Appendix B

Decaf Compiler

B.1 Installing Decaf

The compiler for Decaf shown in this appendix is implemented using SableCC. The bottom-up parser produces a syntax tree, which when visited by the Translation class, 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 bytes 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:

`decaf.grammar`	Grammar file, input to SableCC
`Translation.java`	Class to implement a translation from syntax tree to atoms
`Compiler.java`	Class containing a main method, to invoke the parser and translator.
`Atom.java`	Class to define an atom
`AtomFile.java`	Class to define the file storing atoms
`gen.c`	Code generator
`mini.c`	Target machine simulator
`mini.h`	Header file for simulator
`miniC.h`	Header file for simulator
`cos.decaf`	Decaf program to compute the cosine function
`bisect.decaf`	Decaf program to compute the square root of two by bisection
`exp.decaf`	Decaf program to compute e^x.
`fact.decaf`	Decaf program to compute the factorial function

`compile`	Script to compile a decaf program and write code to stdout
`compileAndGo`	Script to compile a decaf program and execute the resulting code with the mini simulator.

The source files are available at:

`http://www.rowan.edu/~bergmann/books/java/decaf`

These are all plain text files, so you should be able to simply choose `File | Save` As from your browser window. Create a subdirectory named `decaf`, and download the files `*.java` to the `decaf` subdirectory. Download all other files into your current directory (i.e. the parent of `decaf`) .

To build the Decaf compiler, there are two steps (from the directory containing decaf.grammar). First generate the parser, lexer, analysis, and node classes (the exact form of this command could depend on how SableCC has been installed on your system):

```
$   sablecc decaf.grammar
```

The second step is to compile the java classes that were not generated by SableCC:

```
$   javac decaf/*.java
```

You now have a Decaf compiler; the main method is in decaf/Compiler.class. To compile a decaf program, say cos.decaf, invoke the compiler, and redirect stdin to the decaf source file:

```
$   java decaf.Compiler  < cos.decaf
```

This will create a file named atoms, which is the the result of translating cos.decaf into atoms. To create machine code for the mini architecture, and execute it with a simulator, you will need to compile the code generator, and the simulator, both of which are written in standard C:

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

Now you can generate mini code, and execute it. Simply invoke the code generator, which reads from the file atoms, and writes mini instructions to stdout. You can pipe these instructions into the mini machine simulator:

```
$   gen | mini
```

The above can be simplified by using the scripts provided. To compile the file cos.decaf, without executing, use the `compile` script:

```
$   compile cos
```

