

Appendix B

MiniC Compiler

B.1 Software Files

The compiler for MiniC shown in this appendix is implemented using lex for the lexical analysis and yacc for the syntax analysis. The syntax phase puts out a file of atoms, which forms the input to the code generator, written as a separate C program. The code generator puts out hex characters to `stdout` (the standard output file), one instruction per line. This output can be displayed on the monitor, stored in a file, or piped into the Mini simulator and executed.

This software is available in source code from the author via the Internet. The file names included in this package are, at the time of this printing:

<code>mk</code>	Shell script to compile software using make utility
<code>makefile</code>	Input to make utility
<code>MiniC.l</code>	Input to lex
<code>MiniC.y</code>	Input to yacc
<code>MiniC.c</code>	Recursive descent parser
<code>gen.c</code>	Code generator
<code>mini.c</code>	Target machine simulator
<code>mini.h</code>	Header file for simulator
<code>MiniC.h</code>	Header file for compiler
<code>cos.c</code>	MiniC program to compute the cosine function
<code>bisect.c</code>	MiniC program to compute the square root of two by bisection
<code>fact.c</code>	MiniC program to compute the factorial function
<code>locate</code>	Shell script to find file containing specified source code
<code>readme</code>	Updated information regarding the software

The source files are available at:

<http://www.rowan.edu/~bergmann/books/minic>

The unix/linux make command can be used to generate executables.

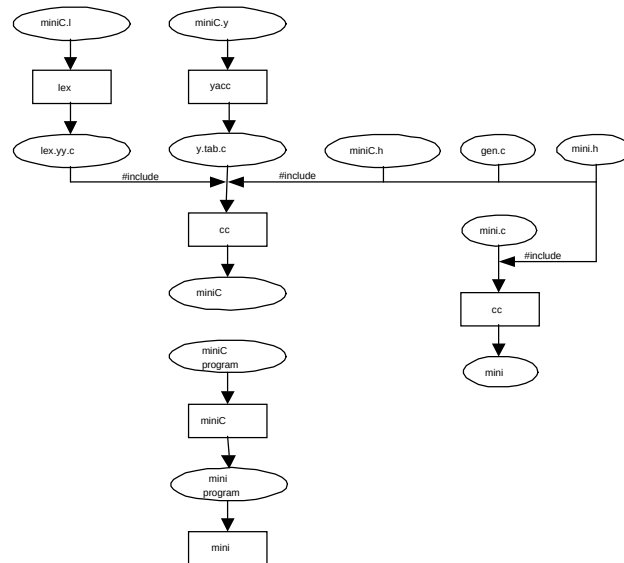


Figure B.1 Flow Diagram to Generate the Compiler and Simulator

After copying these files to a new directory, simply invoke the `make` command to compile and link the software. If you make any changes to the source code, again invoke `make` to compile only those modules which need to be compiled.

To compile the cosine program and view the output on the monitor, use the command:

```
$ MiniC <cos.c
```

To compile the cosine program and store the output in a file named `COS.mini`, use the command:

```
$ MiniC <cos.c >cos.mini
```

To run this program on the simulator, first compile the mini simulator:

```
$ cc mini.c -o mini
```

then run the cosine program using the command:

```
$ mini <cos.mini
```

To compile the cosine program, and execute it on the simulator with one command:

```
$ MiniC <cos.c | mini
```

A flow graph indicating the relationships of these files is shown in Figure B.1 in which input and output file names are shown in ovals and executing programs are shown in rectangles.

B.2 Lexical Phase

The lexical analysis phase is implemented with the Unix `lex` utility. The input to `lex` is the source file `MiniC.l`, and the output is the file `lex.yy.c`, which includes the `yylex()` function. This function reads MiniC source code from `stdin` (the standard input file) and returns tokens to the parser. The tokens are listed in the `MiniC.y` file (though single character tokens are not listed there). In addition to key words, the lexical phase finds comparison tokens, the assignment token, comments, numeric constants, and identifiers. White space and newline characters are taken as token delimiters, but ignored otherwise.

The function `searchIdent()` is used to install and look up identifiers in a hash table, which forms the symbol table for this compiler. The `searchIdent()` function checks for the use of undeclared identifiers and multiply-declared identifiers and puts out an appropriate error message.

The function `searchNums()` is used to install numeric constants into a binary search tree so that constants used more than once in the source program are stored only once in the run-time program. The value of the constant is stored as a floating-point number in the target machine's memory.

