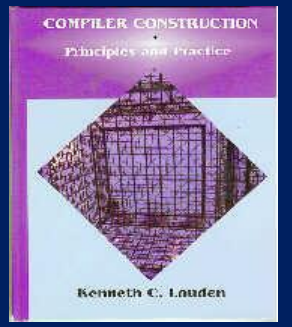
CS 445: Assignment 5

codegen.cpp

emitcode.cpp

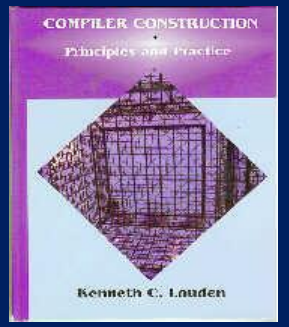
emitcode.h

codegen.h




```
//
//  REGISTER DEFINES for optional use in calling the
//  routines below.
//
#define GP   0    //  The global pointer
#define FP   1    //  The local frame pointer
#define RT   2    //  Return value
#define AC   3    //  Accumulator
#define AC1  4    //  Accumulator
#define AC2  5    //  Accumulator
#define AC3  6    //  Accumulator
#define PC   7    //  The program counter
```

codegen.h

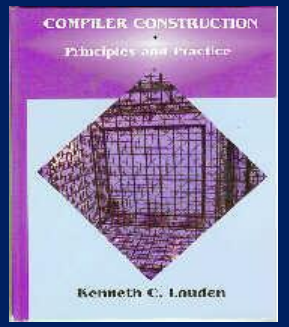



```
//
//  No comment please...
//
#define NO_COMMENT (char *)""
```

```
int emitWhereAmI();          // gives where the next instruction will be placed – Use emitSkip(0) instead
int emitSkip(int howMany);   // emitSkip(0) tells you where the next instruction will be placed
void emitNewLoc(int loc);    // set the instruction counter back to loc

void emitComment(char *c);
void emitComment(char *c, char *cc);
void emitComment(char *c, int n);

void emitGoto(int d, long long int s, char *c);
void emitGoto(int d, long long int s, char *c, char *cc);
void emitGotoAbs(int a, char *c);
void emitGotoAbs(int a, char *c, char *cc);

void emitRM(char *op, long long int r, long long int d, long long int s, char *c);
void emitRM(char *op, long long int r, long long int d, long long int s, char *c, char *cc);
void emitRMAbs(char *op, long long int r, long long int a, char *c);
void emitRMAbs(char *op, long long int r, long long int a, char *c, char *cc);

void emitRO(char *op, long long int r, long long int s, long long int t, char *c);
void emitRO(char *op, long long int r, long long int s, long long int t, char *c, char *cc);

void backPatchAJumpToHere(int addr, char *comment);
void backPatchAJumpToHere(char *cmd, int reg, int addr, char *comment);

int emitStrLit(int goffset, char *s); // for char arrays
```

emitcode.h

You need to make a function/rule for each node kind



void codegenStatement(TreeNode *currnode)
switch (currnode->kind.stmt)

- IfK
- WhileK
- ForK
- CompoundK
- ReturnK
- BreakK
- RangeK

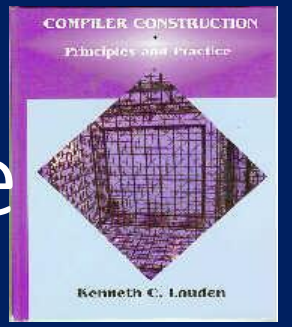
void codegenExpression(TreeNode *currnode)

- AssignK
- CallK
- ConstantK
- IdK
- OpK

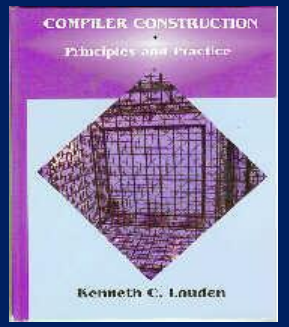
void codegenDecl(TreeNode *currnode)
switch(currnode->kind.decl) {

- VarK
- FuncK
- ParamK

Let’s compile the simplest program possible




```
#DRBC This is the simplest program possible!
main() {
}
```

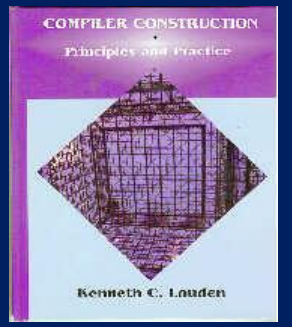



```
void codegen(FILE *codeIn,          // where the code should be written
             char *srcFile,         // name of file compiled
             TreeNode *syntaxTree,  // tree to process
             SymbolTable *globalsIn,// globals so function info can be found
             int globalOffset,      // size of the global frame
             bool linenumFlagIn);   // comment with line numbers
```

In your main:

```
syntaxTree = semanticAnalysis(syntaxTree, true, false, symtab, globalOffset); // Previous assignment
codegen(stdout, (char *)argv[1], syntaxTree, symtab, globalOffset, false);
```

srcFile is likely argv[1];

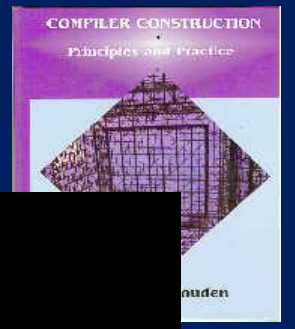



```
extern int numErrors;
extern int numWarnings;
extern void yyparse();
extern int yydebug;
extern TreeNode *syntaxTree;
extern char **largerTokens;
extern void initTokenStrings();

// These offsets that never change
#define OFPOFF  0
#define RETURNOFFSET   -1

int toffset;   // next available temporary space


FILE *code;                         // shared global code – already included
static bool linenumFlag;            // mark with line numbers
static int breakloc;                // which while to break to
static SymbolTable *globals;        // the global symbol table


// this is the top level code generator call
void codegen(FILE *codeIn,          // where the code should be written
        char *srcFile,              // name of file compiled
        TreeNode *syntaxTree,       // tree to process
        SymbolTable *globalsIn,     // globals so function info can be found
        int globalOffset,
        bool linenumFlagIn)
{
   int initJump;

   code = codeIn;
   globals = globalsIn;
   linenumFlag = linenumFlagIn;
   breakloc = 0;

   initJump = emitSkip(1);                      // save a place for the jump to init
   codegenHeader(srcFile);                      // nice comments describing what is compiled
   codegenGeneral(syntaxTree);                  // general code generation including I/O library
   codegenInit(initJump, globalOffset);         // generation of initialization for run
}
```

Dr. BC Note: What was the point of making these static?

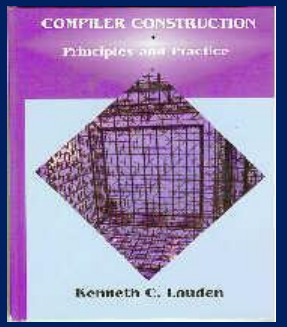
Comment out these lines. We will add these functions one at a time and then uncomment them.

