Appendix C

Cocol grammar for the Clang compiler/interpreter

This appendix gives the Cocol specification and frame file for constructing a compiler for the Clang language as developed by the end of Chapter 18, along with the source for the tree-building code generator.

clang.atg | cgen.h | cgen.cpp | clang.frm

```
----- clang.atg ------------------------------------------------

$CX
COMPILER Clang
/* CLANG level 4 grammar - function, procedures, parameters, concurrency
   Display model.
   Builds an AST for code generation.
   P.D. Terry, Rhodes University, 1996 */

#include "misc.h"
#include "set.h"
#include "table.h"
#include "report.h"
#include "cgen.h"

typedef Set<7> classset;

bool debug;
int blocklevel;
TABLE_idclasses blockclass;

extern TABLE *Table;
extern CGEN *CGen;

/*--------------------------------------------------------------------------*/

IGNORE CASE
IGNORE CHR(9) .. CHR(13)
COMMENTS FROM "(*" TO "*)"

CHARACTERS
  cr         = CHR(13) .
  lf         = CHR(10) .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .
  instring   = ANY - "'" - cr - lf .

TOKENS
  identifier = letter { letter | digit } .
  number     = digit { digit } .
  string     = "'" (instring | "''") { instring | "''" } "'" .

PRODUCTIONS
  Clang
  =                             (. TABLE_entries entry; TABLE_index index; .)
     "PROGRAM"
     Ident<entry.name>          (. debug = (strcmp(entry.name, "DEBUG") == 0);
                                   entry.idclass = TABLE_progs;
                                   Table->enter(entry, index);
                                   Table->openscope(); .)
     WEAK ";"
     Block<entry.level+1, TABLE_progs, 0>
     "." .

  Block<int blklevel, TABLE_idclasses blkclass, int initialframesize>
  =                             (. int framesize = initialframesize;
                                   CGEN_labels entrypoint;
                                   CGen->jump(entrypoint, CGen->undefined);
                                   if (blklevel > CGEN_levmax) SemError(213); .)
     SYNC
     { (   ConstDeclarations
         | VarDeclarations<framesize>
```

```
      | ProcDeclaration
    ) SYNC }                (. /* blockclass, blocklevel global for efficiency */
                               blockclass = blkclass; blocklevel = blklevel;
                               CGen->backpatch(entrypoint);
                               /* reserve space for variables */
                               CGen->openstackframe(framesize
                                                    - initialframesize); .)
   CompoundStatement        (. switch (blockclass)
                               { case TABLE_progs :
                                   CGen->leaveprogram(); break;
                                 case TABLE_procs :
                                   CGen->leaveprocedure(blocklevel); break;
                                 case TABLE_funcs :
                                   CGen->functioncheck(); break;
                               }
                               if (debug) /* demonstration purposes */
                                 Table->printtable(stdout);
                               Table->closescope(); .) .

ConstDeclarations
=  "CONST" OneConst { OneConst } .

OneConst
=                           (. TABLE_entries entry; TABLE_index index; .)
   Ident<entry.name>        (. entry.idclass = TABLE_consts; .)
   WEAK "="
   Number<entry.c.value>    (. Table->enter(entry, index) .)
   ";" .

VarDeclarations<int &framesize>
=  "VAR" OneVar<framesize> { WEAK "," OneVar<framesize> } ";" .

OneVar<int &framesize>
=                           (. TABLE_entries entry; TABLE_index index;
                               entry.idclass = TABLE_vars; entry.v.ref = false;
                               entry.v.size = 1; entry.v.scalar = true;
                               entry.v.offset = framesize + 1; .)
   Ident<entry.name>
   [ UpperBound<entry.v.size> (. entry.v.scalar = false; .)
   ]                        (. Table->enter(entry, index);
                               framesize += entry.v.size; .) .

UpperBound<int &size>
=  "[" Number<size> "]"     (. size++; .) .

ProcDeclaration
=                           (. TABLE_entries entry; TABLE_index index; .)
   (   "PROCEDURE"          (. entry.idclass = TABLE_procs; .)
     | "FUNCTION"           (. entry.idclass = TABLE_funcs; .)
