_entries entry)
// ProcedureCall = ProcIdentifier ActualParameters .
{
    GetSym();
    CGen->markstack();
    ActualParameters(entry, followers);
    CGen->call(entry.level, entry.p.entrypoint);
}

void PARSER::IfStatement(symset followers)
// IfStatement = "IF" Condition "THEN" Statement .
{
    CGEN_labels testlabel;

    GetSym();
    Condition(symset(SCAN_thensym, SCAN_dosym) + followers);
    CGen->jumponfalse(testlabel, CGen->undefined);
    if (SYM.sym == SCAN_thensym)
        GetSym();
    else
    {
        Report->error(23); if (SYM.sym == SCAN_dosym) GetSym(); }
    Statement(followers);
    CGen->backpatch(testlabel);
}

void PARSER::WhileStatement(symset followers)
// WhileStatement = "WHILE" Condition "DO" Statement .
{
    CGEN_labels startloop, testlabel, dummylabel;
    GetSym();
    CGen->storelabel(startloop);
    Condition(symset(SCAN_dosym) + followers);
    CGen->jumponfalse(testlabel, CGen->undefined);
    accept(SCAN_dosym, 25);
    Statement(followers);
}

```

```

    CGen->jump(dummylabel, startloop);
    CGen->backpatch(testlabel);
}

void PARSER::ReturnStatement(symset followers)
// ReturnStatement = "RETURN" [ Expression ]
{
    GetSym();
    switch (blockclass)
    {
        case TABLE_funcs:
            if (!FirstExpression.memb(SYM.sym)) Report->error(220); // an Expression is mandato
            else
            {
                CGen->stackaddress(blocklevel, 1, false);
                Expression(followers);
                CGen->assign();
                CGen->leavefunction(blocklevel);
            }
            break;

        case TABLE_procs: // simple return
            CGen->leaveprocedure(blocklevel);
            if (FirstExpression.memb(SYM.sym)) // we may NOT have an Expression
            {
                Report->error(219); Expression(followers);
            }
            break;

        case TABLE_progs: // okay in main program - just halts
            CGen->leaveprogram();
            if (FirstExpression.memb(SYM.sym)) // we may NOT have an Expression
            {
                Report->error(219); Expression(followers);
            }
            break;
    }
}

void PARSER::WriteElement(symset followers)
// WriteElement = Expression | String .
{
    CGEN_labels startstring;
    if (SYM.sym != SCAN_stringsym)
    {
        Expression(symset(SCAN_comma, SCAN_rparen) + followers);
        CGen->writevalue();
    }
    else
    {
        CGen->stackstring(SYM.name, startstring);
        CGen->writestring(startstring);
        GetSym();
    }
}

void PARSER::WriteStatement(symset followers)
// WriteStatement = "WRITE" ["(" WriteElement { "," WriteElement } ")" ] .
{
    GetSym();
    if (SYM.sym == SCAN_lparen)
    {
        GetSym();
        WriteElement(followers);
        while (SYM.sym == SCAN_comma || FirstExpression.memb(SYM.sym))
        {
            accept(SCAN_comma, 13); WriteElement(followers);
        }
        accept(SCAN_rparen, 17);
    }
    CGen->newline();
}

void PARSER::ReadStatement(symset followers)
// ReadStatement = "READ" "(" Variable { "," Variable } ")" .
{
    GetSym();
    if (SYM.sym != SCAN_lparen) { Report->error(18); return; }
    GetSym();
    Variable(symset(SCAN_comma, SCAN_rparen) + followers, 211);
    CGen->readvalue();
    while (SYM.sym == SCAN_comma || SYM.sym == SCAN_identifier)
    {
        accept(SCAN_comma, 13);
        Variable(symset(SCAN_comma, SCAN_rparen) + followers, 211);
        CGen->readvalue();
    }
    accept(SCAN_rparen, 17);
}

void PARSER::ProcessCall(symset followers, int &processes)
// ProcessCall = ProcIdentifier ActualParameters .
{
    TABLE_entries entry;
    bool found;
    if (!FirstStatement.memb(SYM.sym)) return;
    if (SYM.sym != SCAN_identifier)
    {
        Report->error(217); Statement(followers); return; // recovery
    }
    Table->search(SYM.name, entry, found);
    if (!found) Report->error(202);
}

```

```

    if (entry.idclass != TABLE_procs)
    { Report->error(217); Statement(followers); return; } // recovery
    GetSym();
    CGen->markstack();
    ActualParameters(entry, followers);
    CGen->forkprocess(entry.p.entrypoint);
    processes++;
}

void PARSER::CobeginStatement(symset followers)
// CobeginStatement := "COBEGIN" ProcessCall { ";" ProcessCall } "COEND"
// count number of processes
{ int processes = 0;
  CGEN_labels start;
  if (blockclass != TABLE_progs) Report->error(215); // only from global level
  GetSym(); CGen->cobegin(start);
  ProcessCall(symset(SCAN_semicolon, SCAN_coendsym) + followers, processes);
  while (SYM.sym == SCAN_semicolon || FirstStatement.memb(SYM.sym))
  { accept(SCAN_semicolon, 2);
    ProcessCall(symset(SCAN_semicolon, SCAN_coendsym) + followers, processes);
  }
  CGen->coend(start, processes);
  accept(SCAN_coendsym, 38);
}

void PARSER::SemaphoreStatement(symset followers)
// SemaphoreStatement = ("WAIT" | "SIGNAL") "(" VarDesignator ")" .
{ bool wait = (SYM.sym == SCAN_waitsym);
  GetSym();
  if (SYM.sym != SCAN_lparen) { Report->error(18); return; }
  GetSym();
  Variable(symset(SCAN_rparen) + followers, 206);
  if (wait) CGen->waitop(); else CGen->signalop();
  accept(SCAN_rparen, 17);
}