Table of contents




```
FILE *code;

// this is the top level code generator call
void codegen(FILE *codeIn,                  // where the code should be written
             char *srcFile,                 // name of file compiled
             TreeNode *syntaxTree,    // tree to process
             SymbolTable *globalsIn,        // globals so function info can be found
             int globalOffset,
             bool linenumFlagIn)
{
   int initJump;

   code = codeIn;
   globals = globalsIn;
   linenumFlag = linenumFlagIn;
   breakloc = 0;

   initJump = emitSkip(1);                  // save a place for the jump to init
   codegenHeader(srcFile);                  // nice comments describing what is compiled
   codegenGeneral(syntaxTree);                      // general code generation including I/O library
   codegenInit(initJump, globalOffset);             // generation of initialization for run
}
```

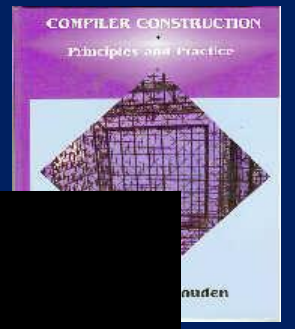
codegenHeader(srcFile);




```
// Generate a header for our code
void codegenHeader(char *srcFile)
{
   emitComment((char *)"bC compiler version bC-Su23");
   emitComment((char *)"File compiled: ", srcFile);
}
```

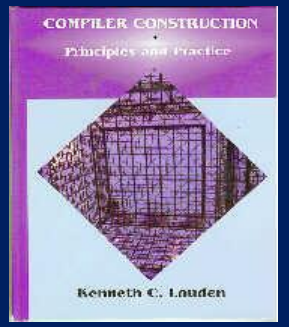
```
* bC compiler version bC-Su23
* File compiled:  test00.bC
```

Table of contents



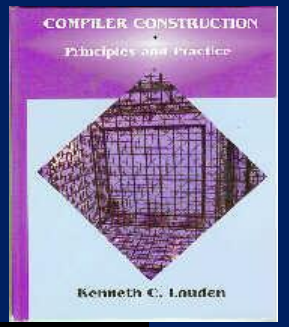

```
// this is the top level code generator call
void codegen(FILE *codeIn,                  // where the code should be written
             char *srcFile,                 // name of file compiled
             TreeNode *syntaxTree,    // tree to process
             SymbolTable *globalsIn,        // globals so function info can be found
             int globalOffset,
             bool linenumFlagIn)
{
   int initJump;

   code = codeIn;
   globals = globalsIn;
   linenumFlag = linenumFlagIn;
   breakloc = 0;

   initJump = emitSkip(1);                  // save a place for the jump to init
   codegenHeader(srcFile);                  // nice comments describing what is compiled
   codegenGeneral(syntaxTree);                      // general code generation including I/O library
   codegenInit(initJump, globalOffset);             // generation of initialization for run
}
```

codegenGeneral(syntaxTree);



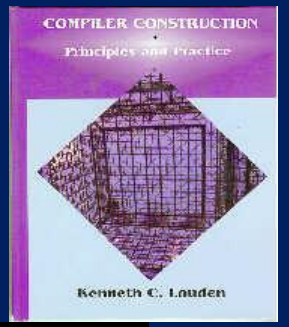

```
void codegenGeneral(TreeNode *currnode)
{
  while (currnode) {
    switch (currnode->nodekind) {
      case StmtK:
        codegenStatement(currnode);
        break;
      case ExpK:
        emitComment((char *)"EXPRESSION");
        codegenExpression(currnode);
        break;
      case DeclK:
        codegenDecl(currnode);
        break;
    }
    currnode = currnode->sibling;
  }
}
```

void codegenDecl(TreeNode *currnode)




```
// given the syntax tree for declarations generate the code
void codegenDecl(TreeNode *currnode)
{
   commentLineNum(currnode);

   switch(currnode->kind.decl) {
   case VarK:
      // You have a LOT to do here!!!!!
      break;
   case FuncK:
      if (currnode->lineno == -1) {          // These are the library functions we just added
         codegenLibraryFun(currnode);
      }
      else {
         codegenFun(currnode);
      }
      break;
   case ParamK:
      // IMPORTANT: no instructions need to be allocated for parameters here
      break;
   }
}
```

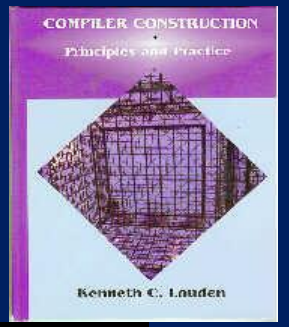
void codegenDecl(TreeNode *currnode)




```
// given the syntax tree for declarations generate the code
void codegenDecl(TreeNode *currnode)
{
   commentLineNum(currnode);

   switch(currnode->kind.decl) {
   case VarK:
      // You have a LOT to do here!!
      break;
   case FuncK:
      if (currnode->lineno == -1) {
         codegenLibraryFun(currnod
      }
      else {
         codegenFun(currnode);
      }
      break;
   case ParamK:
      // IMPORTANT: no instruction
      break;
   }
}
```

```
void commentLineNum(TreeNode *currnode)
{
   char buf[16];

   if (linenumFlag) {
      sprintf(buf, "%d", currnode->lineno);
      emitComment((char *)"Line: ", buf);
   }
}
```

void codegenDecl(TreeNode *currnode)




```
// given the syntax tree for declarations generate the code
void codegenDecl(TreeNode *currnode)
{
   commentLineNum(currnode);

   switch(currnode->kind.decl) {
   case VarK:
      // You have a LOT to do here!!!!!
      break;
   case FuncK:
      if (currnode->lineno == -1) {          // These are the library functions we just added
         codegenLibraryFun(currnode);
      }
      else {
         codegenFun(currnode);
      }
      break;
   case ParamK:
      // IMPORTANT: no instructions need to be allocated for parameters here
      break;
   }
}
```

void codegenLibraryFun(TreeNode *currnode)


```
void codegenLibraryFun(TreeNode *currnode)
{
   emitComment((char *)"");
   emitComment((char *)"** ** ** ** ** ** ** ** ** ** ** **");
   emitComment((char *)"FUNCTION", currnode->attr.name);

   // remember where this function is
   currnode->offset = emitSkip(0);

   // Store return address
   emitRM((char *)"ST", AC, RETURNOFFSET, FP, (char *)"Store return address");

// Next slides here

   emitRM((char *)"LD", AC, RETURNOFFSET, FP, (char *)"Load return address");
   emitRM((char *)"LD", FP, OFPOFF, FP, (char *)"Adjust fp");
   emitGoto(0, AC, (char *)"Return");

   emitComment((char *)"END FUNCTION", currnode->attr.name);
}
```

```
*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION input
  1:    ST  3,-1(1)   Store return address
  2:    IN  2,2,2     Grab int input
  3:    LD  3,-1(1)   Load return address
  4:    LD  1,0(1)    Adjust fp
  5:   JMP  7,0(3)    Return
* END FUNCTION input
```