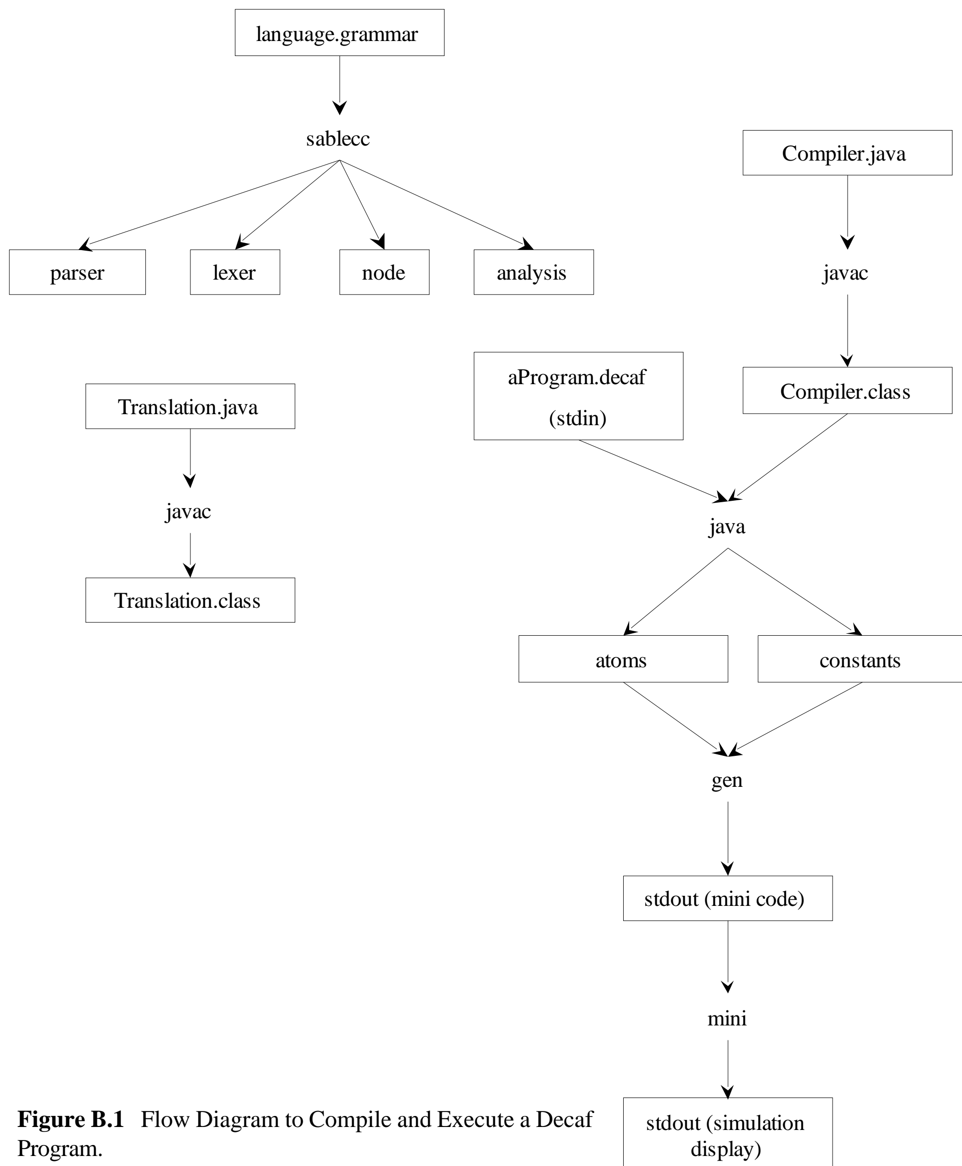

Figure B.1 Flow Diagram to Compile and Execute a Decaf Program.

To compile and execute, use the `compileAndGo` script:

```
$   compileAndGo cos
```

This software was developed and tested using a Sun E450 running SunOS version 5.8. The reader is welcome to adapt it for use on Windows and other systems. 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 rectangles. The source files are shown in Appendix B.2, below, with updated versions available at `http://www.rowan.edu/~bergmann/books`.

B.2 Source Code for Decaf

In this section we show the source code for the Decaf compiler. An updated version may be obtained at `http://www.rowan.edu/~bergmann/books`