void PARSER::Statement(symset followers)
// Statement = [ CompoundStatement | Assignment | ProcedureCall
//               | IfStatement | WhileStatement | ReturnStatement
//               | WriteStatement | ReadStatement | CobeginStatement
//               | WaitStatement | SignalStatement | "STACKDUMP" ] .
{ TABLE_entries entry;
  bool found;
  if (FirstStatement.memb(SYM.sym))
  { switch (SYM.sym)
    { case SCAN_identifier:
      Table->search(SYM.name, entry, found);
      if (!found) Report->error(202);
      if (entry.idclass == TABLE_procs)
        ProcedureCall(followers, entry);
      else
        Assignment(followers, entry);
      break;
      case SCAN_ifsym:      IfStatement(followers); break;
      case SCAN_whilesym:  WhileStatement(followers); break;
      case SCAN_returnsym: ReturnStatement(followers); break;
      case SCAN_writesym:  WriteStatement(followers); break;
      case SCAN_readsym:   ReadStatement(followers); break;
      case SCAN_beginsym:  CompoundStatement(followers); break;
      case SCAN_stacksym:  CGen->dump(); GetSym(); break;
      case SCAN_cobegsym:  CobeginStatement(followers); break;
      case SCAN_waitsym:
      case SCAN_signalsym: SemaphoreStatement(followers); break;
    }
  }
  // test(Followers - symset(SCAN_identifier), EmptySet, 32) or
  test(followers, EmptySet, 32);
}

void PARSER::Block(symset followers, int blklevel, TABLE_idclasses blkclass,
                  int initialframesize)
// Block = { ConstDeclarations | VarDeclarations | ProcDeclaration }
//         CompoundStatement .
{ int framesize = initialframesize; // activation record space
  CGEN_labels entrypoint;
  CGen->jump(entrypoint, CGen->undefined);
  test(FirstBlock, followers, 3);
  if (blklevel > CGEN_levmax) Report->error(213);
  do
  { if (SYM.sym == SCAN_constsym) ConstDeclarations();
    if (SYM.sym == SCAN_varsym) VarDeclarations(framesize);
    while (SYM.sym == SCAN_procsym || SYM.sym == SCAN_funcsym)
      ProcDeclaration(followers);
  }
}

```



```

struct TABLE_nodes {
    TABLE_entries entry;
    TABLE_index next;
};

struct SCOPE_nodes {
    SCOPE_nodes *down;
    TABLE_index first;
};

class TABLE {
public:
    void openscope(void);
    // Opens new scope for a new block

    void closescope(void);
    // Closes scope at block exit

    void enter(TABLE_entries &entry, TABLE_index &position);
    // Adds entry to symbol table, and returns its position

    void search(char *name, TABLE_entries &entry, bool &found);
    // Searches table for presence of name. If found then returns entry

    void update(TABLE_entries &entry, TABLE_index position);
    // Updates entry at known position

    bool isrefparam(TABLE_entries &procentry, int n);
    // Returns true if nth parameter for procentry is passed by reference

    void printtable(FILE *lst);
    // Prints symbol table for diagnostic purposes

    TABLE(REPORT *R);
    // Initializes symbol table

private:
    TABLE_index sentinel;
    SCOPE_nodes *topscope;
    REPORT *Report;
    int currentlevel;
};

#endif /*TABLE_H*/

```

```

----- table.cpp -----

// Handle symbol table for Clang level 3/4 compiler/interpreter
// Includes procedures, functions, parameters, arrays, concurrency
// P.D. Terry, Rhodes University, 1996

#include "misc.h"
#include "table.h"

void TABLE::openscope(void)
{
    SCOPE_nodes *newscope = new SCOPE_nodes;
    newscope->down = topscope; newscope->first = sentinel;
    topscope = newscope;
    currentlevel++;
}

void TABLE::closescope(void)
{
    SCOPE_nodes *old = topscope;
    topscope = topscope->down; delete old;
    currentlevel--;
}

void TABLE::enter(TABLE_entries &entry, TABLE_index &position)
{
    TABLE_index look = topscope->first;
    TABLE_index last = NULL;
    position = new TABLE_nodes;
    sprintf(sentinel->entry.name, "%.s", TABLE_alfalength, entry.name);
    while (strcmp(look->entry.name, sentinel->entry.name))
    {
        last = look; look = look->next;
    }
    if (look != sentinel) Report->error(201);
    entry.level = currentlevel;
    position->entry = entry; position->next = look;
    if (!last) topscope->first = position; else last->next = position;
}

void TABLE::update(TABLE_entries &entry, TABLE_index position)

```

```

{ position->entry = entry; }

void TABLE::search(char *name, TABLE_entries &entry, bool &found)
{ TABLE_index look;
  SCOPE_nodes *scope = toplevel;
  sprintf(sentinel->entry.name, "%.s", TABLE_alphabet, name);
  while (scope)
  { look = scope->first;
    while (strcmp(look->entry.name, sentinel->entry.name)) look = look->next;
    if (look != sentinel) { found = true; entry = look->entry; return; }
    scope = scope->down;
  }
  found = false; entry = sentinel->entry;
}

bool TABLE::isrefparam(TABLE_entries &procentry, int n)
{ if (n > procentry.p.params) return false;
  TABLE_index look = procentry.p.firstparam;
  while (n > 1) { look = look->next; n--; }
  return look->entry.v.ref;
}

void TABLE::printtable(FILE *lst)
{ SCOPE_nodes *scope = toplevel;
  TABLE_index current;
  putc('\n', lst);
  while (scope)
  { current = scope->first;
    while (current != sentinel)
    { fprintf(lst, "%-16s", current->entry.name);
      switch (current->entry.idclass)
      { case TABLE_consts:
        fprintf(lst, " Constant %7d\n", current->entry.c.value);
        break;

        case TABLE_vars:
        fprintf(lst, " Variable %3d%4d%4d\n",
          current->entry.level, current->entry.v.offset,
          current->entry.v.size);
        break;

        case TABLE_procs:
        fprintf(lst, " Procedure %3d%4d%4d\n",
          current->entry.level, current->entry.p.entrypoint,
          current->entry.p.params);
        break;

        case TABLE_funcs:
        fprintf(lst, " Function %3d%4d%4d\n",
          current->entry.level, current->entry.p.entrypoint,
          current->entry.p.params);
        break;

        case TABLE_progs:
        fprintf(lst, " Program %3d\n",
          current->entry.level);
        break;
      }
      current = current->next;
    }
    scope = scope->down;
  }
}