codegenLibraryFun

```
 else if (strcmp(currnode->attr.name, (char *)"input")==0) {
  emitRO((char *)"IN", RT, RT, RT, (char *)"Grab int input");
}
else if (strcmp(currnode->attr.name, (char *)"inputb")==0) {
  emitRO((char *)"INB", RT, RT, RT, (char *)"Grab bool input");
}
else if (strcmp(currnode->attr.name, (char *)"inputc")==0) {
  emitRO((char *)"INC", RT, RT, RT, (char *)"Grab char input");
}
```

```
*
* ** ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION input
  1:    ST  3,-1(1)   Store return address
  2:    IN  2,2,2     Grab int input
  3:    LD  3,-1(1)   Load return address
  4:    LD  1,0(1)    Adjust fp
  5:   JMP  7,0(3)    Return
* END FUNCTION input
*
* ** ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION output
  6:    ST  3,-1(1)   Store return address
  7:    LD  3,-2(1)   Load parameter
  8:   OUT  3,3,3     Output integer
  9:    LD  3,-1(1)   Load return address
 10:    LD  1,0(1)    Adjust fp
 11:   JMP  7,0(3)    Return
* END FUNCTION output
*
* ** ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION inputb
 12:    ST  3,-1(1)   Store return address
 13:   INB  2,2,2     Grab bool input
 14:    LD  3,-1(1)   Load return address
 15:    LD  1,0(1)    Adjust fp
 16:   JMP  7,0(3)    Return
* END FUNCTION inputb
*
* ** ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION outputb
 17:    ST  3,-1(1)   Store return address
 18:    LD  3,-2(1)   Load parameter
 19:  OUTB  3,3,3     Output bool
 20:    LD  3,-1(1)   Load return address
 21:    LD  1,0(1)    Adjust fp
 22:   JMP  7,0(3)    Return
* END FUNCTION outputb
*
* ** ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION inputc
 23:    ST  3,-1(1)   Store return address
 24:   INC  2,2,2     Grab char input
 25:    LD  3,-1(1)   Load return address
 26:    LD  1,0(1)    Adjust fp
 27:   JMP  7,0(3)    Return
```

codegenLibraryFun

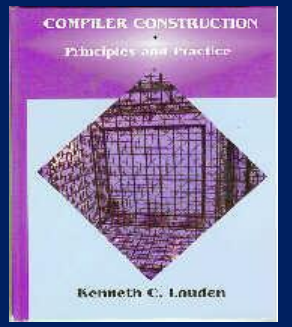
```
 else if (strcmp(currnode->attr.name, (char *)"input")==0) {
   emitRO((char *)"IN", RT, RT, RT, (char *)"Grab int input");
 }
 else if (strcmp(currnode->attr.name, (char *)"inputb")==0) {
   emitRO((char *)"INB", RT, RT, RT, (char *)"Grab bool input");
 }
 else if (strcmp(currnode->attr.name, (char *)"inputc")==0) {
   emitRO((char *)"INC", RT, RT, RT, (char *)"Grab char input");
 }
 else if (strcmp(currnode->attr.name, (char *)"output")==0) {
   emitRM((char *)"LD", AC, -2, FP, (char *)"Load parameter");
   emitRO((char *)"OUT", AC, AC, AC, (char *)"Output integer");
 }
 else if (strcmp(currnode->attr.name, (char *)"outputb")==0) {
   emitRM((char *)"LD", AC, -2, FP, (char *)"Load parameter");
   emitRO((char *)"OUTB", AC, AC, AC, (char *)"Output bool");
 }
 else if (strcmp(currnode->attr.name, (char *)"outputc")==0) {
   emitRM((char *)"LD", AC, -2, FP, (char *)"Load parameter");
   emitRO((char *)"OUTC", AC, AC, AC, (char *)"Output char");
 }
```

```
* END FUNCTION input
*
* ** ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION output
  6:    ST  3,-1(1)   Store return address
  7:    LD  3,-2(1)   Load parameter
  8:   OUT  3,3,3     Output integer
  9:    LD  3,-1(1)   Load return address
 10:    LD  1,0(1)    Adjust fp
 11:   JMP  7,0(3)    Return
* END FUNCTION output
*
* ** ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION inputb
 12:   ST  3,-1(1)   Store return address
 13:  INB  2,2,2     Grab bool input
 14:   LD  3,-1(1)   Load return address
 15:   LD  1,0(1)    Adjust fp
 16:  JMP  7,0(3)    Return
* END FUNCTION inputb
*
* ** ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION outputb
 17:    ST  3,-1(1)   Store return address
 18:    LD  3,-2(1)   Load parameter
 19:  OUTB  3,3,3     Output bool
 20:    LD  3,-1(1)   Load return address
 21:    LD  1,0(1)    Adjust fp
 22:   JMP  7,0(3)    Return
* END FUNCTION outputb
*
* ** ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION inputc
 23:   ST  3,-1(1)   Store return address
 24:  INC  2,2,2     Grab char input
 25:   LD  3,-1(1)   Load return address
 26:   LD  1,0(1)    Adjust fp
 27:  JMP  7,0(3)    Return
* END FUNCTION inputc
*
* ** ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION outputc
 28:    ST  3,-1(1)   Store return address
 29:    LD  3,-2(1)   Load parameter
 30:  OUTC  3,3,3     Output char
 31:    LD  3,-1(1)   Load return address
 32:    LD  1,0(1)    Adjust fp
 33:   JMP  7,0(3)    Return
* END FUNCTION outputc
*
* ** ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION outnl
```

codegenLibraryFun

```
  if (strcmp(currnode->attr.name, (char *)"input")==0) {
    emitRO((char *)"IN", RT, RT, RT, (char *)"Grab int input");
  }
  else if (strcmp(currnode->attr.name, (char *)"inputb")==0) {
    emitRO((char *)"INB", RT, RT, RT, (char *)"Grab bool input");
  }
  else if (strcmp(currnode->attr.name, (char *)"inputc")==0) {
    emitRO((char *)"INC", RT, RT, RT, (char *)"Grab char input");
  }
  else if (strcmp(currnode->attr.name, (char *)"output")==0) {
    emitRM((char *)"LD", AC, -2, FP, (char *)"Load parameter");
    emitRO((char *)"OUT", AC, AC, AC, (char *)"Output integer");
  }
  else if (strcmp(currnode->attr.name, (char *)"outputb")==0) {
    emitRM((char *)"LD", AC, -2, FP, (char *)"Load parameter");
    emitRO((char *)"OUTB", AC, AC, AC, (char *)"Output bool");
  }
  else if (strcmp(currnode->attr.name, (char *)"outputc")==0) {
    emitRM((char *)"LD", AC, -2, FP, (char *)"Load parameter");
    emitRO((char *)"OUTC", AC, AC, AC, (char *)"Output char");
  }
  else if (strcmp(currnode->attr.name, (char *)"outnl")==0) {
    emitRO((char *)"OUTNL", AC, AC, AC, (char *)"Output a newline");
  }
  else {
    emitComment((char *)"ERROR(LINKER): No support for special function");
    emitComment(currnode->attr.name);
  }
```

```
10:   LD  1,0(1)   Adjust fp
11:   JMP  7,0(3)   Return
* END FUNCTION output
*
* ** ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION inputb
12:   ST  3,-1(1)   Store return address
13:   INB  2,2,2    Grab bool input
14:   LD  3,-1(1)   Load return address
15:   LD  1,0(1)   Adjust fp
16:   JMP  7,0(3)   Return
* END FUNCTION inputb
*
* ** ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION outputb
17:   ST  3,-1(1)   Store return address
18:   LD  3,-2(1)   Load parameter
19:   OUTB  3,3,3    Output bool
20:   LD  3,-1(1)   Load return address
21:   LD  1,0(1)   Adjust fp
22:   JMP  7,0(3)   Return
* END FUNCTION outputb
*
* ** ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION inputc
23:   ST  3,-1(1)   Store return address
24:   INC  2,2,2    Grab char input
25:   LD  3,-1(1)   Load return address
26:   LD  1,0(1)   Adjust fp
27:   JMP  7,0(3)   Return
* END FUNCTION inputc
*
* ** ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION outputc
28:   ST  3,-1(1)   Store return address
29:   LD  3,-2(1)   Load parameter
30:   OUTC  3,3,3    Output char
31:   LD  3,-1(1)   Load return address
32:   LD  1,0(1)   Adjust fp
33:   JMP  7,0(3)   Return
* END FUNCTION outputc
*

* ** ** ** ** ** ** ** ** ** ** ** ** **

* FUNCTION outnl
 34:     ST  3,-1(1)   Store return address
 35:  OUTNL  3,3,3     Output a newline
 36:     LD  3,-1(1)   Load return address
 37:     LD  1,0(1)    Adjust fp
 38:    JMP  7,0(3)    Return
* END FUNCTION outnl
```