The first source file is the grammar file for Decaf, used as input to SableCC. This is described in section 5.5.

```
//       decaf.grammar
// SableCC grammar for decaf, a subset of Java.
// March 2003,  sdb

Package decaf;

Helpers                                    // Examples
  letter = ['a'..'z'] | ['A'..'Z'] ;       //   w
  digit =    ['0'..'9'] ;                  //   3
  digits =   digit+ ;                      // 2040099
  exp  =     ['e' + 'E'] ['+' + '-']? digits;   // E-34
  newline = [10 + 13]   ;
  non_star = [[0..0xffff] - '*'];
  non_slash = [[0..0xffff] - '/'];
  non_star_slash = [[0..0xffff] - ['*' + '/']];


Tokens
  comment1 = '//' [[0..0xffff]-newline]* newline ;
  comment2 = '/*' non_star* '*'
             (non_star_slash non_star* '*'+)* '/' ;

  space = ' ' | 9 | newline ;        // '\t'=9   (tab)
  clas = 'class' ;                   // key words (reserved)
  public = 'public' ;
  static = 'static' ;
  void = 'void' ;
  main = 'main' ;
  string = 'String' ;
  int = 'int' ;
  float = 'float' ;
  for = 'for' ;
  while = 'while' ;
  if = 'if' ;
  else = 'else' ;
  assign = '=' ;
  compare = '==' | '<' | '>' | '<=' | '>=' | '!=' ;
```

```
  plus = '+' ;
  minus = '-' ;
  mult = '*' ;
  div = '/' ;
  l_par = '(' ;
  r_par = ')' ;
  l_brace = '{' ;
  r_brace = '}' ;
  l_bracket = '[' ;
  r_bracket = ']' ;
  comma = ',' ;
  semi = ';' ;
  identifier = letter (letter | digit | '_')* ;
  number  =  (digits '.'? digits? | '.'digits) exp? ;
                        // Example:  2.043e+5
  misc = [0..0xffff] ;

Ignored Tokens
  comment1, comment2, space;


Productions

  program =   clas identifier l_brace public static
                  void main l_par string l_bracket
                  r_bracket [arg]: identifier r_par
                  compound_stmt r_brace ;
  type =                  {int}    int
                        | {float}  float ;
  declaration =         type identifier identlist* semi;
  identlist =           comma identifier ;

  stmt =            {dcl}           declaration
                  | {stmt_no_trlr}  stmt_no_trailer
                  | {if_st}         if_stmt
                  | {if_else_st}    if_else_stmt
                  | {while_st}      while_stmt
                  | {for_st}        for_stmt
                  ;
  stmt_no_short_if = {stmt_no_trlr} stmt_no_trailer
          | {if_else_no_short}    if_else_stmt_no_short_if
          | {while_no_short}      while_stmt_no_short_if
          | {for_no_short}        for_stmt_no_short_if
                        ;
  stmt_no_trailer =       {compound}    compound_stmt
```

```
                    | {null}        semi
                    | {assign}      assign_stmt
                    ;

assign_stmt =         assign_expr semi
                      ;

for_stmt =      for l_par assign_expr? semi bool_expr?
                  [s2]: semi [a2]: assign_expr? r_par stmt
                  ;
for_stmt_no_short_if = for l_par assign_expr? semi
                        bool_expr? [s2]: semi [a2]:
                        assign_expr? r_par
                        stmt_no_short_if
                        ;

while_stmt =    while l_par bool_expr r_par stmt
                ;
while_stmt_no_short_if = while l_par bool_expr r_par
                            stmt_no_short_if
                            ;

if_stmt =       if l_par bool_expr r_par stmt
                ;
if_else_stmt =  if l_par bool_expr r_par stmt_no_short_if
                else stmt
                ;
if_else_stmt_no_short_if =  if l_par bool_expr r_par
                            [if1]: stmt_no_short_if else
                            [if2]: stmt_no_short_if
                            ;

compound_stmt = l_brace stmt* r_brace
                ;

bool_expr =     expr compare [right]: expr
                ;

expr =            {assn} assign_expr
                | {rval} rvalue
                ;
assign_expr =   identifier assign expr ;
rvalue =          {plus}  rvalue plus term
                | {minus} rvalue minus term
                | {term}  term
```

```
                 ;
  term =         {mult}  term mult factor
               | {div}   term div factor
               | {fac}   factor
               ;
  factor =       {pars}   l_par expr r_par
               | {uplus}  plus factor
               | {uminus} minus factor
               | {id}     identifier
               | {num}    number
               ;
```