TABLE::TABLE(REPORT *R)
{ sentinel = new TABLE_nodes;
  sentinel->entry.name[0] = '\0'; sentinel->entry.level = 0;
  sentinel->entry.idclass = TABLE_progs;
  currentlevel = 0; toplevel = NULL; // for level 0 identifiers
  Report = R; openscope(); currentlevel = 0;
}

----- cgen.h -----

// Code Generation for Clang level 4 compiler/interpreter
// Includes procedures, functions, parameters, arrays, concurrency.
// Display machine.
// P.D. Terry, Rhodes University, 1996

#ifdef CGEN_H
#define CGEN_H

#include "misc.h"

```

```

#include "stkmc.h"
#include "report.h"

#define CGEN_headersize STKMC_headersize
#define CGEN_levmax STKMC_levmax

enum CGEN_operators {
    CGEN_opadd, CGEN_opsub, CGEN_opmul, CGEN_opdvd, CGEN_opeql, CGEN_opneq,
    CGEN_oplss, CGEN_opgeq, CGEN_opgtr, CGEN_opleq
};

typedef short CGEN_labels;
typedef char CGEN_levels;

class CGEN {
public:
    CGEN_labels undefined;    // for forward references

    CGEN(REPORT *R);
    // Initializes code generator

    void negateinteger(void);
    // Generates code to negate integer value on top of evaluation stack

    void binaryintegerop(CGEN_operators op);
    // Generates code to pop two values A,B from evaluation stack
    // and push value A op B

    void comparison(CGEN_operators op);
    // Generates code to pop two integer values A,B from stack
    // and push Boolean value A OP B

    void readvalue(void);
    // Generates code to read value; store on address found on top of stack

    void writevalue(void);
    // Generates code to output value from top of stack

    void newline(void);
    // Generates code to output line mark

    void writestring(CGEN_labels location);
    // Generates code to output string stored at known location

    void stackstring(char *str, CGEN_labels &location);
    // Stores str in literal pool in memory and return its location

    void stackconstant(int number);
    // Generates code to push number onto evaluation stack

    void stackaddress(int level, int offset, bool indirect);
    // Generates code to push address for known level, offset onto evaluation stack.
    // Addresses of reference parameters are treated as indirect

    void subscript(void);
    // Generates code to index an array and check that bounds are not exceeded

    void dereference(void);
    // Generates code to replace top of evaluation stack by the value found at the
    // address currently stored on top of the stack

    void assign(void);
    // Generates code to store value currently on top-of-stack on the address
    // given by next-to-top, popping these two elements

    void openstackframe(int size);
    // Generates code to reserve space for size variables

    void leaveprogram(void);
    // Generates code needed to leave a program (halt)

    void leaveprocedure(int blocklevel);
    // Generates code needed to leave a regular procedure at given blocklevel

    void leavefunction(int blocklevel);
    // Generates code needed as we leave a function at given blocklevel

    void functioncheck(void);
    // Generate code to ensure that a function has returned a value

    void cobegin(CGEN_labels &location);
    // Generates code to initiate concurrent processing

```

```

void coend(CGEN_labels location, int number);
// Generates code to terminate concurrent processing

void storelabel(CGEN_labels &location);
// Stores address of next instruction in location for use in backpatching

void jump(CGEN_labels &here, CGEN_labels destination);
// Generates unconditional branch from here to destination

void jumponfalse(CGEN_labels &here, CGEN_labels destination);
// Generates branch from here to destination, conditional on the Boolean
// value currently on top of the evaluation stack, popping this value

void backpatch(CGEN_labels location);
// Stores the current location counter as the address field of the branch
// instruction currently held in an incomplete form at location

void markstack(void);
// Generates code to reserve mark stack storage before calling procedure

void call(int level, CGEN_labels entrypoint);
// Generates code to enter procedure at known level and entrypoint

void forkprocess(CGEN_labels entrypoint);
// Generates code to initiate process at known entrypoint

void signalop(void);
// Generates code for semaphore signalling operation

void waitop(void);
// Generates code for semaphore wait operation

void dump(void);
// Generates code to dump the current state of the evaluation stack

void getsize(int &codelength, int &initsp);
// Returns length of generated code and initial stack pointer

int gettop(void);
// Return codetop

void emit(int word);
// Emits single word

private:
    REPORT *Report;
    bool generatingcode;
    STKMC_address codetop, stktop;
};

#endif /*CGEN_H*/

```

```

----- cgen.cpp -----

// Code Generation for Clang Level 4 compiler/interpreter
// Includes procedures, functions, parameters, arrays, concurrency.
// Display machine.
// P.D. Terry, Rhodes University, 1996

#include "cgen.h"

extern STKMC* Machine;