Reverse Engineering

See tmDescription.pdf on Canvas

```
int emitSkip(int howMany);    // emitSkip(0) tells you where the next instruction will be placed

void emitComment(char *c);
void emitComment(char *c, char *cc);
void emitComment(char *c, int n);

void emitGoto(int d, long long int s, char *c);
void emitGoto(int d, long long int s, char *c, char *cc);
void emitGotoAbs(int a, char *c);
void emitGotoAbs(int a, char *c, char *cc);

void emitRM(char *op, long long int r, long long int d, long long int s, char *c);
void emitRM(char *op, long long int r, long long int d, long long int s, char *c, char *cc);

void emitRO(char *op, long long int r, long long int s, long long int t, char *c);
void emitRO(char *op, long long int r, long long int s, long long int t, char *c, char *cc);

void backPatchAJumpToHere(int addr, char *comment);
void backPatchAJumpToHere(char *cmd, int reg, int addr, char *comment);

int emitStrLit(int goffset, char *s); // for char arrays
```

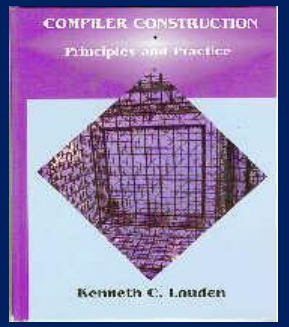
Comments stat with a *

emitcode.h

See tmDescription for when to use each of these.

LITERAL INSTRUCTIONS

emitStrLit - Only used once in String Constants



LIT 666 load into data memory the single "word" value given at the address.

LIT 'x' load into data memory the single "word" value given at the address.

LIT "stuff" load into data memory the string starting with the first character at the address given and then *decrementing* from there. The size is then stored in the address+1.

codegen.cpp

```
case ConstantK:
case Char:
        if (currnode->isArray) {
           emitStrLit(currnode->offset, currnode->attr.string);
           emitRM((char *)"LDA", AC, currnode->offset, 0, (char *)"Load address of char array");
        }
        else {
           emitRM((char *)"LDC", AC, int(currnode->attr.cvalue), 6, (char *)"Load char constant");
        }
        break;
```

"word" value given at the address.

LIT "stuff" load into data memory the string starting with the first character at the address given and then *decrementing* from there. The size is then stored in the address+1.

file.bC

```
main() {
  "myString";
}
```

file.tm

```
* FUNCTION main
* TOFF set: -2
 39:     ST  3,-1(1) Store return address
* COMPOUND
* TOFF set: -2
* Compound Body
* EXPRESSION
  1:    LIT  "myString"
 40:    LDA  3,-1(0) Load address of char array
* TOFF set: -2
* END COMPOUND
```

LITERAL INSTRUCTIONS
emitStrLit - Only used once in String Constants

```
main() {
  char a[8]:"myString";
}
```

LIT 666 load into data memory the single "word" value given at the address.

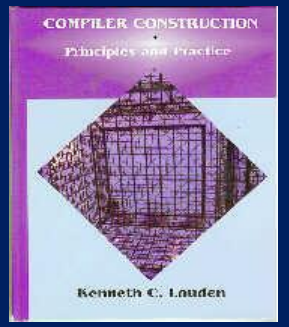
LIT 'x' load into data memory the single "word" value given at the address.

LIT "stuff" load into data memory the string starting with the first character at the address given and then *decrementing* from there. The size is then stored in the address+1.

```
* FUNCTION main
* TOFF set: -2
 39:     ST  3,-1(1) Store return address
* COMPOUND
* TOFF set: -11
 40:    LDC  3,8(6)  load size of array a
 41:     ST  3,-2(1) save size of array a
  1:    LIT  "myString"
 42:    LDA  3,-1(0) Load address of char array
 43:    LDA  4,-3(1) address of lhs
 44:     LD  5,1(3)  size of rhs
 45:     LD  6,1(4)  size of lhs
 46:    SWP  5,6,6   pick smallest size
 47:    MOV  4,3,5   array op =
* Compound Body
```

Why pick the smallest size?

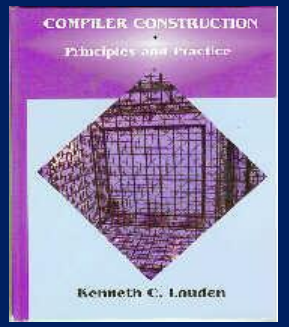
REGISTER ONLY INSTRUCTIONS emitRO



HALT X, X, X stop execution (all registers ignored)
NOP X, X, X does nothing but take space (all registers ignored)
IN r, X, X reg[r] <- input integer value of register r from stdin
INB r, X, X reg[r] <- input boolean value of register r from stdin
INC r, X, X reg[r] <- input char value of register r from stdin
OUT r, X, X reg[r] -> output integer value of register r to stdout
OUTB r, X, X reg[r] -> output boolean value of register r to stdout
OUTC r, X, X reg[r] -> output char value of register r to stdout
OUTNL X, X, X output a newline to stdout

ADD r, s, t reg[r] = reg[s] + reg[t]
SUB r, s, t reg[r] = reg[s] - reg[t]
MUL r, s, t reg[r] = reg[s] * reg[t]
DIV r, s, t reg[r] = reg[s] / reg[t] (only a truncating integer divide)
MOD r, s, t reg[r] = reg[s] % reg[t] (always returns the NONNEGATIVE modulus of reg[s] % reg[t])
AND r, s, t reg[r] = reg[s] & reg[t] (bitwise and)
OR r, s, t reg[r] = reg[s] | reg[t] (bitwise or)
XOR r, s, t reg[r] = reg[s] ^ reg[t] (bitwise xor)
NOT r, s, X reg[r] = ~ reg[s] (bitwise complement)
NEG r, s, X reg[r] = - reg[s] negative
SWP r, s, X reg[r] = min(reg[r], reg[s]), reg[s] = max(reg[r], reg[s]) (useful for min or max)
RND r, s, X reg[r] = random(0, |reg[s]-1|) (get random num between 0 and |reg[s]-1| inclusive; X ignored,)

REGISTER TO MEMORY INSTRUCTIONS emitRM



LDC r, c(X) reg[r] = c (load constant; immediate; X ignored)

LDA r, d(s) reg[r] = d + reg[s] (load direct address)

LD r, d(s) reg[r] = dMem[d + reg[s]] (load indirect)

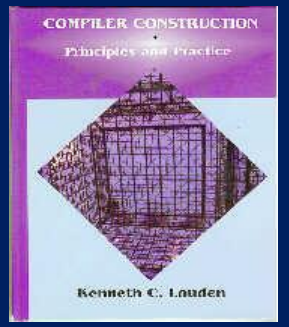
ST r, d(s) dMem[d + reg[s]] = reg[r]