The file Translation.java is the Java class which visits every node in the syntax tree and produces a file of atoms and a file of constants. It is described in section 5.5.

```
//         Translation.java
// Translation class for decaf, a subset of Java.
// Output atoms from syntax tree
//   sdb  March 2003
//   sdb  updated May 2007
//      to use generic maps instead of hashtables.

package decaf;
import decaf.analysis.*;
import decaf.node.*;
import java.util.*;
import java.io.*;

class Translation extends DepthFirstAdapter
{

// All stored values are doubles, key=node, value is memory
      loc or label number
Map  <Node, Integer> hash = new HashMap <Node, Integer> ();
      // May 2007

Integer zero = new Integer (0);
Integer one  = new Integer (1);

AtomFile out;

//////////////////////////////////////////////////
// Definition of Program

public void inAProgram (AProgram prog)
```

```
//   The class name and main args need to be entered into
      symbol table
//     to avoid error message.
//   Also, open the atom file for output
{  identifiers.put (prog.getIdentifier().toString(),
      alloc());   // class name
   identifiers.put (prog.getArg().toString(), alloc());
            // main (args)
   out = new AtomFile ("atoms");
}

public void outAProgram (AProgram prog)
//  Write the run-time memory values to a file "constants".
//  Close the binary file of atoms so it can be used for
//    input by the code generator
{  outConstants();
   out.close();
}

//////////////////////////////////////////////////
// Definitions of declaration and identlist

public void inADeclaration (ADeclaration node)
{   install (node.getIdentifier()); }

public void outAIdentlist (AIdentlist node)
{   install (node.getIdentifier()); }

void install (TIdentifier id)
//  Install id into the symbol table
{  Integer loc;
   loc = identifiers.get (id.toString());
   if (loc==null)
      identifiers.put (id.toString(), alloc());
   else
      System.err.println ("Error: " + id + " has already
      been declared ");
}


//////////////////////////////////////////////////
// Definition of for_stmt

public void caseAForStmt (AForStmt stmt)
{  Integer lbl1, lbl2, lbl3;
```

```
    lbl1 = lalloc();
    lbl2 = lalloc();
    lbl3 = lalloc();
    inAForStmt (stmt);
    if (stmt.getFor() !=null)  stmt.getFor().apply(this);
    if (stmt.getLPar() !=null) stmt.getLPar().apply(this);
    if (stmt.getAssignExpr()!=null)              // initial-
       ize
       {     stmt.getAssignExpr().apply(this);
             atom ("LBL", lbl1);
       }
    if (stmt.getSemi() != null)
       stmt.getSemi().apply(this);
    if (stmt.getBoolExpr() != null)              // test for
       termination
       {     stmt.getBoolExpr().apply(this);
             atom ("JMP", lbl2);
             atom ("LBL", lbl3);
       }
    if (stmt.getS2() != null)  stmt.getS2().apply(this);
    if (stmt.getA2() != null)
       {     stmt.getA2().apply(this);           // increment
             atom ("JMP", lbl1);
             atom ("LBL", lbl2);
       }
    if (stmt.getRPar() != null)
       stmt.getRPar().apply(this);
    if (stmt.getStmt() != null)
       {     stmt.getStmt().apply(this);
             atom ("JMP", lbl3);
             atom ("LBL", (Integer) hash.get
       (stmt.getBoolExpr()));
       }
    outAForStmt(stmt);
}

public void caseAForStmtNoShortIf (AForStmtNoShortIf stmt)
{  Integer lbl1, lbl2, lbl3;
    lbl1 = lalloc();
    lbl2 = lalloc();
    lbl3 = lalloc();
    inAForStmtNoShortIf (stmt);
    if (stmt.getFor() !=null)  stmt.getFor().apply(this);
    if (stmt.getLPar() !=null) stmt.getLPar().apply(this);
```

```
   if (stmt.getAssignExpr()!=null)              // initial-
      ize
      {     stmt.getAssignExpr().apply(this);
            atom ("LBL", lbl1);
      }
   if (stmt.getSemi() != null)
      stmt.getSemi().apply(this);
   if (stmt.getBoolExpr() != null)              // test for
      termination
      {     stmt.getBoolExpr().apply(this);
            atom ("JMP", lbl2);
            atom ("LBL", lbl3);
      }
   if (stmt.getS2() != null)  stmt.getS2().apply(this);
   if (stmt.getA2() != null)
      {     stmt.getA2().apply(this);           // increment
            atom ("JMP", lbl1);
            atom ("LBL", lbl2);
      }
   if (stmt.getRPar() != null)
      stmt.getRPar().apply(this);
   if (stmt.getStmtNoShortIf() != null)
      {     stmt.getStmtNoShortIf().apply(this);
            atom ("JMP", lbl3);
            atom ("LBL", (Integer) hash.get
      (stmt.getBoolExpr()));
      }
   outAForStmtNoShortIf (stmt);
}

//////////////////////////////////////////////////
// Definition of while_stmt

public void inAWhileStmt (AWhileStmt stmt)
{  Integer lbl = lalloc();
   hash.put (stmt, lbl);
   atom ("LBL", lbl);
}

public void outAWhileStmt (AWhileStmt stmt)
{  atom ("JMP", (Integer) hash.get(stmt));
   atom ("LBL", (Integer) hash.get (stmt.getBoolExpr()));