   ) Ident<entry.name>      (. entry.p.params = 0; entry.p.paramsize = 0;
                               entry.p.firstparam = NULL;
                               CGen->storelabel(entry.p.entrypoint);
                               Table->enter(entry, index);
                               Table->openscope() .)
   [
   FormalParameters<entry>  (. Table->update(entry, index) .)
   ] WEAK ";"
   Block<entry.level+1, entry.idclass, entry.p.paramsize + CGEN_headersize>
   ";" .

FormalParameters<TABLE_entries &proc>
=                           (. TABLE_index p; .)
   "(" OneFormal<proc, proc.p.firstparam>
   { WEAK "," OneFormal<proc, p> } ")" .

OneFormal<TABLE_entries &proc, TABLE_index &index>
=                           (. TABLE_entries formal;
                               formal.idclass = TABLE_vars; formal.v.ref = false;
                               formal.v.size = 1; formal.v.scalar = true;
                               formal.v.offset = proc.p.paramsize
                                                 + CGEN_headersize + 1 .)
   Ident<formal.name>
   [ "[" "]"                (. formal.v.size = 2; formal.v.scalar = false;
                               formal.v.ref = true; .)
   ]                        (. Table->enter(formal, index);
                               proc.p.paramsize += formal.v.size;
                               proc.p.params++; .) .

CompoundStatement
=  "BEGIN" Statement { WEAK ";" Statement } "END" .

Statement
=  SYNC [  CompoundStatement | AssignmentOrCall | ReturnStatement
```

```
          | IfStatement      | WhileStatement
          | CobeginStatement | SemaphoreStatement
          | ReadStatement    | WriteStatement
          | "STACKDUMP"       (. CGen->dump(); .)
        ] .

AssignmentOrCall
=                           (. TABLE_entries entry;
                               AST des, exp;.)
   Designator<des, classset(TABLE_vars, TABLE_procs), entry, true>
   (   /* assignment */     (. if (entry.idclass != TABLE_vars) SemError(210); .)
      ":=" Expression<exp, true>
      SYNC                  (. CGen->assign(des, exp); .)
     | /* procedure call */ (. if (entry.idclass < TABLE_procs)
                               { SemError(210); return; }
                               CGen->markstack(des, entry.level,
                                               entry.p.entrypoint); .)
      ActualParameters<des, entry>
                           (. CGen->call(des); .)
   ) .

Designator<AST &D, classset allowed, TABLE_entries &entry, bool entire>
=                           (. TABLE_alfa name;
                               AST index, size;
                               bool found;
                               D = CGen->emptyast(); .)
   Ident<name>              (. Table->search(name, entry, found);
                               if (!found) SemError(202);
                               if (!allowed.memb(entry.idclass)) SemError(206);
                               if (entry.idclass != TABLE_vars) return;
                               CGen->stackaddress(D, entry.level, entry.v.offset,
                                                  entry.v.ref); .)
   (   "["                  (. if (entry.v.scalar) SemError(204); .)
       Expression<index, true>
                           (. if (!entry.v.scalar)
                               /* determine size for bounds check */
                               { if (entry.v.ref)
                                   CGen->stackaddress(size, entry.level,
                                                      entry.v.offset + 1, false);
                                 else
                                   CGen->stackconstant(size, entry.v.size);
                                 CGen->subscript(D, entry.v.ref, entry.level,
                                                 entry.v.offset, size, index);
                               } .)
       "]"
     |                      (. if (!entry.v.scalar)
                               { if (entire) SemError(205);
                                 if (entry.v.ref)
                                   CGen->stackaddress(size, entry.level,
                                                      entry.v.offset + 1, false);
                                 else
                                   CGen->stackconstant(size, entry.v.size);
                                 CGen->stackreference(D, entry.v.ref, entry.level,
                                                      entry.v.offset, size);
                               } .)
   ) .

ActualParameters<AST &p, TABLE_entries proc>
=                           (. int actual = 0; .)
   [  "("                   (. actual++; .)
      OneActual<p, (*Table).isrefparam(proc, actual)>
      { WEAK "," }          (. actual++; .)
      OneActual<p, (*Table).isrefparam(proc, actual)> } ")"
   ]                        (. if (actual != proc.p.params) SemError(209); .) .

OneActual<AST &p, bool byref>
=                           (. AST par; .)
   Expression<par, !byref>  (. if (byref && !CGen->isrefast(par)) SemError(214);
                               CGen->linkparameter(p, par); .) .