The function `alloc(size)` allocates the specified number of words of memory for use at run time. It is used to allocate space on the target machine for declared identifiers.

As explained in Appendix B.3 below, the parser stack may store three different types of data: a target machine address (`address`), an integer code (`code`), and a structure of three label numbers (`labels`). Consequently in our lex source file when we assign a value to the global variable `yylval`, we must indicate which of the three types is being returned, because it will be pushed onto the parser stack. For example, when a numeric constant is found, we assign its location in the target machine memory to `yylval`: `yylval.address = searchNums();`.

The source code forming the input to lex, taken from the file `Minic.l`, is shown below:

```

INT    [0-9]+
EXP    ([eE][+-]?{INT})
NUM    ({INT}\.?.{INT}?|\.{INT}){EXP}?
%{
#include <stdlib.h>
ADDRESS searchIdent(void);
ADDRESS searchNums(void);
ADDRESS alloc(int size);
%}
%start COMMENT1 COMMENT2
%%
<COMMENT1>.+      ;
<COMMENT1>\n      BEGIN 0;          /* end comment */
<COMMENT2>[^\*]+  ;
<COMMENT2>\*[^/]  ;
<COMMENT2>\*\/    BEGIN 0;          /* end comment */
"//"              BEGIN COMMENT1;
"/*"              BEGIN COMMENT2;
int                {yylval.code = 2; return INT;}
float              {yylval.code = 3; return FLOAT;}
for                return FOR;
while              return WHILE;
if                 return IF;
else               return ELSE;
"=="               {yylval.code = 1; return COMPARISON;}
\<                {yylval.code = 2; return COMPARISON;}
>                  {yylval.code = 3; return COMPARISON;}
"<="               {yylval.code = 4; return COMPARISON;}

```

```

">="      {yylval.code = 5; return COMPARISON;}
"!="      {yylval.code = 6; return COMPARISON;}
[a-zA-Z][a-zA-Z0-9_]* {yylval.address = searchIdent();
                        return IDENTIFIER;}
{NUM}     {yylval.address = searchNums(); return NUM;}
[ \t]     ; /* white space */
\n        lineno++;      /* free format */
.         return yytext[0]; /* any other char */
%%
yywrap ()
{return 1; /* terminate when reaching end of stdin */}

ADDRESS searchIdent(void)
/* search the hash table for the identifier in yytext.
insert if
necessary */
{ struct Ident * ptr;
  int h;
  h = hash(yytext);
  ptr = HashTable[h];
  while ((ptr!=NULL) && (strcmp(ptr->name,yytext)!=0))
    ptr = ptr->link;
  if (ptr==NULL)
    if (dcl)
      { ptr = malloc (sizeof (struct Ident));
        ptr->link = HashTable[h];
        strcpy (ptr->name = malloc (yyleng+1), yytext);
        HashTable[h] = ptr;
        ptr->memloc = alloc(1);
        ptr->type = identType;
      }
    else { printf ("%s \n", yytext);
          yyerror ("undeclared identifier");
          return 0;
        }
  else if (dcl)
    { printf ("%s \n", yytext);
      yyerror ("multiply defined identifier");
    }
  return ptr->memloc;
}

int hash(char * str)
{ int h=0, i;
  for (i=0; i<yyleng; i++) h += str[i];

```

```

    return h % HashMax;
}

ADDRESS searchNums(void)
/* search the binary search tree of numbers for the number
in
    yytext. Insert if not found. */
{ struct nums * ptr;
  struct nums * parent;
  double val;
  sscanf (yytext, "%lf", &val);
  if (numsBST==NULL)
    { numsBST = malloc (sizeof (struct nums));
      memory[numsBST->memloc = alloc(1)].data = val;
      numsBST->left = numsBST->right = NULL;
      return numsBST->memloc;
    }
  ptr = numsBST;
  while (ptr!=NULL)
    { if (memory[ptr->memloc].data==val) return ptr->memloc;
      parent = ptr;
      if (memory[ptr->memloc].data<val) ptr = ptr->left;
      else ptr = ptr->right;
    }
  ptr = malloc (sizeof (struct nums));
  memory[ptr->memloc = alloc(1)].data = val;
  ptr->left = ptr->right = NULL;
  if (memory[parent->memloc].data<val) parent->left = ptr;
  else parent->right = ptr;
  return ptr->memloc;
}

ADDRESS alloc (int size)
/* allocate words on the target machine memory */
{
  ADDRESS t;
  t = avail;
  avail += size;
  return t;
}