}

public void inAWhileStmtNoShortIf (AWhileStmtNoShortIf stmt)
```

```
{  Integer lbl = lalloc();
   hash.put (stmt, lbl);
   atom ("LBL", lbl);
}

public void outAWhileStmtNoShortIf (AWhileStmtNoShortIf
      stmt)
{  atom ("JMP", (Integer) hash.get(stmt));
   atom ("LBL", (Integer) hash.get (stmt.getBoolExpr()));
}


/////////////////////////////////////////////
// Definition of if_stmt

public void outAIfStmt (AIfStmt stmt)
{  atom ("LBL", (Integer) hash.get (stmt.getBoolExpr())); }
      // Target for bool_expr's TST

// override the case of if_else_stmt
public void caseAIfElseStmt (AIfElseStmt node)
{  Integer lbl = lalloc();
   inAIfElseStmt (node);
   if (node.getIf() != null) node.getIf().apply(this);
   if (node.getLPar() != null) node.getLPar().apply(this);
   if (node.getBoolExpr() !=
      null)node.getBoolExpr().apply(this);
   if (node.getRPar() != null) node.getRPar().apply(this);
   if (node.getStmtNoShortIf() != null)
     {       node.getStmtNoShortIf().apply(this);
      atom ("JMP", lbl);                            //
      Jump over else part
      atom ("LBL", (Integer) hash.get (node.getBoolExpr()));
     }
   if (node.getElse() != null) node.getElse().apply(this);
   if  (node.getStmt() != null) node.getStmt().apply(this);
   atom ("LBL", lbl);
   outAIfElseStmt (node);
}

// override the case of if_else_stmt_no_short_if
public void caseAIfElseStmtNoShortIf (AIfElseStmtNoShortIf
      node)
{  Integer lbl = lalloc();
   inAIfElseStmtNoShortIf (node);
```

```
   if (node.getIf() != null) node.getIf().apply(this);
   if (node.getLPar() != null) node.getLPar().apply(this);
   if (node.getBoolExpr() !=
      null)node.getBoolExpr().apply(this);
   if (node.getRPar() != null) node.getRPar().apply(this);
   if (node.getIf1() != null)
     {      node.getIf1().apply(this);
      atom ("JMP", lbl);                              //
      Jump over else part
      atom ("LBL", (Integer) hash.get (node.getBoolExpr()));
     }
   if (node.getElse() != null) node.getElse().apply(this);
   if  (node.getIf2() != null) node.getIf2().apply(this);
   atom ("LBL", lbl);
   outAIfElseStmtNoShortIf (node);
}

///////////////////////////////////////////////////
// Definition of bool_expr

public void outABoolExpr (ABoolExpr node)
{  Integer lbl = lalloc();
   hash.put (node, lbl);
   atom ("TST", (Integer) hash.get(node.getExpr()),
            (Integer) hash.get(node.getRight()),
            zero,
            new Integer (7  - getComparisonCode
      (node.getCompare().toString())),
                        // Negation of a comparison code is
      7 - code.
            lbl);
}


////////////////////////////////////////////////
// Definition of expr

public void outAAssnExpr (AAssnExpr node)
// out of alternative {assn} in expr
{ hash.put (node, hash.get (node.getAssignExpr())); }

public void outARvalExpr (ARvalExpr node)
// out of alternative {rval} in expr
{  hash.put (node, hash.get (node.getRvalue())); }
```

```
int getComparisonCode (String cmp)
//  Return the integer comparison code for a comparison
{   if (cmp.indexOf ("==")>=0) return  1;
    if (cmp.indexOf ("<")>=0)  return  2;
    if (cmp.indexOf (">")>=0)  return  3;
    if (cmp.indexOf ("<=")>=0) return  4;
    if (cmp.indexOf (">=")>=0) return  5;
    if (cmp.indexOf ("!=")>=0) return  6;
  return  0;                  // this should never occur
}

////////////////////////////////////////////////
// Definition of assign_expr

public void outAAssignExpr (AAssignExpr node)
//  Put out the MOV atom
{  Integer assignTo = getIdent (node.getIdentifier());
   atom ("MOV", (Integer) hash.get (node.getExpr()),
       zero,
       assignTo);
   hash.put (node, assignTo);
}


////////////////////////////////////////////////
// Definition of rvalue

public void outAPlusRvalue (APlusRvalue node)
{// out of alternative {plus} in Rvalue, generate an atom
      ADD.
    Integer i = alloc();
    hash.put (node, i);
    atom ("ADD", (Integer)hash.get(node.getRvalue()),
               (Integer) hash.get(node.getTerm()) , i);
}

public void outAMinusRvalue(AMinusRvalue node)
{// out of alternative {minus} in Rvalue, generate an atom
      SUB.
    Integer i = alloc();
    hash.put (node, i);
    atom ("SUB", (Integer) hash.get(node.getRvalue()),
             (Integer) hash.get(node.getTerm()), i);
}
```

```
public void outATermRvalue (ATermRvalue node)
//  Attribute of the rvalue is the same as the term.
{   hash.put (node, hash.get (node.getTerm())); }


////////////////////////////////////////////////
// Definition of term

public void outAMultTerm (AMultTerm node)
{// out of alternative {mult} in Term, generate an atom MUL.
    Integer i = alloc();
    hash.put (node, i);
    atom ("MUL", (Integer)hash.get(node.getTerm()),
               (Integer) hash.get(node.getFactor()) , i);
}

public void outADivTerm(ADivTerm node)