ReturnStatement
=                           (. AST dest, exp; .)
   "RETURN"
   (                        (. if (blockclass != TABLE_funcs) SemError(219);
                               CGen->stackaddress(dest, blocklevel, 1, false); .)
       Expression<exp, true>
                           (. CGen->assign(dest, exp);
                               CGen->leavefunction(blocklevel); .)
     | /* empty */          (. switch (blockclass)
                               { case TABLE_procs :
                                   CGen->leaveprocedure(blocklevel); break;
                                 case TABLE_progs :
                                   CGen->leaveprogram(); break;
```

```
                                 case TABLE_funcs : SemError(220); break;
                               } .)
   ) .

IfStatement
=                          (. CGEN_labels testlabel;
                              AST C; .)
   "IF" Condition<C> "THEN" (. CGen->jumponfalse(C, testlabel, CGen->undefined) .)
   Statement               (. CGen->backpatch(testlabel); .) .

WhileStatement
=                          (. CGEN_labels startloop, testlabel, dummylabel;
                              AST  C; .)
   "WHILE"                 (. CGen->storelabel(startloop) .)
   Condition<C> "DO"       (. CGen->jumponfalse(C, testlabel, CGen->undefined) .)
   Statement               (. CGen->jump(dummylabel, startloop);
                              CGen->backpatch(testlabel) .) .

Condition<AST &C>
=                          (. AST E;
                              CGEN_operators op; .)
   Expression<C, true>
   (  RelOp<op>
      Expression<E, true>  (. CGen->comparison(op, C, E); .)
    | /* Missing op */     (. SynError(91) .)
   ) .

CobeginStatement
=                          (. int processes = 0;
                              CGEN_labels start; .)
   "COBEGIN"               (. if (blockclass != TABLE_progs) SemError(215);
                              CGen->cobegin(start); .)
   ProcessCall             (. processes++; .)
   { WEAK ";" ProcessCall  (. processes++; .)
   }
   "COEND"                 (. CGen->coend(start, processes); .) .

ProcessCall
=                          (. TABLE_entries entry;
                              AST P; .)
   Designator<P, classset(TABLE_procs), entry, true>
                           (. if (entry.idclass < TABLE_procs) return;
                              CGen->markstack(P, entry.level,
                                              entry.p.entrypoint); .)
   ActualParameters<P, entry>
                           (. CGen->forkprocess(P); .) .

SemaphoreStatement
=                          (. bool wait;
                              AST sem; .)
   (   "WAIT"              (. wait = true; .)
     | "SIGNAL"            (. wait = false; .)
   )
   "(" Variable<sem>       (. if (wait) CGen->waitop(sem);
                              else CGen->signalop(sem); .)
   ")" .

ReadStatement
=                          (. AST V; .)
   "READ" "(" Variable<V>  (. CGen->readvalue(V); .)
   { WEAK "," Variable<V>  (. CGen->readvalue(V); .)
   } ")" .

Variable<AST &V>
=                          (. TABLE_entries entry; .)
   Designator<V, classset(TABLE_vars), entry, true> .

WriteStatement
=  "WRITE" [ "(" WriteElement { WEAK "," WriteElement }  ")" ]
                           (. CGen->newline(); .) .

WriteElement
=                          (. AST exp;
                              char str[600];
                              CGEN_labels startstring; .)
    String<str>            (. CGen->stackstring(str, startstring);
                              CGen->writestring(startstring); .)
  | Expression<exp, true>  (. CGen->writevalue(exp) .) .

Expression<AST &E, bool entire>
=                          (. AST T;
                              CGEN_operators op;
                              E = CGen->emptyast(); .)
```

```
   (   "+" Term<E, true>
     | "-" Term<E, true>    (. CGen->negateinteger(E); .)
     | Term<E, entire>
   )
   { AddOp<op> Term<T, true>(. CGen->binaryintegerop(op, E, T); .)
   } .

Term<AST &T, bool entire>
=                          (. AST F;
                              CGEN_operators op; .)
   Factor<T, entire>
   { (  MulOp<op>
     | /* missing op */    (. SynError(92); op = CGEN_opmul; .)
     ) Factor<F, true>     (. CGen->binaryintegerop(op, T, F); .)