JNZ r, d(s) if reg[r]!=0 reg[PC] = d + reg[s] (jump nonzero)

JZR r, d(s) if reg[r]==0 reg[PC] = d + reg[s] (jump zero)

JMP x, d(s) reg[PC] = d + reg[s] (jump)

TEST INSTRUCTIONS
emitRO



TLT r, s, t if reg[s]<reg[t] reg[r] = 1 else reg[r] = 0
TLE r, s, t if reg[s]<=reg[t] reg[r] = 1 else reg[r] = 0
TEQ r, s, t if reg[s]==reg[t] reg[r] = 1 else reg[r] = 0
TNE r, s, t if reg[s]!=reg[t] reg[r] = 1 else reg[r] = 0
TGE r, s, t if reg[s]>=reg[t] reg[r] = 1 else reg[r] = 0
TGT r, s, t if reg[s]>reg[t] reg[r] = 1 else reg[r] = 0
SLT r, s, t if (reg[r]>=0) reg[r] = (reg[s]<reg[t] ? 1 : 0);
  else reg[r] = (-reg[s] < -reg[t] ? 1 : 0);
SGT r, s, t if (reg[r]>=0) reg[r] = (reg[s]>reg[t] ? 1 : 0);
  else reg[r] = (-reg[s] > -reg[t] ? 1 : 0);

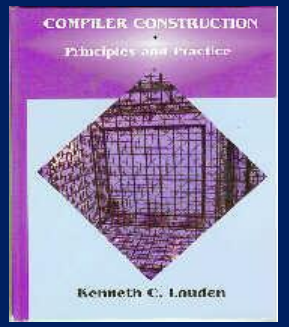
file.bC

```
main() {
 if (2<4) then {

  }
}
```

file.tm

```
* FUNCTION main
* TOFF set: -2
 39:     ST  3,-1(1)    Store return address
* COMPOUND
* TOFF set: -2
* Compound Body
* IF
 40:    LDC  3,2(6)     Load integer constant
 41:     ST  3,-2(1)    Push left side
* TOFF dec: -3
 42:    LDC  3,4(6)     Load integer constant
* TOFF inc: -2
 43:     LD  4,-2(1)    Pop left into ac1
 44:    TLT  3,4,3      Op <
* THEN
* COMPOUND
* TOFF set: -2
* Compound Body
* TOFF set: -2
* END COMPOUND
 45:    JZR  3,0(7)     Jump around the THEN if false [backpatch]
* END IF
* TOFF set: -2
* END COMPOUND
```

BLOCK MEMORY TO MEMORY INSTRUCTIONS emitRO



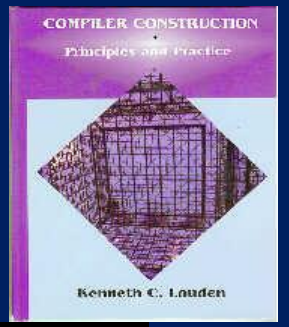
MOV r, s, t dMem[reg[r] - (0..reg[t]-1)] = dMem[reg[s] - (0..reg[t]-1)]
(overlapping source and target is undefined)

SET r, s, t dMem[reg[r] - (0..reg[t]-1)] = reg[s] makes reg[t] copies of reg[s]

CO r, s, t reg[5] = dMem[reg[r] + k]
(for the first k that yields a diff or the last tested if no diff)
reg[6] = dMem[reg[s] + k]
(for the first k that yields a diff or the last tested if no diff)
WARNING: memory is scanned from higher addresses to lower

COA r, s, t reg[5] = reg[r] + k
(for the first k that yields a diff at that address or the last tested if no diff)
reg[6] = reg[s] + k
(for the first k that yields a diff at that address or the last tested if no diff)
WARNING: memory is scanned from higher addresses to lower

void codegenDecl(TreeNode *currnode)




```
// given the syntax tree for declarations generate the code
void codegenDecl(TreeNode *currnode)
{
   commentLineNum(currnode);

   switch(currnode->kind.decl) {
   case VarK:
      // You have a LOT to do here!!!!!
      break;
   case FuncK:
      if (currnode->lineno == -1) {          // These are the library functions we just added
         codegenLibraryFun(currnode);
      }
      else {
         codegenFun(currnode);
      }
      break;
   case ParamK:
      // IMPORTANT: no instructions need to be allocated for parameters here
      break;
   }
}
```

void codegenFun(TreeNode *currnode)

```
*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION main
* TOFF set: -2
 39:     ST  3,-1(1)    Store return address
* COMPOUND
* TOFF set: -2
* Compound Body
* TOFF set: -2
* END COMPOUND
* Add standard closing in case there is no return statement
 40:    LDC  2,0(6)     Set return value to 0
 41:     LD  3,-1(1)    Load return address
 42:     LD  1,0(1)     Adjust fp
 43:    JMP  7,0(3)     Return
* END FUNCTION main
```

```
// process functions
void codegenFun(TreeNode *currnode)
{
    emitComment((char *)"");
    emitComment((char *)"** ** ** ** ** ** ** ** ** ** ** **");
    emitComment((char *)"FUNCTION", currnode->attr.name);
    toffset = currnode->size;          // recover the end of activation record
    emitComment((char *)"TOFF set:", toffset);
```

void codegenFun(TreeNode *currnode)

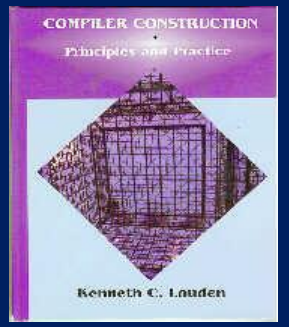
```
*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION main
* TOFF set: -2
 39:     ST  3,-1(1)    Store return address
* COMPOUND
* TOFF set: -2
* Compound Body
* TOFF set: -2
* END COMPOUND
* Add standard closing in case there is no return statement
 40:    LDC  2,0(6)     Set return value to 0
 41:     LD  3,-1(1)    Load return address
 42:     LD  1,0(1)     Adjust fp
 43:    JMP  7,0(3)     Return
* END FUNCTION main
```

```
// IMPORTANT: For function nodes the offset is defined to be the position of the
// function in the code space!  This is accessible via the symbol table.
// remember where this function is:
currnode->offset = emitSkip(0);    // offset holds the instruction address!!

// Store return address
emitRM((char *)"ST", AC, RETURNOFFSET, FP, (char *)"Store return address");
```

void codegenFun(TreeNode *currnode)

```
*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION main
* TOFF set: -2
 39:     ST  3,-1(1)    Store return address
* COMPOUND
* TOFF set: -2
* Compound Body
* TOFF set: -2
* END COMPOUND
* Add standard closing in case there is no return statement
 40:    LDC  2,0(6)     Set return value to 0
 41:     LD  3,-1(1)    Load return address
 42:     LD  1,0(1)     Adjust fp
 43:    JMP  7,0(3)     Return
* END FUNCTION main
```

```
    // Generate code for the statements...
    codegenGeneral(currnode->child[1]);

    // In case there was no return statement
    // set return register to 0 and return
    emitComment((char *)"Add standard closing in case there is no return statement");
    emitRM((char *)"LDC", RT, 0, 6, (char *)"Set return value to 0");
    emitRM((char *)"LD", AC, RETURNOFFSET, FP, (char *)"Load return address");
    emitRM((char *)"LD", FP, OFPOFF, FP, (char *)"Adjust fp");
    emitGoto(0, AC, (char *)"Return");

    emitComment((char *)"END FUNCTION", currnode->attr.name);
}
```