{// out of alternative {div} in Term, generate an atom DIV.
    Integer i = alloc();
    hash.put (node, i);
    atom ("DIV", (Integer) hash.get(node.getTerm()),
             (Integer) hash.get(node.getFactor()), i);
}

public void outAFacTerm (AFacTerm node)
{ //  Attribute of the term is the same as the factor
      hash.put (node, hash.get(node.getFactor()));
 }


Map <Double, Integer> nums = new HashMap <Double, Integer>
      ();
Map <String, Integer > identifiers = new HashMap <String,
      Integer> ();

final int MAX_MEMORY = 1024;
Double memory [] = new Double [MAX_MEMORY];
int memHigh = 0;
// No, only memory needs to remain for codegen.


// Maintain a hash table of numeric constants, to avoid
      storing
//   the same number twice.
```

```
// Move the number to a run-time memory location.
// That memory location will be the attribute of the Number
      token.
public void caseTNumber(TNumber num)
{ Integer loc;
  Double dnum;
  dnum = new Double (num.toString());           // The
      number as a Double
  loc = (Integer) nums.get (dnum);        // Get its memory
      location
  if (loc==null)                    // Already in table?
     {      loc = alloc();                       // No,
      install in table of nums
      nums.put (dnum, loc);
      memory[loc.intValue()] = dnum;            // Store
      value in run-time memory
      if (loc.intValue() > memHigh)       // Retain highest
      memory loc
         memHigh = loc.intValue();
     }
  hash.put (num, loc);                    // Set attribute
      to move up tree
}


Integer getIdent(TIdentifier id)
// Get the run-time memory location to which this id is
      bound
{  Integer loc;
   loc = identifiers.get (id.toString());
   if (loc==null)
      System.err.println ("Error: " + id + " has not been
      declared");
   return loc;
}

////////////////////////////////////////////////
// Definition of factor

public void outAParsFactor (AParsFactor node)
{   hash.put (node, hash.get (node.getExpr())); }

// Unary + doesn't need any atoms to be put out.
public void outAUplusFactor (AUplusFactor node)
{   hash.put (node, hash.get (node.getFactor())); }
```

```
// Unary - needs a negation atom (NEG).
public void outAUminusFactor (AUminusFactor node)
{   Integer loc = alloc();    // result of negation
    atom ("NEG", (Integer)hash.get(node.getFactor()), zero,
      loc);
    hash.put (node, loc);
}

public void outAIdFactor (AIdFactor node)
{   hash.put (node, getIdent (node.getIdentifier())); }

public void outANumFactor (ANumFactor node)
{   hash.put (node, hash.get (node.getNumber())); }


////////////////////////////////////////////////////////////
      ///////
//  Send the run-time memory constants to a file for use by
      the code generator.

void outConstants()
{  FileOutputStream fos = null;
   DataOutputStream ds = null;
   int i;

   try
    { fos = new FileOutputStream ("constants");
      ds = new DataOutputStream (fos);
    }
   catch (IOException ioe)
    { System.err.println ("IO error opening constants file
      for output: "
            + ioe);
    }

   try
    { for (i=0; i<=memHigh ; i++)
          if (memory[i]==null) ds.writeDouble (0.0);
      // a variable is bound here
          else
            ds.writeDouble (memory[i].doubleValue());
    }
   catch (IOException ioe)
```

```
        { System.err.println ("IO error writing to constants
          file: "
                + ioe);
        }
      try { fos.close(); }
      catch (IOException ioe)
        { System.err.println ("IO error closing constants file:
          "
                + ioe);
        }
    }


    //////////////////////////////////////////////////////////
    // Put out atoms for conversion to machine code.
    // These methods display to stdout, and also write to a
    //   binary file of atoms suitable as input to the code
          generator.

    void atom (String atomClass, Integer left, Integer right,
          Integer result)
    {   System.out.println (atomClass +  " T" + left + " T"  +
          right +  " T" +
          result);
        Atom atom = new Atom (atomClass, left, right, result);
        atom.write(out);
    }

    void atom (String atomClass, Integer left, Integer right,
          Integer result,
          Integer cmp, Integer lbl)
    {   System.out.println (atomClass +  " T" + left + " T"  +
          right +  " T" +
          result + " C" + cmp + " L" + lbl);
        Atom atom = new Atom (atomClass, left, right, result,
          cmp, lbl);
        atom.write(out);
    }

    void atom (String atomClass, Integer lbl)
    {   System.out.println (atomClass +  " L" + lbl);
        Atom atom = new Atom (atomClass, lbl);
        atom.write(out);
    }
```

```
static int avail = 0;
static int lavail = 0;

Integer alloc()
{ return new Integer (++avail); }

Integer lalloc()
{ return new Integer (++lavail); }


}
```