   } .

Factor<AST &F, bool entire>
=                          (. TABLE_entries entry;
                              int value;
                              F = CGen->emptyast(); .)
   Designator<F, classset(TABLE_consts, TABLE_vars, TABLE_funcs), entry, entire>
                           (. switch (entry.idclass)
                              { case TABLE_consts :
                                  CGen->stackconstant(F, entry.c.value);
                                  return;
                                case TABLE_procs :
                                case TABLE_funcs :
                                  CGen->markstack(F, entry.level,
                                                  entry.p.entrypoint); break;
                                case TABLE_vars :
                                case TABLE_progs :
                                  return;
                              } .)
   ActualParameters<F, entry>
 | Number<value>           (. CGen->stackconstant(F, value) .)
 | "(" Expression<F, true> ")" .

AddOp<CGEN_operators &op>
=    "+"                   (. op = CGEN_opadd; .)
   | "-"                   (. op = CGEN_opsub; .) .

MulOp<CGEN_operators &op>
=    "*"                   (. op = CGEN_opmul; .)
   | "/"                   (. op = CGEN_opdvd; .) .

RelOp<CGEN_operators &op>
=    "="                   (. op = CGEN_opeql; .)
   | "<>"                  (. op = CGEN_opneq; .)
   | "<"                   (. op = CGEN_oplss; .)
   | "<="                  (. op = CGEN_opleq; .)
   | ">"                   (. op = CGEN_opgtr; .)
   | ">="                  (. op = CGEN_opgeq; .) .

Ident<char *name>
=  identifier              (. LexName(name, TABLE_alfalength); .) .

String<char *str>
=  string                  (. char local[100];
                              LexString(local, sizeof(local) - 1);
                              int i = 0;
                              while (local[i]) /* strip quotes */
                              { local[i] = local[i+1]; i++; }
                              local[i-2] = '\0';
                              i = 0;
                              while (local[i]) /* find internal quotes */
                              { if (local[i] == '\'')
                                  { int j = i;
                                    while (local[j])
                                    { local[j] = local[j+1]; j++; }
                                  }
                                  i++;
                              }
                              strcpy(str, local); .) .

Number <int &num>
=  number                  (. char str[100];
                              int i = 0, l, digit, overflow = 0;
                              num = 0;
                              LexString(str, sizeof(str) - 1);
                              l = strlen(str);
                              while (i <= l && isdigit(str[i])) {
                                digit = str[i] - '0'; i++;
                                if (num <= (maxint - digit) / 10)
```

```
                                    num = 10 * num + digit;
                                  else overflow = 1;
                                }
                                if (overflow) SemError(200); .) .

END Clang.


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

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

#ifndef 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
};

struct NODE;
typedef NODE *AST;
typedef short CGEN_labels;

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

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

    AST emptyast(void);
    // Returns an empty (undefined) AST

    bool isrefast(AST a);
    // Returns true if a corresponds to a reference AST

    void negateinteger(AST &i);
    // Generates code to negate integer i

    void binaryintegerop(CGEN_operators op, AST &l, AST &r);
    // Generates code to perform infix operation op on l, r

    void comparison(CGEN_operators op, AST &l, AST &r);
    // Generates code to perform comparison operation op on l, r

    void readvalue(AST i);
    // Generates code to read value for i

    void writevalue(AST i);
    // Generates code to output value i

    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(AST &c, int number);
    // Creates constant AST for constant c from number

    void stackaddress(AST &v, int level, int offset, bool byref);
    // Creates address AST for variable v with known level, offset