CGEN::CGEN(REPORT *R)
{ undefined = 0; // for forward references (exported)
  Report = R;
  generatingcode = true;
  codetop = 0;
  stktop = STKMC_memsize - 1;
}

void CGEN::emit(int word)
// Code generator for single word
{ if (!generatingcode) return;
  if (codetop >= stktop)
  { Report->error(212); generatingcode = false; }
  else
  { Machine->mem[codetop] = word; codetop++; }
}

void CGEN::negateinteger(void)
{ emit(int(STKMC_neg)); }

```

```

void CGEN::binaryintegerop(CGEN_operators op)
{ switch (op)
  { case CGEN_opmul:   emit(int(STKMC_mul)); break;
    case CGEN_opdvd:   emit(int(STKMC_dvd)); break;
    case CGEN_opadd:   emit(int(STKMC_add)); break;
    case CGEN_opsub:   emit(int(STKMC_sub)); break;
  }
}

void CGEN::comparison(CGEN_operators op)
{ switch (op)
  { case CGEN_opeql:   emit(int(STKMC_eql)); break;
    case CGEN_opneq:   emit(int(STKMC_neq)); break;
    case CGEN_oplss:   emit(int(STKMC_lss)); break;
    case CGEN_opleq:   emit(int(STKMC_leq)); break;
    case CGEN_opgtr:   emit(int(STKMC_gtr)); break;
    case CGEN_opgeq:   emit(int(STKMC_geq)); break;
  }
}

void CGEN::readvalue(void)
{ emit(int(STKMC_inn)); }

void CGEN::writevalue(void)
{ emit(int(STKMC_prn)); }

void CGEN::newline(void)
{ emit(int(STKMC_nln)); }

void CGEN::writestring(CGEN_labels location)
{ emit(int(STKMC_prs)); emit(location); }

void CGEN::stackstring(char *str, CGEN_labels &location)
{ int l = strlen(str);
  if (stktop <= codetop + 1 + 1)
    { Report->error(212); generatingcode = false; return; }
  location = stktop - 1;
  for (int i = 0; i < l; i++) { stktop--; Machine->mem[stktop] = str[i]; }
  stktop--; Machine->mem[stktop] = 0;
}

void CGEN::stackconstant(int number)
{ emit(int(STKMC_lit)); emit(number); }

void CGEN::stackaddress(int level, int offset, bool indirect)
{ emit(int(STKMC_adr)); emit(level); emit(-offset);
  if (indirect) emit(int(STKMC_val));
}

void CGEN::subscript(void)
{ emit(int(STKMC_ind)); }

void CGEN::dereference(void)
{ emit(int(STKMC_val)); }

void CGEN::assign(void)
{ emit(int(STKMC_sto)); }

void CGEN::openstackframe(int size)
{ if (size > 0) { emit(int(STKMC_dsp)); emit(size); } }

void CGEN::leaveprogram(void)
{ emit(int(STKMC_hlt)); }

void CGEN::leavefunction(int blocklevel)
{ emit(int(STKMC_ret)); emit(blocklevel); emit(1); }

void CGEN::functioncheck(void)
{ emit(int(STKMC_nfn)); }

void CGEN::leaveprocedure(int blocklevel)
{ emit(int(STKMC_ret)); emit(blocklevel); emit(0); }

void CGEN::cobegin(CGEN_labels &location)
{ location = codetop; emit(int(STKMC_cbg)); emit(undefined); }

void CGEN::coend(CGEN_labels location, int number)
{ if (number >= STKMC_procmax) Report->error(216);
  else { Machine->mem[location+1] = number; emit(int(STKMC_cnd)); }
}

void CGEN::storelabel(CGEN_labels &location)

```

```

{ location = codetop; }

void CGEN::jump(CGEN_labels &here, CGEN_labels destination)
{ here = codetop; emit(int(STKMC_brn)); emit(destination); }

void CGEN::jumponfalse(CGEN_labels &here, CGEN_labels destination)
{ here = codetop; emit(int(STKMC_bze)); emit(destination); }

void CGEN::backpatch(CGEN_labels location)
{ if (codetop == location + 2 &&
     STKMC_opcodes(Machine->mem[location]) == STKMC_brn)
    codetop -= 2;
  else
    Machine->mem[location+1] = codetop;
}

void CGEN::markstack(void)
{ emit(int(STKMC_mst)); }

void CGEN::forkprocess(CGEN_labels entrypoint)
{ emit(int(STKMC_frk)); emit(entrypoint); }

void CGEN::call(int level, CGEN_labels entrypoint)
{ emit(int(STKMC_cal)); emit(level); emit(entrypoint); }

void CGEN::signalop(void)
{ emit(int(STKMC_sig)); }

void CGEN::waitop(void)
{ emit(int(STKMC_wgt)); }

void CGEN::dump(void)
{ emit(int(STKMC_stk)); }

void CGEN::getsize(int &codelength, int &initsp)
{ codelength = codetop; initsp = stktop; }

int CGEN::gettop(void)
{ return codetop; }

----- stkmc.h -----

// Definition of simple stack machine and simple emulator for Clang level 4
// Includes procedures, functions, parameters, arrays, concurrency.
// This version emulates one CPU time sliced between processes.
// Display machine
// P.D. Terry, Rhodes University, 1996

#ifndef STKMC_H
#define STKMC_H

#define STKMC_version "Clang 4.0"
const int STKMC_memsize = 512; // Limit on memory
const int STKMC_levmax = 5; // Limit on Display
const int STKMC_headersize = 5; // Size of minimum activation record
const int STKMC_procmax = 10; // Limit on concurrent processes