The file `Compiler.java` defines the Java class which contains a `main` method which invokes the parser to get things started. It is described in section 5.5.

```
//       Compiler.java
//  main method which invokes the parser and reads from
//     stdin
//  March 2003   sdb

package decaf;
import decaf.parser.*;
import decaf.lexer.*;
import decaf.node.*;
import java.io.*;

public class Compiler
{

public static void main(String[] arguments)
{ try
  { System.out.println();

    // Create a Parser instance.
    Parser p = new Parser
        ( new Lexer
          ( new PushbackReader
            ( new InputStreamReader(System.in),
                                              1024)));

    // Parse the input.
```

```
    Start tree = p.parse();

    // Apply the translation.
    tree.apply(new Translation());

    System.out.println();
  }
  catch(Exception e)
    { System.out.println(e.getMessage()); }
}
}
```

The file `Atom.java` defines the Java class Atom, which describes an Atom and permits it to be written to an output file. It is described in section 5.5.

```
//      Atom.java
// Define an atom for output by the Translation class

package decaf;
import java.io.*;

class Atom
// Put out atoms to a binary file in the format expected by
//  the old code generator.
{
static final int ADD = 1;
static final int SUB = 2;
static final int MUL = 3;
static final int DIV = 4;
static final int JMP = 5;
static final int NEG = 10;
static final int LBL = 11;
static final int TST = 12;
static final int MOV = 13;

int cls;
int left;
int right;
int result;
int cmp;
int lbl;
```

```
// constructors for Atom
Atom (String cls, Integer l, Integer r, Integer res)
{   setClass (cls);
    left = l.intValue();
    right = r.intValue();
    result = res.intValue();
    cmp = 0;
    lbl = 0;
}

Atom (String cls, Integer l, Integer r, Integer res,
            Integer c, Integer lb)
{   setClass (cls);
    left = l.intValue();
    right = r.intValue();
    result = res.intValue();
    cmp = c.intValue();
    lbl = lb.intValue();
}

Atom (String cls,  Integer lb)
{   setClass (cls);
    left = 0;
    right = 0;
    result = 0;
    cmp = 0;
    lbl = lb.intValue();
}
}

void setClass (String c)
//  set the atom class as an int code
{   if (c.equals("ADD")) cls = ADD;
    else if (c.equals("SUB")) cls = SUB;
    else if (c.equals("MUL")) cls = MUL;
    else if (c.equals("DIV")) cls = DIV;
    else if (c.equals("JMP")) cls = JMP;
    else if (c.equals("NEG")) cls = NEG;
    else if (c.equals("LBL")) cls = LBL;
    else if (c.equals("TST")) cls = TST;
    else if (c.equals("MOV")) cls = MOV;
}

void write (AtomFile out)
 // write a single atom out to the binary file
```

```
{
    try
     {  out.ds.writeInt (cls);
        out.ds.writeInt (left);
        out.ds.writeInt (right);
        out.ds.writeInt (result);
        out.ds.writeInt (cmp);
        out.ds.writeInt (lbl);
     }
    catch (IOException ioe)
     {
        System.out.println
            ("IO Error writing to atom file, atom class is "
                + cls + ", error is " + ioe);
     }
}
}
```