    void dereference(AST &a);
    // Generates code to replace address a by the value stored there

    void stackreference(AST &base, bool byref,
                        int level, int offset, AST &size);
```

```
    // Creates an actual parameter node for a reference parameter corresponding
    // to an array with given base and size

    void subscript(AST &base, bool byref,
                   int level, int offset, AST &size, AST &index);
    // Prepares to apply an index to an array with given base, with checks
    // that the limit on the bounds is not exceeded

    void assign(AST dest, AST expr);
    // Generates code to store value of expr on dest

    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 a given blocklevel

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

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

    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(AST condition, CGEN_labels &here, CGEN_labels destination);
    // Generates branch from here to destination, dependent on condition

    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(AST &p, int level, int entrypoint);
    // Generates code to reserve mark stack storage before calling procedure p
    // with known level and entrypoint

    void linkparameter(AST &p, AST &par);
    // Adds par to the actual parameter list for call to procedure p

    void call(AST &p);
    // Generates code to enter procedure p

    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 forkprocess(AST &p);
    // Generates code to initiate process p

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

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

    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);
    // Returns codetop

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

  private:
    REPORT *Report;
    bool generatingcode;
    STKMC_address codetop, stktop;
    void binaryop(CGEN_operators op, AST &left, AST &right);
};
```

```
#endif /*CGEN_H*/


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

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

#include "misc.h"
#include "cgen.h"
#include "report.h"

extern STKMC  *Machine;
extern CGEN   *CGen;
extern REPORT *Report;

// ++++++++++++++ AST node classes ++++++++++++++++++++++++++++++++++++++++

struct NODE {
  int value;       // value derived from this node
  bool defined;    // true if value is defined (for constant nodes)
  bool refnode;    // true if node corresponds to a reference
  NODE()                          { defined = false; refnode = false; }
  virtual void emit1(void)    = 0;
  virtual void emit2(void)    = 0;
  virtual void link(AST next) = 0;
};

struct BINOPNODE : public NODE {
  CGEN_operators op;
  AST left, right;
  BINOPNODE(CGEN_operators O, AST L, AST R) { op = O; left = L; right = R; }
  virtual void emit1(void);      // load value onto stack
  virtual void emit2(void)       {;}
  virtual void link(AST next)    {;}
};

void BINOPNODE::emit1(void)
// load value onto stack resulting from binary operation
{ bool folded = false;
  if (left && right)
  { // Some optimizations (others are left as an exercise).
    // These need improvement so as to perform range checking
    switch (op)
    { case CGEN_opadd:
        if (right->defined && right->value == 0) // x + 0 = x
          { left->emit1(); folded = true; }
        else if (left->defined && left->value == 0) // 0 + x = x
          { right->emit1(); folded = true; }
        break;

      case CGEN_opsub:
        if (right->defined && right->value == 0) // x - 0 = x
          { left->emit1(); folded = true; }
        else if (left->defined && left->value == 0) // 0 - x = -x
          { right->emit1(); CGen->emit(int(STKMC_neg)); folded = true; }
        break;

      case CGEN_opmul:
        if (right->defined && right->value == 1) // x * 1 = x
          { left->emit1(); folded = true; }
        else if (left->defined && left->value == 1) // 1 * x = x
          { right->emit1(); folded = true; }
        else if (right->defined && right->value == 0) // x * 0 = 0
          { right->emit1(); folded = true; }
        else if (left->defined && left->value == 0) // 0 * x = 0
          { left->emit1(); folded = true; }
        break;

      case CGEN_opdvd:
        if (right->defined && right->value == 1) // x / 1 = x
          { left->emit1(); folded = true; }
        else if (right->defined && right->value == 0) // x / 0 = error