// machine instructions - order important
enum STKMC_opcodes {
    STKMC_cal, STKMC_ret, STKMC_adr, STKMC_frk, STKMC_cbg, STKMC_lit, STKMC_dsp,
    STKMC_brn, STKMC_bze, STKMC_prs, STKMC_wgt, STKMC_sig, STKMC_cnd, STKMC_nfn,
    STKMC_mst, STKMC_add, STKMC_sub, STKMC_mul, STKMC_dvd, STKMC_eql, STKMC_neq,
    STKMC_lss, STKMC_geq, STKMC_gtr, STKMC_leq, STKMC_neg, STKMC_val, STKMC_sto,
    STKMC_ind, STKMC_stk, STKMC_hlt, STKMC_inn, STKMC_prn, STKMC_nln, STKMC_nop,
    STKMC_nul
};

typedef enum {
    running, finished, badmem, baddata, nodata, divzero, badop, badind,
    badfun, badsem, deadlock
} status;
typedef int STKMC_address;
typedef int STKMC_levels;
typedef int STKMC_procindex;

class STKMC {
public:
    int mem[STKMC_memsize]; // virtual machine memory

    void listcode(char *filename, STKMC_address codelen);
    // Lists the codelen instructions stored in mem on named output file

```

```

void emulator(STKMC_address initpc, STKMC_address codelen,
              STKMC_address initsp, FILE *data, FILE *results,
              bool tracing);
// Emulates action of the codelen instructions stored in mem, with
// program counter initialized to initpc, stack pointer initialized to
// initsp. data and results are used for I/O. Tracing at the code level
// may be requested

void interpret(STKMC_address codelen, STKMC_address initsp);
// Interactively opens data and results files. Then interprets the
// codelen instructions stored in mem, with stack pointer initialized
// to initsp

STKMC_opcodes opcode(char *str);
// Maps str to opcode, or to MC_nul if no match can be found

STKMC();
// Initializes stack machine

private:
struct processor {
    STKMC_opcodes ir; // Instruction register
    int bp;           // Base pointer
    int sp;           // Stack pointer
    int mp;           // Mark Stack pointer
    int pc;           // Program counter
};
struct processrec { // Process descriptors
    STKMC_address bp, mp, sp, pc; // Shadow registers
    STKMC_procindex next;         // Ring pointer
    STKMC_procindex queue;        // Linked, waiting on semaphore
    bool ready;                   // Process ready flag
    STKMC_address stackmax, stackmin; // Memory limits
    int display[STKMC_levmax];    // Display registers
};
processor cpu;
status ps;

bool inbounds(int p);

char *mnemonics[STKMC_nul+1];
void stackdump(STKMC_address initsp, FILE *results, STKMC_address pcnow);
void trace(FILE *results, STKMC_address pcnow);
void postmortem(FILE *results, STKMC_address pcnow);

int slice;
STKMC_procindex current, nexttorun;
processrec process[STKMC_procmax + 1];
void swapregisters(void);
void chooseprocess(void);
void signal(STKMC_address semaddress);
void wait(STKMC_address semaddress);
};

#endif /*STKMC_H*/

----- stkmc.cpp -----

// Definition of simple stack machine and simple emulator for Clang level 4
// Includes procedures, functions, parameters, arrays, concurrency.
// This version emulates one CPU time sliced between processes.
// Display machine.
// P.D. Terry, Rhodes University, 1996

#include "misc.h"
#include "stkmc.h"
#include <time.h>

#define random(num) (rand() % (num))
#define BLANKS " "

const int maxslice      = 8; // maximum time slice
const int processreturn = 0; // fictitious return address

STKMC_opcodes STKMC::opcode(char *str)
{ STKMC_opcodes l = STKMC_opcodes(0);
  for (int i = 0; str[i]; i++) str[i] = toupper(str[i]);
  while (l != STKMC_nul && strcmp(str, mnemonics[l]))
    l = STKMC_opcodes(long(l) + 1);
  return l;
}

```



```

void STKMC::listcode(char *filename, STKMC_address codelen)
{
    STKMC_address i, j;
    STKMC_opcodes op;
    if (*filename == '\0') return;
    FILE *codefile = fopen(filename, "w");
    if (codefile == NULL) return;
    /* The following may be useful for debugging the interpreter
    i = 0;
    while (i < codelen)
    {
        fprintf(codefile, "%4d", mem[i]);
        if ((i + 1) % 16 == 0) putc('\n', codefile);
        i++;
    }
    putc('\n', codefile);
    */
    i = 0;
    while (i < codelen)
    {
        op = STKMC_opcodes(mem[i] % (int(STKMC_nul) + 1)); // force in range
        fprintf(codefile, "%10d %s", i, mnemonics[op]);
        switch (op)
        {
            case STKMC_cal:
            case STKMC_ret:
            case STKMC_adr:
                i = (i + 1) % STKMC_memsize; fprintf(codefile, "%3d", mem[i]);
                i = (i + 1) % STKMC_memsize; fprintf(codefile, "%6d", mem[i]);
                break;

            case STKMC_frk:
            case STKMC_cbg:
            case STKMC_lit:
            case STKMC_dsp:
            case STKMC_brn:
            case STKMC_bze:
                i = (i + 1) % STKMC_memsize; fprintf(codefile, "%9d", mem[i]);
                break;

            case STKMC_prs:
                i = (i + 1) % STKMC_memsize;
                j = mem[i]; fprintf(codefile, "    ");
                while (mem[j] != 0) { putc(mem[j], codefile); j--; }
                putc('\'', codefile);
                break;
        }
        i = (i + 1) % STKMC_memsize;
        putc('\n', codefile);
    }
    fclose(codefile);
}

void STKMC::swapregisters(void)
// Save current machine registers; restore from next process
{
    process[current].bp = cpu.bp;    cpu.bp = process[nexttorun].bp;
    process[current].mp = cpu.mp;    cpu.mp = process[nexttorun].mp;
    process[current].sp = cpu.sp;    cpu.sp = process[nexttorun].sp;
    process[current].pc = cpu.pc;    cpu.pc = process[nexttorun].pc;
}