```

B.3 Syntax Analysis

The syntax analysis phase is implemented with the yacc utility and is stored in the file `MiniC.y`. This file contains the main program, which shows that control is initially given to the `yyparse()` function, which calls the `yylex()` function when it needs a token. The `yyparse()` function creates a file of atoms, and when there are no more tokens in the input stream, it calls the code generator. The file `MiniC.y` is shown below:

```
%{
#include <stdio.h>
#include "mini.h"
#include "miniC.h"
}%
%union {
    ADDRESS address;
    int code; /* comparison code 1-6 */
    struct {int L1;
            int L2;
            int L3;
            int L4;} labels;
}
%token <address> IDENTIFIER
%token <code> INT
%token <code> FLOAT
%token FOR
%token WHILE
%token <code> COMPARISON
%token IF
%token ELSE
%token <address> NUM
/* nonterminal types are: */
%type <code> Type
%type <address> Expr
%type <address> OptExpr
%type <labels> WhileStmt
%type <labels> ForStmt
%type <labels> IfStmt
%type <labels> Label
%right '='
%left COMPARISON
%left '+' '-'
%left '*' '/'
%left UMINUS UPLUS
%%
```



```

                                atom (LBL, NULL, NULL, NULL, 0,
                                    $<labels>8.L3);}
OptExpr:      ;
              Expr          {$$ = $1;}
              |              {$$ = one;} /* default to inf
loop */
              ;
WhileStmt:    WHILE          {$$.L1 = newlabel();
                                atom(LBL, NULL, NULL, NULL, 0, $$.L1);}
              '(' Expr ')'    {$$.L2 = newlabel();
                                atom (TST, $4, zero,
                                    NULL, 1, $$.L2);}
              Stmt            {atom (JMP, NULL, NULL, NULL, 0,
                                    $<labels>2.L1);
                                atom (LBL, NULL, NULL, NULL, 0,
                                    $<labels>6.L2);}
              ;
IfStmt:       IF '(' Expr ')' {$$.L1 = newlabel();
                                atom (TST, $3, zero, NULL, 1, $$.L1);}
              Stmt            {$$.L2 = newlabel();
                                atom (JMP, NULL, NULL, NULL, 0, $$.L2);
                                atom (LBL, NULL, NULL, NULL, 0,
                                    $<labels>5.L1);}
              ElsePart        {atom (LBL, NULL, NULL, NULL, 0,
                                    $<labels>7.L2);}
              ;
ElsePart:     ;
              | ELSE Stmt
              ;
CompoundStmt: '{' StmtList '}'
              ;
StmtList:     StmtList Stmt
              |
              ;
Expr:         IDENTIFIER '=' Expr {atom (MOV, $3, NULL,
                                    $1, 0, 0);
                                    $$ = $3;}
              | Expr COMPARISON Expr
              Label {$$ = alloc(1);
                    atom (MOV, one, NULL, $$, 0, 0);
                    atom (TST, $1, $3, NULL, $2,
                        $4.L1);
                    atom (MOV, zero, NULL, $$, 0, 0);
                    atom (LBL, NULL, NULL, NULL, 0, $4.L1);}

```

```

        | '+' Expr %prec UPLUS  {$$ = $2;}
        | '-' Expr %prec UMINUS {$$ = alloc(1);
                                atom (NEG, $2, NULL, $$, 0, 0); }
        | Expr '+' Expr      {$$ = alloc(1);
                                atom (ADD, $1, $3, $$, 0, 0); }
        | Expr '-' Expr      {$$ = alloc(1);
                                atom (SUB, $1, $3, $$, 0, 0); }
        | Expr '*' Expr      {$$ = alloc(1);
                                atom (MUL, $1, $3, $$, 0, 0); }
        | Expr '/' Expr      {$$ = alloc(1);
                                atom (DIV, $1, $3, $$, 0, 0); }
        | '(' Expr ')'        {$$ = $2;}
        | IDENTIFIER          {$$ = $1; }
        | NUM                  {$$ = $1; }
        ;

Label:                                     {$$.L1 = newlabel();}
        ;                               /* Used to store a label in
                                         compare expr above */

%%
char *programe;
char * op_text();
int lineno = 1;
ADDRESS save;
ADDRESS one;
ADDRESS zero;
int nextlabel = 1;
#include "lex.yy.c"
#include "gen.c"

main (int argc, char *argv[])
{
    programe = argv[0];
    atom_file_ptr = fopen ("atoms", "wb");
    strcpy (yytext, "0.0");
    zero = searchNums(); /* install the constant 0.0
                          in table */
    strcpy (yytext, "1.0");
    one = searchNums(); /* also 1.0 */
    yyparse();
    fclose (atom_file_ptr);
    if (!err_flag) code_gen();
}

yyerror (char * s)
{