          { Report->error(224); folded = true; }
        break;
      // no folding attempted here for relational operations
    }
  }
  if (!folded)
  { if (left) left->emit1(); if (right) right->emit1();
```

```
    CGen->emit(int(STKMC_add) + int(op));   // careful - ordering used
  }
  if (left) delete left; if (right) delete right;
}

struct MONOPNODE : public NODE {
  CGEN_operators op;  // for expansion - only negation used here
  AST operand;
  MONOPNODE(CGEN_operators O, AST E) { op = O; operand = E; }
  virtual void emit1(void);      // load value onto stack
  virtual void emit2(void)       {;}
  virtual void link(AST next)    {;}
};

void MONOPNODE::emit1(void)
// load value onto stack resulting from unary operation
{ if (operand) { operand->emit1(); delete operand; }
  CGen->emit(int(STKMC_neg));
}

struct VARNODE : public NODE {
  bool ref;       // direct or indirectly accessed
  int level;      // static level of declaration
  int offset;     // offset of variable assigned by compiler
  VARNODE() {;}   // default
  VARNODE(bool byref, int L, int O)
    { ref = byref; level = L; offset = O; }
  virtual void emit1(void);      // load variable value onto stack
  virtual void emit2(void);      // load variable address onto stack
  virtual void link(AST next)    {;}
};

void VARNODE::emit1(void)
// load variable value onto stack
{ emit2(); CGen->emit(int(STKMC_val)); }

void VARNODE::emit2(void)
// load variable address onto stack
{ CGen->emit(int(STKMC_adr)); CGen->emit(level); CGen->emit(-offset);
  if (ref) CGen->emit(int(STKMC_val));
}

struct INDEXNODE : public VARNODE {
  AST size;      // for range checking
  AST index;     // subscripting expression
  INDEXNODE(bool byref, int L, int O, AST S, AST I)
    { ref = byref; level = L; offset = O; size = S; index = I; }
  virtual void emit2(void);      // load array element address and check
  virtual void link(AST next)    {;}
};

void INDEXNODE::emit2(void)
// load array element address and check in range
{ CGen->emit(int(STKMC_adr)); CGen->emit(level); CGen->emit(-offset);
  if (ref) CGen->emit(int(STKMC_val));
  if (index) { index->emit1(); delete index; }
  if (size) { size->emit1(); delete size; }
  CGen->emit(int(STKMC_ind));
}

// void INDEXNODE::emit1(void) is inherited from VARNODE

struct REFNODE : public VARNODE {
  AST size;
  REFNODE(bool byref, int L, int O, AST S)
    { ref = byref; level = L; offset = O; size = S; refnode = 1; }
  virtual void emit1(void);       // load array argument address and size
  virtual void emit2(void)        {;}
  virtual void link(AST next)     {;}
};

void REFNODE::emit1(void)
// load array argument address and size
{ CGen->emit(int(STKMC_adr)); CGen->emit(level); CGen->emit(-offset);
  if (ref) CGen->emit(int(STKMC_val));
  if (size) { size->emit1(); delete size; }
}

struct CONSTNODE : public NODE {
  CONSTNODE(int V)               { value = V; defined = true; }
  virtual void emit1(void);      // load constant value onto stack
  virtual void emit2(void)       {;}
  virtual void link(AST next)    {;}
```

```
};

void CONSTNODE::emit1(void)
// load constant value onto stack
{ CGen->emit(int(STKMC_lit)); CGen->emit(value); }

struct PARAMNODE : public NODE {
  AST par, next;
  PARAMNODE(AST P)               { par = P; next = NULL; }
  virtual void emit1(void);      // load actual parameter onto stack
  virtual void emit2(void)       {;}
  virtual void link(AST param)   { next = param; }
};

void PARAMNODE::emit1(void)
// load actual parameter onto stack
{ if (par) { par->emit1(); delete par; }
  if (next) { next->emit1(); delete next; }  // follow link to next parameter
}

struct PROCNODE : public NODE {
  int level, entrypoint;     // static level and first instruction
  AST firstparam, lastparam; // pointers to argument list
  PROCNODE(int L, int ent)
    { level = L; entrypoint = ent; firstparam = NULL; lastparam = NULL; }
  virtual void emit1(void);      // generate procedure/function call
  virtual void emit2(void);      // generate process call
  virtual void link(AST next);   // link next actual parameter
};

void PROCNODE::emit1(void)
// generate procedure/function call
{ CGen->emit(int(STKMC_mst));
  if (firstparam) { firstparam->emit1(); delete firstparam; }
  CGen->emit(int(STKMC_cal));
  CGen->emit(level);
  CGen->emit(entrypoint);
}

void PROCNODE::emit2(void)
// generate process call
{ CGen->emit(int(STKMC_mst));