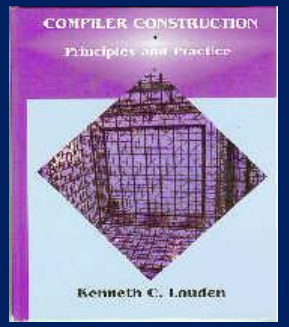
- Reverse Engineering - See tmDescription

```
int emitSkip(int howMany);    // emitSkip(0) tells you where the next instruction will be placed

void emitComment(char *c);
void emitComment(char *c, char *cc);
void emitComment(char *c, int n);

void emitGoto(int d, long long int s, char *c);
void emitGoto(int d, long long int s, char *c, char *cc);
void emitGotoAbs(int a, char *c);
void emitGotoAbs(int a, char *c, char *cc);

void emitRM(char *op, long long int r, long long int d, long long int s, char *c);
void emitRM(char *op, long long int r, long long int d, long long int s, char *c, char *cc);

void emitRO(char *op, long long int r, long long int s, long long int t, char *c);
void emitRO(char *op, long long int r, long long int s, long long int t, char *c, char *cc);

void backPatchAJumpToHere(int addr, char *comment);
void backPatchAJumpToHere(char *cmd, int reg, int addr, char *comment);

int emitStrLit(int goffset, char *s); // for char arrays
```

emitcode.h

See tmDescription for when to use each of these.

emitSkip



- emitSkip(int n)

  skips n code locations for later backpatch.
  It also returns the current code position.

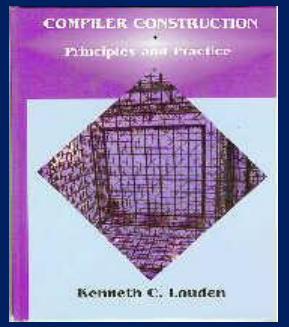
- emitSkip(0)

  Tells you where the next instruction will go
  Tells you where you are and reserves no space.
  (Could also use emitWhereAmI())

- backPatchAJumpToHere(int addr, char *comment)

```
int currloc;
currloc = emitWhereAmI();          // remember where we are
emitNewLoc(addr);                  // go to addr
emitGotoAbs(currloc, comment);     // the LDA to here
emitNewLoc(currloc);               // restore addr
```

emitSkip/backPatchAJumpToHere



- IfK
- WhileK
- ForK

- FuncK

```
void codegenFun(TreeNode *currnode)
{
    emitComment((char *)"");
    emitComment((char *)"** ** ** ** ** ** ** ** ** ** ** **");
    emitComment((char *)"FUNCTION", currnode->attr.name);
    toffset = currnode->size;          // recover the end of activation record
    emitComment((char *)"TOFF set:", toffset);

    currnode->offset = emitSkip(0);    // offset holds the instruction addr

    // Store return address
    emitRM((char *)"ST", AC, RETURNOFFSET, FP, (char *)"Store return a

    // Generate code for the statements...
    codegenGeneral(currnode->child[1]);




}
```

```
*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION main
* TOFF set: -2
 39:     ST  3,-1(1)    Store return address
```

```
void codegenStatement(TreeNode *currnode)
{
 // local state to remember stuff
   int skiploc=0, skiploc2=0, currloc=0;  // some temporary instuction addresse
   TreeNode *loopindex=NULL;              // a pointer to the index variable declar

   commentLineNum(currnode);

   switch (currnode->kind.stmt) {
////////////////Other cases
 case CompoundK:
     {
       int savedToffset;

       savedToffset = toffset;
       toffset = currnode->size;            // recover the end of activation record
       emitComment((char *)"COMPOUND");
       emitComment((char *)"TOFF set:", toffset);
       codegenGeneral(currnode->child[0]);     // process inits
       emitComment((char *)"Compound Body");
       codegenGeneral(currnode->child[1]);     // process body
       toffset = savedToffset;
       emitComment((char *)"TOFF set:", toffset);
       emitComment((char *)"END COMPOUND");
     }
     break;
   default:
     break;
   }
}
```

```
*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION main
* TOFF set: -2
 39:     ST  3,-1(1)    Store return address
* COMPOUND
* TOFF set: -2
* Compound Body
* TOFF set: -2
* END COMPOUND
```

```
void codegenFun(TreeNode *currnode)
{
    emitComment((char *)"");
    emitComment((char *)"** ** ** ** ** ** ** ** ** ** ** **");
    emitComment((char *)"FUNCTION", currnode->attr.name);
    toffset = currnode->size;          // recover the end of activation record
    emitComment((char *)"TOFF set:", toffset);

    currnode->offset = emitSkip(0);    // offset holds the instruction addr

    // Store return address
    emitRM((char *)"ST", AC, RETURNOFFSET, FP, (char *)"Store return a

    // Generate code for the statements...
    codegenGeneral(currnode->child[1]);


    // In case there was no return statement
    // set return register to 0 and return
    emitComment((char *)"Add standard closing in case there is no return statement");
    emitRM((char *)"LDC", RT, 0, 6, (char *)"Set return value to 0");
    emitRM((char *)"LD", AC, RETURNOFFSET, FP, (char *)"Load return address");
    emitRM((char *)"LD", FP, OFPOFF, FP, (char *)"Adjust fp");
    emitGoto(0, AC, (char *)"Return");

    emitComment((char *)"END FUNCTION", currnode->attr.name);
}
```

```
*
* ** ** ** ** ** ** ** ** ** ** ** **
* FUNCTION main
* TOFF set: -2
 39:     ST  3,-1(1)    Store return address
* COMPOUND
* TOFF set: -2
* Compound Body
* TOFF set: -2
* END COMPOUND
* Add standard closing in case there is no
 40:    LDC  2,0(6)     Set return value to 0
 41:     LD  3,-1(1)    Load return address
 42:     LD  1,0(1)     Adjust fp
 43:    JMP  7,0(3)     Return
* END FUNCTION main
```

emitSkip/backPatchAJumpToHere
case WhileK:

```
emitComment((char *)"WHILE");
currloc = emitSkip(0);                  // return to here to do the test
codegenExpression(currnode->child[0]);  // test expression

emitRM((char *)"JNZ", AC, 1, PC, (char *)"Jump to while part");
emitComment((char *)"DO");

skiploc = breakloc;             // save the old break statement return point
breakloc = emitSkip(1);         // addr of instr that jumps to end of loop
                                // this is also the backpatch point
codegenGeneral(currnode->child[1]);     // do body of loop
emitGotoAbs(currloc, (char *)"go to beginning of loop");
backPatchAJumpToHere(breakloc, (char *)"Jump past loop [backpatch]");
                                        // backpatch jump to end of loop

breakloc = skiploc;             // restore for break statement
emitComment((char *)"END WHILE");
break;
```

The 1 would have been bigger if the {} was not empty. As it is we skip the one instruction that jumps us back