void STKMC::chooseprocess(void)
// From current process, traverse ring of descriptors to next ready process
{
    if (slice != 0) { slice--; return; }
    do { nexttorun = process[nexttorun].next; } while (!process[nexttorun].ready);
    if (nexttorun != current) swapregisters();
    slice = random(maxslice) + 3;
}

bool STKMC::inbounds(int p)
// Check that memory pointer P does not go out of bounds. This should not
// happen with correct code, but it is just as well to check
{
    if (p < process[current].stackmin || p >= STKMC_memsize) ps = badmem;
    return (ps == running);
}

void STKMC::stackdump(STKMC_address initsp, FILE *results, STKMC_address pcnow)
// Dump data area - useful for debugging
{
    int online = 0;
    fprintf(results, "\nStack dump at %4d CPU:%4d", pcnow, current);
    fprintf(results, " SP:%4d BP:%4d", cpu.sp, cpu.bp);
    fprintf(results, " SM:%4d", process[current].stackmin);
    if (cpu.bp < initsp) fprintf(results, " Return Address:%4d", mem[cpu.bp - 4]);
    putc('\n', results);
    for (int l = process[current].stackmax - 1; l >= cpu.sp; l--)
    {
        fprintf(results, "%7d:%5d", l, mem[l]);
        online++; if (online % 6 == 0) putc('\n', results);
    }
}

```

```

    }
    fprintf(results, "\nDisplay");
    for (l = 0; l < STKMC_levmax; l++)
        fprintf(results, "%4d", process[current].display[l]);
    putc('\n', results);
}

void STKMC::trace(FILE *results, STKMC_address pcnow)
// Simple trace facility for run time debugging
{
    fprintf(results, "CPU:%4d PC:%4d BP:%4d", current, pcnow, cpu.bp);
    fprintf(results, " SP:%4d TOS:", cpu.sp);
    if (cpu.sp < STKMC_memsize)
        fprintf(results, "%4d", mem[cpu.sp]);
    else
        fprintf(results, "????");
    fprintf(results, " %s", mnemonics[cpu.ir]);
    switch (cpu.ir)
    {
        case STKMC_cal:
        case STKMC_ret:
        case STKMC_adr:
            fprintf(results, "%3d%6d", mem[cpu.pc], mem[cpu.pc + 1]);
            break;
        case STKMC_frk:
        case STKMC_cbg:
        case STKMC_lit:
        case STKMC_dsp:
        case STKMC_brn:
        case STKMC_bze:
        case STKMC_prs:
            fprintf(results, "%9d", mem[cpu.pc]); break;
        // no default needed
    }
    putc('\n', results);
}

void STKMC::postmortem(FILE *results, int pcnow)
// Report run time error and position
{
    putc('\n', results);
    switch (ps)
    {
        case badop:    fprintf(results, "Illegal opcode"); break;
        case nodata:   fprintf(results, "No more data"); break;
        case baddata:  fprintf(results, "Invalid data"); break;
        case divzero:  fprintf(results, "Division by zero"); break;
        case badmem:   fprintf(results, "Memory violation"); break;
        case badind:   fprintf(results, "Subscript out of range"); break;
        case badfun:   fprintf(results, "Function did not return value"); break;
        case badsem:   fprintf(results, "Bad Semaphore operation"); break;
        case deadlock: fprintf(results, "Deadlock"); break;
    }
    fprintf(results, " at %4d in process %d\n", pcnow, current);
}

void STKMC::signal(STKMC_address semaddress)
{
    if (mem[semaddress] >= 0) // do we need to waken a process?
    {
        mem[semaddress]++; return; // no - simply increment semaphore
    }
    STKMC_procindex woken = -mem[semaddress]; // negate to find index
    mem[semaddress] = -process[woken].queue; // bump queue pointer
    process[woken].queue = 0; // remove from queue
    process[woken].ready = true; // and allow to be reactivated
}

void STKMC::wait(STKMC_address semaddress)
{
    STKMC_procindex last, now;
    if (mem[semaddress] > 0) // do we need to suspend?
    {
        mem[semaddress]--; return; // no - simply decrement semaphore
    }
    slice = 0; chooseprocess(); // choose the next process
    process[current].ready = false; // and suspend this one
    if (current == nexttorun) { ps = deadlock; return; }
    now = -mem[semaddress]; // look for end of semaphore queue
    while (now != 0) { last = now; now = process[now].queue; }
    if (mem[semaddress] == 0)
        mem[semaddress] = -current; // first in queue
    else
        process[last].queue = current; // place at end of existing queue
    process[current].queue = 0; // and mark as the new end of queue
}

void STKMC::emulator(STKMC_address initpc, STKMC_address codelen,
                     STKMC_address initsp, FILE *data, FILE *results,
                     bool tracing)
{
    STKMC_address pcnow; // Current program counter
    STKMC_address parentsp; // Original stack pointer of parent
    STKMC_procindex nprocs; // Number of concurrent processes

```

```

int partition;          // Memory allocated to each process
int loop;
srand(time(NULL));      // Initialize random number generator
process[0].stackmax = initsp;
process[0].stackmin = codelen;
process[0].queue = 0;
process[0].ready = true;
cpu.sp = initsp;
cpu.bp = initsp;        // initialize registers
cpu.pc = initpc;        // initialize program counter
for (int l = 0; l < STKMC_levmax; l++) process[0].display[l] = initsp;
nexttorun = 0;
nprocs = 0;
slice = 0;
ps = running;
do
{
    pcnow = cpu.pc; current = nexttorun;
    if (unsigned(mem[cpu.pc]) > int(STKMC_nul)) ps = badop;
    else
    {
        cpu.ir = STKMC_opcodes(mem[cpu.pc]); cpu.pc++; // fetch
        if (tracing) trace(results, pcnow);
        switch (cpu.ir) // execute
        {
            case STKMC_cal:
                mem[cpu.mp - 2] = process[current].display[mem[cpu.pc]];
                mem[cpu.mp - 3] = cpu.bp; // save display element
                mem[cpu.mp - 4] = cpu.pc + 2; // save dynamic link
                process[current].display[mem[cpu.pc]] = cpu.mp; // save return address
                // update display
                cpu.bp = cpu.mp; // reset base pointer
                cpu.pc = mem[cpu.pc + 1]; // enter procedure
                break;