```

```

    fprintf(stderr, "%s[%d]: %s\n", progname, lineno, s);
    printf ("yytext is <%s>", yytext);
    err_flag = TRUE;
}

newlabel (void)
{ return nextlabel++;}

/* testing only */  char mne[4];

atom (int operation, ADDRESS operand1, ADDRESS operand2,
      ADDRESS result, int comparison, int dest)
/* put out an atom. destination will be a label number. */
{ struct atom outp;

  outp.op = operation;
  outp.left = operand1;
  outp.right = operand2;
  outp.result = result;
  outp.cmp = comparison;
  outp.dest = dest;

  fwrite (&outp, sizeof (struct atom), 1, atom_file_ptr);
  /* testing only
  printf ("%d %x %x %x %d %d\n", operation, operand1,
          operand2, result, comparison, dest);
  decode (operation);
  printf ("  %s\n", mne);
  */
}

decode (int atom)
/* Convert an atom number to a readable mnemonic */
{
  switch (atom)
    { case ADD:  strcpy (mne, "ADD");
      break;
      case SUB: strcpy (mne, "SUB");
      break;
      case MUL: strcpy (mne, "MUL");
      break;
      case DIV: strcpy (mne, "DIV");
      break;
      case JMP: strcpy (mne, "JMP");

```

```

        break;
    case NEG: strcpy (mne, "NEG");
        break;
    case LBL: strcpy (mne, "LBL");
        break;
    case TST: strcpy (mne, "TST");
        break;
    case MOV: strcpy (mne, "MOV");
    }
}

```

B.4 Code Generator

The code generator is written in the C language; the main function is named `code_gen()` and is stored in the file `gen.c`. There is a `#include` statement in the yacc program in `MiniC.y`, which incorporates the code generator into the MiniC compiler. The function `code_gen()` is called from the main program after the `yparse()` function terminates. `Code_gen()` reads from the file of atoms and writes instructions in hex characters for the Mini machine simulator to `stdout`, the standard output file. This can be displayed on the monitor, stored in a file, or piped directly into the Mini simulator as described, above, in Appendix B.2.

The code generator output also includes a hex location and disassembled op code on each line. These are ignored by the Mini machine simulator and are included only so that the student will be able to read the output and understand how the compiler works.

The first line of output is the starting location of the program instructions. Program variables and temporary storage are located beginning at memory location 0, consequently the Mini machine simulator needs to know where the first instruction is located. The function `out_mem()` sends the constants which have been stored in the target machine memory to `stdout`. The function `dump_atom()` is included for debugging purposes only; the student may use it to examine the atoms produced by the parser.

The code generator solves the problem of forward jump references by making two passes over the input atoms. The first pass is implemented with a function named `build_labels()` which builds a table of Labels (a one dimensional array), associating a machine address with each Label.

The file of atoms is closed and reopened for the second pass, which is implemented with a switch statement on the input atom class. The important function involved here is called `gen()`, and it actually generates a Mini machine instruction, given the operation code (atom class codes and corresponding machine operation codes are the same whenever possible), register number, memory operand address (all addressing is absolute), and a comparison code for compare instructions. Register allocation is kept as

simple as possible by always using floating-point register 1, and storing all results in temporary locations.