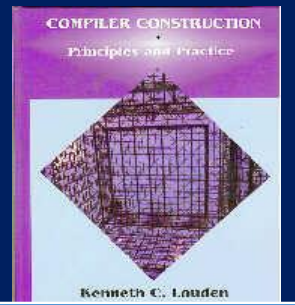
currloc

breakloc

```
* WHILE
40:   LDC  3,1(6)    Load Boolean constant
41:   JNZ  3,1(7)    Jump to while part
* DO
42:   JMP  7,1(7)    Jump past loop [backpatch]
* COMPOUND
* TOFF set: -2
* Compound Body
* TOFF set: -2
* END COMPOUND
43:   JMP  7,-4(7)   go to beginning of loop
* END WHILE
```

Where did the -4 come from?
emitLoc (the address of the next instruction) is 44.
currlock is 40
currlock – emitLoc = -4

emitSkip/backPatchAJumpToHere case WhileK:




```
* WHILE
40:  LDC  3,1(6)   Load Boolean constant
41:  JNZ  3,1(7)   Jump to while part
* DO
42:  JMP  7,1(7)   Jump past loop [backpatch]
* COMPOUND
* TOFF set: -2
* Compound Body
* TOFF set: -2
* END COMPOUND
43:  JMP  7,-4(7)  go to beginning of loop
* END WHILE
```

PC (Reg 7) → 40: LDC 3,1(6) Load Boolean constant

The child will load 1 (true) or 0 (false) into register 3 (AC)

PC (Reg 7) → 41: JNZ 3,1(7) Jump to while part

Skip next instruction if Reg 3 != 0 (Reg3 = 1 (true). Skip)

(jump nonzero)
if reg[r]!=0 reg[PC] = d + reg[s]
if reg[AC]!=0 reg[PC] = 1 + reg[PC]

PC (Reg 7) → * COMPOUND … * END COMPOUND

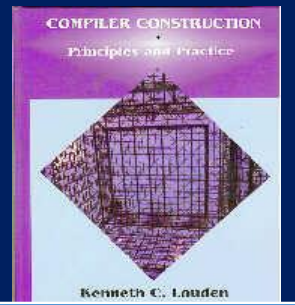
Do whatever is in the {}

PC (Reg 7) → 43: JMP 7,-4(7) go to beginning of loop

Jump back 4 instucitons

(jump)
reg[PC] = d + reg[s]
reg[PC] = -4 + reg[PC]

emitSkip/backPatchAJumpToHere case WhileK:




```
* WHILE
40:  LDC  3,1(6)   Load Boolean constant
41:  JNZ  3,1(7)   Jump to while part
* DO
42:  JMP  7,1(7)   Jump past loop [backpatch]
* COMPOUND
* TOFF set: -2
* Compound Body
* TOFF set: -2
* END COMPOUND
43:  JMP  7,-4(7)  go to beginning of loop
* END WHILE
```

PC (Reg 7) → 40: LDC 3,1(6) Load Boolean constant

The child will load 1 (true) or 0 (false) into register 3 (AC)

PC (Reg 7) → 41: JNZ 3,1(7) Jump to while part

Skip next instruction if Reg 3 != 0 (Reg3 = 1 (true). Skip)

(jump nonzero)
if reg[r]!=0 reg[PC] = d + reg[s]
if reg[AC]!=0 reg[PC] = 1 + reg[PC]

PC (Reg 7) → * COMPOUND … * END COMPOUND

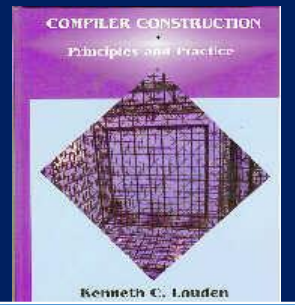
Do whatever is in the {}

PC (Reg 7) → 43: JMP 7,-4(7) go to beginning of loop

Jump back 4 instucitons

(jump)
reg[PC] = d + reg[s]
reg[PC] = -4 + reg[PC]

emitSkip/backPatchAJumpToHere case WhileK:




```
* WHILE
40:    LDC  3,1(6)     Load Boolean constant
41:    JNZ  3,1(7)     Jump to while part
* DO
42:    JMP  7,1(7)     Jump past loop [backpatch]
* COMPOUND
* TOFF set: -2
* Compound Body
* TOFF set: -2
* END COMPOUND
43:    JMP  7,-4(7)    go to beginning of loop
* END WHILE
```

PC (Reg 7)

PC (Reg 7)

PC (Reg 7)

PC (Reg 7)

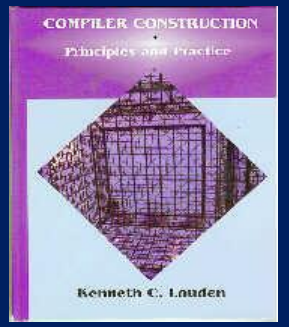
The child will load 1 (true) or 0 (false) into register 3 (AC)

Skip next instruction if Reg 3 != 0 (Reg3 = 0 (false). No skip)

Jump ahead 1 instruction

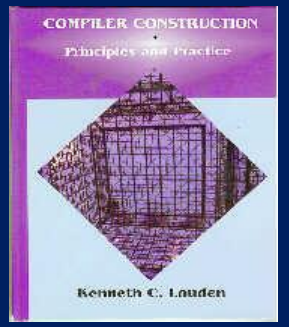
(jump)
reg[PC] = d + reg[s]
reg[PC] = 1 + reg[PC]

Helper function for IdK, AssignK & VarK



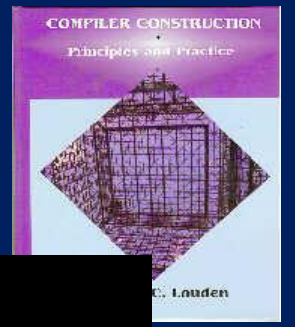

```
int offsetRegister(VarKind v) {
   switch (v) {
   case Local:       return FP;
   case Parameter:   return FP;
   case Global:      return GP;
   case LocalStatic: return GP;
   default:
      printf((char *)"ERROR(codegen): looking up offset register for a variable of type %d\n", v);
      return 666;
   }
}
```

AssignK




```
AssignK
 if (lhs->attr.op == '[‘) {
   // stuff
 }
 else{
    int offReg;
    offReg = offsetRegister(lhs->varKind);
    // Lots of cases that use it. Here is a sample:
    case ADDASS:
           emitRM((char *)"LD", AC1, lhs->offset, offReg,
                 (char *)"load lhs variable", lhs->attr.name);
           emitRO((char *)"ADD", AC, AC1, AC, (char *)"op +=");
           emitRM((char *)"ST", AC, lhs->offset, offReg,
                 (char *)"Store variable", lhs->attr.name);
           break;
 }
```

toffset // next available temporary space

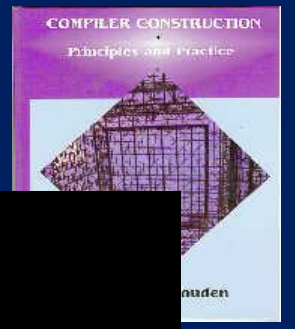


- OpK:

```
if (currnode->child[1]) {
    emitRM((char *)"ST", AC, toffset, FP, (char *)"Push left side");
    toffset--;  emitComment((char *)"TOFF dec:", toffset);
    codegenExpression(currnode->child[1]);
    toffset++;  emitComment((char *)"TOFF inc:", toffset);
    emitRM((char *)"LD", AC1, toffset, FP, (char *)"Pop left into ac1");
  }
  // More code here
```