            case STKMC_ret:
                process[current].display[mem[cpu.pc] - 1] = mem[cpu.bp - 2]; // restore display
                // restore display
                cpu.sp = cpu.bp - mem[cpu.pc + 1]; // discard stack frame
                cpu.mp = mem[cpu.bp - 5]; // restore mark pointer
                cpu.pc = mem[cpu.bp - 4]; // get return address
                cpu.bp = mem[cpu.bp - 3]; // reset base pointer
                if (cpu.pc == processreturn) // kill a concurrent process
                {
                    nprocs--; slice = 0; // force choice of new process
                    if (nprocs == 0) // reactivate main program
                    {
                        nexttorun = 0; swapregisters();
                    }
                    else // complete this process only
                    {
                        chooseprocess(); // may fail
                        process[current].ready = false;
                        if (current == nexttorun) ps = deadlock;
                    }
                }
                break;

            case STKMC_adr:
                cpu.sp--;
                if (inbounds(cpu.sp))
                {
                    mem[cpu.sp] = process[current].display[mem[cpu.pc] - 1]
                        + mem[cpu.pc + 1];
                    cpu.pc += 2;
                }
                break;

            case STKMC_frk:
                nprocs++;
                // first initialize the shadow CPU registers and Display
                process[nprocs].bp = cpu.mp; // base pointer
                process[nprocs].mp = cpu.mp; // mark pointer
                process[nprocs].sp = cpu.sp; // stack pointer
                process[nprocs].pc = mem[cpu.pc]; // process entry point
                process[nprocs].display[0] = process[0].display[0]; // for global access
                process[nprocs].display[1] = cpu.mp; // for local access
                // now initialize activation record
                mem[process[nprocs].bp - 2] = process[0].display[1]; // display copy
                mem[process[nprocs].bp - 3] = cpu.bp; // dynamic link
                mem[process[nprocs].bp - 4] = processreturn; // return address
                // descriptor house keeping
                process[nprocs].stackmax = cpu.mp; // memory limits
                process[nprocs].stackmin = cpu.mp - partition;
                process[nprocs].ready = true; // ready to run
                process[nprocs].queue = 0; // clear semaphore queue
                process[nprocs].next = nprocs + 1; // link to next descriptor
                cpu.sp = cpu.mp - partition; // bump parent SP below
                cpu.pc++; // reserved memory
                break;

            case STKMC_cbg:

```

```

        if (mem[cpu.pc] > 0)
        { partition = (cpu.sp - codelen) / mem[cpu.pc]; // divide rest of memory
          parentsp = cpu.sp;                          // for restoration by cnd
        }
        cpu.pc++;
        break;
case STKMC_lit:
    cpu.sp--;
    if (inbounds(cpu.sp)) { mem[cpu.sp] = mem[cpu.pc]; cpu.pc++; }
    break;
case STKMC_dsp:
    cpu.sp -= mem[cpu.pc];
    if (inbounds(cpu.sp)) cpu.pc++;
    break;
case STKMC_brn:
    cpu.pc = mem[cpu.pc]; break;
case STKMC_bze:
    cpu.sp++;
    if (inbounds(cpu.sp))
    { if (mem[cpu.sp - 1] == 0) cpu.pc = mem[cpu.pc]; else cpu.pc++; }
    break;
case STKMC_prs:
    if (tracing) fputs(BLANKS, results);
    loop = mem[cpu.pc];
    cpu.pc++;
    while (inbounds(loop) && mem[loop] != 0)
    { putc(mem[loop], results); loop--; }
    if (tracing) putc('\n', results);
    break;
case STKMC_wgt:
    if (current == 0) ps = badsem;
    else { cpu.sp++; wait(mem[cpu.sp - 1]); }
    break;
case STKMC_sig:
    if (current == 0) ps = badsem;
    else { cpu.sp++; signal(mem[cpu.sp - 1]); }
    break;
case STKMC_cnd:
    if (nprocs > 0)
    { process[nprocs].next = 1;          // close ring
      nexttorun = random(nprocs) + 1;    // choose first process at random
      cpu.sp = parentsp;                 // restore parent stack pointer
    }
    break;
case STKMC_nfn:
    ps = badfun; break;
case STKMC_mst:
    if (inbounds(cpu.sp-STKMC_headersize)) // check space available
    { mem[cpu.sp-5] = cpu.mp;              // save mark pointer
      cpu.mp = cpu.sp;                    // set mark stack pointer
      cpu.sp -= STKMC_headersize;         // bump stack pointer
    }
    break;
case STKMC_add:
    cpu.sp++;
    if (inbounds(cpu.sp)) mem[cpu.sp] += mem[cpu.sp - 1];
    break;
case STKMC_sub:
    cpu.sp++;
    if (inbounds(cpu.sp)) mem[cpu.sp] -= mem[cpu.sp - 1];
    break;
case STKMC_mul:
    cpu.sp++;
    if (inbounds(cpu.sp)) mem[cpu.sp] *= mem[cpu.sp - 1];
    break;
case STKMC_dvd:
    cpu.sp++;
    if (inbounds(cpu.sp))
    { if (mem[cpu.sp - 1] == 0)
        ps = divzero;
      else
        mem[cpu.sp] /= mem[cpu.sp - 1];
    }
    break;
case STKMC_eql:
    cpu.sp++;
    if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] == mem[cpu.sp - 1]);
    break;
case STKMC_neq:
    cpu.sp++;
    if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] != mem[cpu.sp - 1]);
    break;
case STKMC_lss:

```