  if (firstparam) { firstparam->emit1(); delete firstparam; }
  CGen->emit(int(STKMC_frk));
  CGen->emit(entrypoint);
}

void PROCNODE::link(AST param)
// link next actual parameter
{ if (!firstparam) firstparam = param; else lastparam->link(param);
  lastparam = param;
}

// +++++++++++++++ code generator constructor ++++++++++++++++++++++++++

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++; }
}

bool CGEN::isrefast(AST a)
{ return a && a->refnode; }

// +++++++++++++++ routines that build the tree ++++++++++++++++++++++++++

AST CGEN::emptyast(void)
{ return NULL; }

void CGEN::negateinteger(AST &i)
{ if (i && i->defined) { i->value = -i->value; return; }  // simple folding
  i = new MONOPNODE(CGEN_opsub, i);
}
```

```
void CGEN::binaryop(CGEN_operators op, AST &left, AST &right)
{ if (left && right && left->defined && right->defined)
  { // simple constant folding - better range checking needed
    switch (op)
    { case CGEN_opadd: left->value += right->value; break;
      case CGEN_opsub: left->value -= right->value; break;
      case CGEN_opmul: left->value *= right->value; break;
      case CGEN_opdvd:
        if (right->value == 0) Report->error(224);
        else left->value /= right->value;
        break;
      case CGEN_oplss: left->value = (left->value < right->value); break;
      case CGEN_opgtr: left->value = (left->value > right->value); break;
      case CGEN_opleq: left->value = (left->value <= right->value); break;
      case CGEN_opgeq: left->value = (left->value >= right->value); break;
      case CGEN_opeql: left->value = (left->value == right->value); break;
      case CGEN_opneq: left->value = (left->value != right->value); break;
    }
    delete right;
    return;
  }
  left = new BINOPNODE(op, left, right);
}

void CGEN::binaryintegerop(CGEN_operators op, AST &l, AST &r)
{ binaryop(op, l, r); }

void CGEN::comparison(CGEN_operators op, AST &l, AST &r)
{ binaryop(op, l, r); }

void CGEN::stackconstant(AST &c, int number)
{ c = new CONSTNODE(number); }

void CGEN::stackaddress(AST &v, int level, int offset, bool byref)
{ v = new VARNODE(byref, level, offset); }

void CGEN::linkparameter(AST &p, AST &par)
{ AST param = new PARAMNODE(par); p->link(param); }

void CGEN::stackreference(AST &base, bool byref, int level, int offset,
                          AST &size)
{ if (base) delete base; base = new REFNODE(byref, level, offset, size); }

void CGEN::subscript(AST &base, bool byref, int level, int offset,
                     AST &size, AST &index)
// Note the folding of constant indexing of arrays, and compile time
// range checking
{ if (!index || !index->defined || !size || !size->defined)
  { if (base) delete base;
    base = new INDEXNODE(byref, level, offset, size, index);
    return;
  }
  if (unsigned(index->value) >= size->value) // range check immediately
    Report->error(223);
  else
  { if (base) delete base;
    base = new VARNODE(byref, level, offset + index->value);
  }
  delete index; delete size;
}

void CGEN::markstack(AST &p, int level, int entrypoint)
{ p = new PROCNODE(level, entrypoint); }

// +++++++++++++++ code generation requiring tree walk ++++++++++++++++++

void CGEN::jumponfalse(AST condition, CGEN_labels &here,
                       CGEN_labels destination)
{ if (condition) { condition->emit1(); delete condition; }
  here = codetop; emit(int(STKMC_bze)); emit(destination);
}

void CGEN::assign(AST dest, AST expr)
{ if (dest) { dest->emit2(); delete dest; }
  if (expr) { expr->emit1(); delete expr; emit(int(STKMC_sto)); }
}

void CGEN::readvalue(AST i)
{ if (i) { i->emit2(); delete i; } emit(int(STKMC_inn)); }

void CGEN::writevalue(AST i)
{ if (i) { i->emit1(); delete i; } emit(int(STKMC_prn)); }
```

```
void CGEN::call(AST &p)
{ if (p) { p->emit1(); delete p; } }

void CGEN::signalop(AST s)
{ if (s) { s->emit2(); delete s; } emit(int(STKMC_sig)); }

void CGEN::waitop(AST s)
{ if (s) { s->emit2(); delete s; } emit(int(STKMC_wgt)); }

void CGEN::forkprocess(AST &p)
{ if (p) { p->emit2(); delete p; } }

// +++++++++++++++ code generation not requiring tree walk ++++++++++++++

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 + l + 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::dereference(AST &a)
{ /* not needed */ }

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

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

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