The file `AtomFile.java` defines the Java class AtomFile, which permits output of an Atom to a file.

```
//          AtomFile.java
//  Create the binary output file for atoms
//  March 2003   sdb

package decaf;
import java.io.*;

class AtomFile
{
   FileOutputStream fos;
   DataOutputStream ds;
   String fileName;


AtomFile (String name)
{  fileName = new String (name);
   try
     {  fos = new FileOutputStream (fileName);
        ds = new DataOutputStream (fos);
     }
   catch (IOException ioe)
     {
        System.err.println ("IO error opening atom file
                  (" + fileName + "): " + ioe);
```

```
        }
    }

    void close()
    {   try
         { ds.close(); }
        catch (IOException ioe)
         {  System.err.println ("IO error closing atom file
                      (" + fileName + "): " + ioe);
         }


    }
    }
```

B.3 Code Generator

The code generator is written in the C language; we re-use the code generator written for the C/C++ version of this book, and it is stored in the file `gen.c`. This program reads from a file named 'atoms' and a file named 'constants' which are produced by the Translation class from the syntax tree. It 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.1.

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. For an updated version of this source code, see
`http://www.rowan.edu/~bergmann/books`

```
/*      gen.c
   Code generator for mini architecture.
  Input should be a file of atoms, named "atoms"

********************************************************
   Modified March 2003 to work with decaf as well as miniC.
   sdb
*/
#define NULL 0
#include "mini.h"
#include "miniC.h"

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

/* code_gen ()  */
void main()     /* March 2003, for Java.  sdb  */
{ int r;

  /* send target machine memory containing constants to
      stdout */
/*  end_data = alloc(0);         March 2003   sdb
    constants precede instructions */
/* send run-time memory  constants to stdout */
  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);

  /* open file of atoms for */
```

```
 atom_file_ptr = fopen ("atoms","rb");
 get_atom();                        /* second pass */
 pc = end_data;
 ok = TRUE;
while (ok)
 {
/*  dump_atom();  */          /* for debugging, etc */

 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 it's 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 wordin the object file is
      the starting address of the program; the next word is
      memory location 0. */
{
  ADDRESS i;
  data_file_ptr = fopen ("constants","rb");
                              /* open file of constants
```

```
                                March 2003   sdb*/
  get_data();


  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);
}


void get_data()
/* March 2003  sdb
   read a number from the file of constants into inp_const
   and store into memory.
*/
{ int i,status=1;
  double inp_const;
  for (i=0; status; i++)
    {   status = fread (&inp_const, sizeof (double), 1,
      data_file_ptr);
        memory[i].data = inp_const ;
    }
  end_data = i-1;
}


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";
        }
}
```