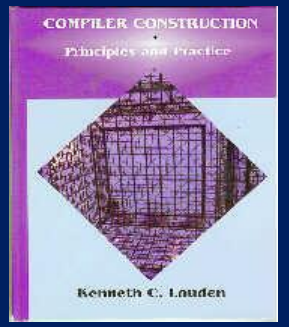
- AssignK:
- CallK:
- ForK:
- ConmoundK:
- FuncK:

Table of contents



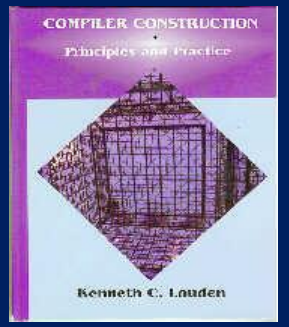

```
// this is the top level code generator call
void codegen(FILE *codeIn,                  // where the code should be written
             char *srcFile,                 // name of file compiled
             TreeNode *syntaxTree,    // tree to process
             SymbolTable *globalsIn,        // globals so function info can be found
             int globalOffset,
             bool linenumFlagIn)
{
   int initJump;

   code = codeIn;
   globals = globalsIn;
   linenumFlag = linenumFlagIn;
   breakloc = 0;

   initJump = emitSkip(1);                  // save a place for the jump to init
   codegenHeader(srcFile);                  // nice comments describing what is compiled
   codegenGeneral(syntaxTree);                      // general code generation including I/O library
   codegenInit(initJump, globalOffset);             // generation of initialization for run
}
```

codegenInit(initJump, globalOffset);




```
// Generate init code ...
void codegenInit(int initJump, int globalOffset)
{
   backPatchAJumpToHere(initJump, (char *)"Jump to init [backpatch]");

   emitComment((char *)"INIT");
//OLD pre 4.6 TM    emitRM((char *)"LD",  GP, 0, 0, (char *)"Set the global pointer");
   emitRM((char *)"LDA", FP, globalOffset, GP, (char *)"set first frame at end of globals");
   emitRM((char *)"ST",  FP, 0, FP, (char *)"store old fp (point to self)");

   initGlobalArraySizes();

   emitRM((char *)"LDA", AC, 1, PC, (char *)"Return address in ac");

   {  // jump to main
      TreeNode *funcNode;

      funcNode = (TreeNode *)(globals->lookup((char *)"main"));
      if (funcNode) {
         emitGotoAbs(funcNode->offset, (char *)"Jump to main");
      }
      else {
         printf((char *)"ERROR(LINKER): Procedure main is not defined.\n");
         numErrors++;
      }
   }

   emitRO((char *)"HALT", 0, 0, 0, (char *)"DONE!");
   emitComment((char *)"END INIT");
}
```

```
0:   JMP  7,43(7)   Jump to init [backpatch]
* INIT
 44:   LDA  1,0(0)    set first frame at end of globals
 45:    ST  1,0(1)    store old fp (point to self)
* INIT GLOBALS AND STATICS
* END INIT GLOBALS AND STATICS
 46:   LDA  3,1(7)    Return address in ac
 47:   JMP  7,-9(7)   Jump to main
 48:  HALT  0,0,0     DONE!
* END INIT
```

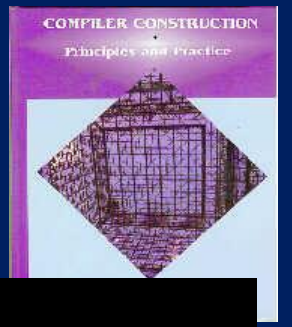



```
void initGlobalArraySizes()
{
   emitComment((char *)"INIT GLOBALS AND STATICS");
   globals->applyToAllGlobal(initAGlobalSymbol);
   emitComment((char *)"END INIT GLOBALS AND STATICS");
}
```

```
0:   JMP  7,43(7)   Jump to init [backpatch]
* INIT
 44:   LDA  1,0(0)    set first frame at end of globals
 45:    ST  1,0(1)    store old fp (point to self)
* INIT GLOBALS AND STATICS
* END INIT GLOBALS AND STATICS
 46:   LDA  3,1(7)    Return address in ac
 47:   JMP  7,-9(7)   Jump to main
 48:  HALT  0,0,0     DONE!
* END INIT
```

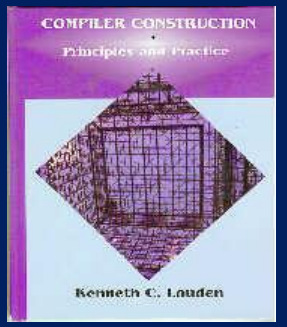
```
void initAGlobalSymbol(std::string sym, void *ptr)
{
   TreeNode *currnode;

//    printf("Symbol: %s\n", sym.c_str());      // dump the symbol table
   currnode = (TreeNode *)ptr;
//    printf("lineno: %d\n", currnode->lineno);      // dump the symbol table
   if (currnode->lineno != -1) {
      if (currnode->isArray) {
         emitRM((char *)"LDC", AC, currnode->size-1, 6, (char *)"load size of array", currnode->attr.name);
         emitRM((char *)"ST", AC, currnode->offset+1, GP, (char *)"save size of array", currnode->attr.name);
      }

      if (currnode->kind.decl==VarK &&
         (currnode->varKind == Global || currnode->varKind == LocalStatic)) {
         if (currnode->child[0]) {
            // compute rhs -> AC;
            codegenExpression(currnode->child[0]);

            // save it
            emitRM((char *)"ST", AC, currnode->offset, GP,
                  (char *)"Store variable", currnode->attr.name);
         }
      }
   }

}
```

void codegenDecl(TreeNode *currnode) case VarK:




```
case VarK:
if (currnode->isArray) {
        switch (currnode->varKind) {
        case Local:
           emitRM((char *)"LDC", AC, currnode->size-1, 6, (char *)"load size of array", currnode->attr.name);
           emitRM((char *)"ST", AC, currnode->offset+1, offsetRegister(currnode->varKind),
                        (char *)"save size of array", currnode->attr.name);
           break;
        case LocalStatic:
        case Parameter:
        case Global:
           // do nothing here
           break;
        case None:
           // Error Condition
        }
        // ARRAY VALUE initialization
        if (currnode->child[0]) {
           codegenExpression(currnode->child[0]);
           emitRM((char *)"LDA", AC1, currnode->offset, offsetRegister(currnode->varKind), (char *)"address of lhs");
           emitRM((char *)"LD", AC2, 1, AC, (char *)"size of rhs");
           emitRM((char *)"LD", AC3, 1, AC1, (char *)"size of lhs");
           emitRO((char *)"SWP", AC2, AC3, 6, (char *)"pick smallest size");
           emitRO((char *)"MOV", AC1, AC, AC2, (char *)"array op =");
        }
    }
```

void codegenDecl(TreeNode *currnode) case VarK:




```
else { // !currnode->isArray
        // SCALAR VALUE initialization
        if (currnode->child[0]) {
            switch (currnode->varKind) {
            case Local:
               // compute rhs -> AC;
               codegenExpression(currnode->child[0]);

               // save it
               emitRM((char *)"ST", AC, currnode->offset, FP,  (char *)"Store variable", currnode->attr.name);
            case LocalStatic:
            case Parameter:
            case Global:
               // do nothing here
               break;
            case None:
               ///Error condition!!!
            }
        }
    }
    break;
```