The source code for the code generator, from the file `gen.c`, is shown below:

```

struct atom inp;
long labels[MAXL];
ADDRESS pc=0;
int ok = TRUE;

code_gen ()
{ int r;

    /* send target machine memory containing constants to
       stdout */
    end_data = alloc(0);          /* constants precede
                                   instructions */
    out_mem();

    atom_file_ptr = fopen ("atoms","rb"); /* open file of
                                           atoms */
    pc = end_data;                /* starting address of
                                   instructions */
    build_labels();               /* first pass */
    fclose (atom_file_ptr);

    atom_file_ptr = fopen ("atoms","rb"); /* open file of
                                           atoms for */
    get_atom();                  /* second pass */
    pc = end_data;
    ok = TRUE;
    while (ok)
    {
        /* dump_atom(); */

        switch (inp.op)          /* check atom class */
        { case ADD: gen (LOD, r=regalloc(),inp.left);
              gen (ADD, r, inp.right);
              gen (STO, r, inp.result);
              break;
          case SUB:  gen (LOD, r=regalloc(), inp.left);
              gen (SUB, r, inp.right);
              gen (STO, r, inp.result);
              break;
          case NEG:  gen (CLR, r=regalloc());
              gen (SUB, r, inp.left);
        }
    }
}

```

```

        gen (STO, r, inp.result);
        break;
    case MUL:    gen (LOD, r=regalloc(), inp.left);
                gen (MUL, r, inp.right);
                gen (STO, r, inp.result);
                break;
    case DIV:    gen (LOD, r=regalloc(), inp.left);
                gen (DIV, r, inp.right);
                gen (STO, r, inp.result);
                break;
    case JMP:    gen (CMP, 0, 0, 0);
                gen (JMP);
                break;
    case TST:    gen (LOD, r=regalloc(), inp.left);
                gen (CMP, r, inp.right, inp.cmp);
                gen (JMP);
                break;
    case MOV:    gen (LOD, r=regalloc(), inp.left);
                gen (STO, r, inp.result);
                break;
    }
    get_atom();
}
gen (HLT);
}

get_atom()
/* read an atom from the file of atoms into inp */
/* ok indicates that an atom was actually read */
{ int n;

    n = fread (&inp, sizeof (struct atom), 1, atom_file_ptr);
    if (n==0) ok = FALSE;
}

dump_atom()
{ printf ("op: %d  left: %04x  right: %04x  result: %04x
cmp: %d  dest: %d\n",
        inp.op, inp.left, inp.right, inp.result, inp.cmp,
inp.dest); }

gen (int op, int r, ADDRESS add, int cmp)
/* generate an instruction to stdout
   op is the simulated machine operation code

```

```

r is the first operand register
add is the second operand address
cmp is the comparison code for compare instructions
    1 is ==
    2 is <
    3 is >
    4 is <=
    5 is >=
    6 is !=

    jump destination is taken from the atom inp.dest
*/
{union {struct fmt instr;
        unsigned long word;
        } outp;

    outp.word = 0;

    outp.instr.op = op;                /* op code */
    if (op!=JMP)
        { outp.instr.r1 = r;          /* first operand */
          outp.instr.s2 = add;         /* second operand */
        }
    else outp.instr.s2 = lookup (inp.dest); /* jump
                                             destination */
    if (op==CMP) outp.instr.cmp = cmp; /* comparison
                                       code 1-6 */

    printf ("%08x\t%04x\t%s\n", outp.word, pc, op_text(op));
    pc++;
}

int regalloc ()
/* allocate a register for use in an instruction */
{ return 1; }

build_labels()
/* Build a table of label values on the first pass */
{
    get_atom();
    while (ok)
    {
        if (inp.op==LBL)
            labels[inp.dest] = pc;
    }
}

```

```

        /* MOV and JMP atoms require two instructions,
           all other atoms require three instructions. */
        else if (inp.op==MOV || inp.op==JMP) pc += 2;
           else pc += 3;
    get_atom();
    }
}

long lookup (int label_num)
/* look up a label in the table and return its memory address */
{ return labels[label_num];
}

out_mem()
/* send target machine memory contents to stdout.  this is
the beginning of the object file, to be followed by the
instructions.  the first word in the object file is the
starting address of the program; the next word is memory
location 0. */
{
    ADDRESS i;

    printf ("%08x\tLoc\tDisassembled Contents\n", end_data);
        /* starting address of instructions */
    for (i=0; i<end_data; i++)
        printf ("%08x\t%04x\t%8lf\n", memory[i].instr, i,
            memory[i].data);
}

char * op_text(int operation)
/* convert op_codes to mnemonics */
{
    switch (operation)
        { case CLR: return "CLR";
          case ADD: return "ADD";

```

```
        case SUB: return "SUB";  
        case MUL: return "MUL";  
        case DIV: return "DIV";  
        case JMP: return "JMP";  
        case CMP: return "CMP";  
        case LOD: return "LOD";  
        case STO: return "STO";  
        case HLT: return "HLT";  
    }  
}
```