```

        cpu.sp++;
        if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] < mem[cpu.sp - 1]);
        break;
    case STKMC_geq:
        cpu.sp++;
        if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] >= mem[cpu.sp - 1]);
        break;
    case STKMC_gtr:
        cpu.sp++;
        if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] > mem[cpu.sp - 1]);
        break;
    case STKMC_leq:
        cpu.sp++;
        if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] <= mem[cpu.sp - 1]);
        break;
    case STKMC_neg:
        if (inbounds(cpu.sp)) mem[cpu.sp] = -mem[cpu.sp];
        break;
    case STKMC_val:
        if (inbounds(cpu.sp) && inbounds(mem[cpu.sp]))
            mem[cpu.sp] = mem[mem[cpu.sp]];
        break;
    case STKMC_sto:
        cpu.sp++;
        if (inbounds(cpu.sp) && inbounds(mem[cpu.sp]))
            mem[mem[cpu.sp]] = mem[cpu.sp - 1];
        cpu.sp++;
        break;
    case STKMC_ind:
        if ((mem[cpu.sp + 1] < 0) || (mem[cpu.sp + 1] >= mem[cpu.sp]))
            ps = badind;
        else
        {
            cpu.sp += 2;
            if (inbounds(cpu.sp)) mem[cpu.sp] -= mem[cpu.sp - 1];
        }
        break;
    case STKMC_stk:
        stackdump(initsp, results, pcnow); break;
    case STKMC_hlt:
        ps = finished; break;
    case STKMC_inn:
        if (inbounds(cpu.sp) && inbounds(mem[cpu.sp]))
        {
            if (fscanf(data, "%d", &mem[mem[cpu.sp]]) == 0)
                ps = baddata;
            else
                cpu.sp++;
        }
        break;
    case STKMC_prn:
        if (tracing) fputs(BLANKS, results);
        cpu.sp++;
        if (inbounds(cpu.sp)) fprintf(results, " %d", mem[cpu.sp - 1]);
        if (tracing) putc('\n', results);
        break;
    case STKMC_nln:
        putc('\n', results); break;
    case STKMC_nop:
        break;
    default:
        ps = badop; break;
    }
}
if (nexttorun != 0) chooseprocess();
} while (ps == running);
if (ps != finished) postmortem(results, pcnow);
}

void STKMC::interpret(STKMC_address codelen, STKMC_address initisp)
{
    char filename[256];
    FILE *data, *results;
    bool tracing;
    char reply, dummy;
    printf("\nTrace execution (y/N/q)? ");
    reply = getc(stdin); dummy = reply;
    while (dummy != '\n') dummy = getc(stdin);
    if (toupper(reply) != 'Q')
    {
        tracing = toupper(reply) == 'Y';
        printf("\nData file [STDIN] ? "); gets(filename);
        if (filename[0] == '\0') data = NULL;
        else data = fopen(filename, "r");
        if (data == NULL)
        {
            printf("taking data from stdin\n"); data = stdin;
        }
        printf("\nResults file [STDOUT] ? "); gets(filename);
    }
}

```

```

    if (filename[0] == '\0') results = NULL;
    else results = fopen(filename, "w");
    if (results == NULL)
    { printf("sending results to stdout\n"); results = stdout; }
    emulator(0, codelen, initsp, data, results, tracing);
    if (results != stdout) fclose(results);
    if (data != stdin) fclose(data);
}
}

```

```

STKMC::STKMC()
{ for (int i = 0; i <= STKMC_memsize - 1; i++) mem[i] = 0;
  // Initialize mnemonic table this way for ease of modification in exercises
  mnemonics[STKMC_add] = "ADD"; mnemonics[STKMC_adr] = "ADR";
  mnemonics[STKMC_brn] = "BRN"; mnemonics[STKMC_bze] = "BZE";
  mnemonics[STKMC_cal] = "CAL"; mnemonics[STKMC_cbg] = "CBG";
  mnemonics[STKMC_cnd] = "CND"; mnemonics[STKMC_dsp] = "DSP";
  mnemonics[STKMC_dvd] = "DVD"; mnemonics[STKMC_eql] = "EQL";
  mnemonics[STKMC_frk] = "FRK"; mnemonics[STKMC_geq] = "GEQ";
  mnemonics[STKMC_gtr] = "GTR"; mnemonics[STKMC_hlt] = "HLT";
  mnemonics[STKMC_ind] = "IND"; mnemonics[STKMC_inn] = "INN";
  mnemonics[STKMC_leq] = "LEQ"; mnemonics[STKMC_lit] = "LIT";
  mnemonics[STKMC_lss] = "LSS"; mnemonics[STKMC_mst] = "MST";
  mnemonics[STKMC_mul] = "MUL"; mnemonics[STKMC_neg] = "NEG";
  mnemonics[STKMC_neq] = "NEQ"; mnemonics[STKMC_nfn] = "NFN";
  mnemonics[STKMC_nln] = "NLN"; mnemonics[STKMC_nop] = "NOP";
  mnemonics[STKMC_nul] = "NUL"; mnemonics[STKMC_prn] = "PRN";
  mnemonics[STKMC_prs] = "PRS"; mnemonics[STKMC_ret] = "RET";
  mnemonics[STKMC_sig] = "SIG"; mnemonics[STKMC_stk] = "STK";
  mnemonics[STKMC_sto] = "STO"; mnemonics[STKMC_sub] = "SUB";
  mnemonics[STKMC_val] = "VAL"; mnemonics[STKMC_wgt] = "WGT";
}

```