----- clang.frm -----------------------------------------------------

/* Clang compiler generated by Coco/R 1.06 (C++ version) */

#include  <stdio.h>
```

```
#include  <fcntl.h>
#include  <stdlib.h>
#include  <string.h>

#if __MSDOS__ || MSDOS || WIN32 || __WIN32__
#  include  <io.h>
#else
#  include <unistd.h>
#  define  O_BINARY     0
#endif

#include "misc.h"
#include "set.h"
#include "table.h"
#include "cgen.h"
#include "stkmc.h"

typedef Set<7> classset;

#include -->ScanHeader
#include -->ParserHeader
#include "cr_error.hpp"

static FILE *lst;
static char SourceName[256], ListName[256], CodeName[256];

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

class clangError : public CRError {
  public:
    clangError(char *name, AbsScanner *S) : CRError(name, S, MAXERROR) {};
    virtual char *GetUserErrorMsg(int n);
    virtual char *GetErrorMsg(int n)
      { if (n <= MAXERROR) return ErrorMsg[n]; else return GetUserErrorMsg(n); }
  private:
    static char *ErrorMsg[];
};

char *clangError::ErrorMsg[] = {
#include -->ErrorHeader
"User error number clash",
""
};

char *clangError::GetUserErrorMsg(int n)
{ switch (n) {
  // first few are extra syntax errors
    case 91:  return "Relational operator expected";
    case 92:  return "Malformed expression";
    case 93:  return "Bad declaration order";
  // remainder are constraint (static semantic) errors
    case 200: return "Constant out of range";
    case 201: return "Identifier redeclared";
    case 202: return "Undeclared identifier";
    case 203: return "Unexpected parameters";
    case 204: return "Unexpected subscript";
    case 205: return "Subscript required";
    case 206: return "Invalid class of identifier";
    case 207: return "Variable expected";
    case 208: return "Too many formal parameters";
    case 209: return "Wrong number of parameters";
    case 210: return "Invalid assignment";
    case 211: return "Cannot read this type of variable";
    case 212: return "Program too long";
    case 213: return "Too deeply nested";
    case 214: return "Invalid parameter";
    case 215: return "COBEGIN only allowed in main program";
    case 216: return "Too many concurrent processes";
    case 217: return "Only global procedure calls allowed here";
    case 218: return "Type mismatch";
    case 219: return "Unexpected expression";
    case 220: return "Missing expression";
    case 221: return "Boolean expression required";
    case 222: return "Invalid expression";
    case 223: return "Index out of range";
    case 224: return "Division by zero";
    default:  return "Compiler error";
  }
}
```

```
class clangReport : public REPORT {
// interface for code generators and other auxiliaries
  public:
    clangReport(clangError *E)
      { Error = E; }
    virtual void error(int errorcode)
      { Error->ReportError(errorcode); errors = true; }
  private:
    clangError *Error;
};

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

  // check on correct parameter usage
  if (argc < 2) { fprintf(stderr, "No input file specified\n"); exit(1); }

  // open the source file
  strcpy(SourceName, argv[1]);
  if ((S_src = open(SourceName, O_RDONLY | O_BINARY)) == -1)
  { fprintf(stderr, "Unable to open input file %s\n", SourceName); exit(1); }

  if (argc > 2) strcpy(ListName, argv[2]);
    else appendextension(SourceName, ".lst", ListName);
  if ((lst = fopen(ListName, "w")) == NULL)
  { fprintf(stderr, "Unable to open list file %s\n", ListName); exit(1); }

  // instantiate Scanner, Parser and Error handlers

  -->ScanClass *Scanner = new -->ScanClass(S_src, -->IgnoreCase);
  clangError   *Error = new clangError(SourceName, Scanner);
  -->ParserClass  *Parser = new -->ParserClass(Scanner, Error);
  Report  = new clangReport(Error);
  CGen    = new CGEN(Report);
  Table   = new TABLE(Report);
  Machine = new STKMC();

  // parse the source
  Parser->Parse();
  close(S_src);

  // generate source listing
  Error->SetOutput(lst);
  Error->PrintListing(Scanner);
  fclose(lst);

  // list generated code for interest
  CGen->getsize(codelength, initsp);
  appendextension(SourceName, ".cod", CodeName);
  Machine->listcode(CodeName, codelength);

  if (Error->Errors)
    fprintf(stderr, "Compilation failed - see %s\n", ListName);
  else
  { fprintf(stderr, "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 Scanner;
  delete Parser;
  delete Error;
  delete Table;
  delete Report;
  delete CGen;
  delete Machine;
}
```