CHAPTER EIGHT

Interpretation

An interpreter takes a source program and executes it immediately. Immediacy is the key characteristic of interpretation; there is no prior time-consuming translation of the source program into a low-level representation.

In an interactive environment, immediacy is highly advantageous. For example, the user of a command language expects an immediate response from each command; it would be absurd to expect the user to enter an entire sequence of commands before seeing the response from the first one. Similarly, the user of a database query language expects an immediate answer to each query. In this mode of working, the 'program' is entered once and then discarded.

The user of a programming language, on the other hand, is much more likely to retain the program for further use, and possibly further development. Even so, translation from the programming language to an intermediate language followed by interpretation of the intermediate language (i.e., interpretive compilation) is a good alternative to full compilation, especially during the early stages of program development.

In this chapter we study two kinds of interpretation:

- *iterative interpretation*
- *recursive interpretation.*

Iterative interpretation is suitable when the source language's instructions are all primitive. The instructions of the source program are fetched, analyzed, and executed, one after another. Iterative interpretation is suitable for real and abstract machine codes, for some very simple programming languages, and for command languages.

Recursive interpretation is necessary if the source language has composite instructions. (In this context, 'instructions' could be statements, expressions, and/or declarations.) Interpretation of an instruction may trigger interpretation of its component instructions. An interpreter for a high-level programming language or query language must be recursive. However, recursive interpretation is slower and more complex than iterative interpretation, so we usually prefer to compile high-level languages, or at least translate them to lower-level intermediate languages that are suitable for iterative interpretation.

8.1 Iterative interpretation

Conventional interpreters are iterative: they work in a fetch–analyze–execute cycle. This is captured by the following ***iterative interpretation scheme***:

```
initialize
do {
    fetch the next instruction
    analyze this instruction
    execute this instruction
} while (still running);
```

First, an instruction is fetched from storage, or in some cases entered directly by the user. Second, the instruction is analyzed into its component parts. Third, the instruction is executed. The whole cycle is then repeated.

Typically the source language has several forms of instruction, so execution of an instruction decomposes into several cases, one case for each form of instruction.

In the following subsections we apply this scheme to the interpretation of machine code, the interpretation of simple command languages, and the interpretation of simple programming languages.

8.1.1 Iterative interpretation of machine code

An interpreter of machine code is often called an ***emulator***. It is worth recalling here that a real machine M is functionally equivalent to an emulator of M's machine code. The only difference is that a real machine uses electronic (and perhaps parallel) hardware to fetch, analyze, and execute instructions, and is therefore much faster than an emulator. (Refer back to Section 2.3 for a fuller discussion of this point.)

A machine-code instruction is essentially a record, consisting of an operation field (usually called the *op-code*) and some operand fields. Instruction analysis (or *decoding*) is simply unpacking these fields. Instruction execution is controlled by the op-code.

To implement an emulator, we employ the following simple techniques:

- Represent the machine's storage by an array. If storage is partitioned, for example into separate stores for code and data, then represent each store by a separate array.
- Represent the machine's registers by variables. This applies equally to visible and hidden registers.[1] One register, the *code pointer* or *program counter*, will contain the address of the next instruction to be executed. Another, the *status register*, will be used to control program termination.

[1] Hidden registers are those that cannot be accessed explicitly, even by a machine-code program.

- Fetch each instruction from the (code) store.
- Analyze each instruction by isolating its op-code and operand field(s).
- Execute each instruction by means of a switch-statement, with one case for each possible value of the op-code. In each case, emulate the instruction by updating storage and/or registers.

Example 8.1 Interpreter for Hypo

Consider the hypothetical machine, *Hypo*, summarized in Table 8.1.

Hypo has a 4096-word code store that contains the instructions of the program. An instruction consists of a 4-bit op-code *op* and a 12-bit operand field *d*. Details of the instructions are given in Table 8.1. The program counter, PC, contains the address of the next instruction to be executed. The instruction located at address 0 will be executed first.

Hypo also has a 4096-word data store and a 1-word accumulator, ACC. Each word consists of 16 bits. Data may be placed anywhere in the data store.

Figure 8.1 illustrates the Hypo code store and data store. The illustrative program in the code store takes two integers (already stored at addresses 0 and 1) and computes their product (at address 2). (For greater readability, the program is also shown with mnemonic op-codes.)

Figure 8.2 illustrates how the machine's state – data store, accumulator, and program counter – would change during the first few execution steps of the stored program.

The following class represents Hypo instructions:

```
public class HypoInstruction {
    public byte op;                          // op-code field (0 .. 7)
    public short d;                          // operand field (0 .. 4095)

    public static final byte                 // op-codes, as in Table 8.1
        STOREop = 0, LOADop  = 1,
        LOADLop = 2, ADDop   = 3,
        SUBop   = 4, JUMPop  = 5,
        JUMPZop = 6, HALTop  = 7;
}
```

The following class represents the machine's state:

```
public class HypoState {

    public static final short CODESIZE = 4096;
    public static final short DATASIZE = 4096;

    // Code store ...
    public HypoInstruction[] code =
          new HypoInstruction[CODESIZE];
```

```
    // Data store ...
    public short[] data = new short[DATASIZE];

    // Registers ...
    public short PC;
    public short ACC;
    public byte status;

    public static final byte  // status values
        RUNNING = 0, HALTED = 1, FAILED = 2;
}
```

Here the code store is represented by an array of instructions, `code`; the data store is represented by an array of words, `data`; and the registers are represented by variables `PC`, `ACC`, and `status`.

The following class will implement the Hypo loader and emulator:

```
public class HypoInterpreter extends HypoState {

    public void load () {
        ... // Load the program into the code store,
            // starting at address 0.
    }

    public void emulate () {
        ... // Run the program contained in the code store,
            // starting at address 0.
    }

}
```

*The following method is the emulator proper. Its control structure is a switch-*statement within a loop, preceded by initialization of the registers. Each case of the switch-statement follows directly from Table 8.1.

```
public void emulate () {

    // Initialize ...
    PC = 0;  ACC = 0;  status = RUNNING;

    do {

        // Fetch the next instruction ...
        HypoInstruction instr = code[PC++];

        // Analyze this instruction ...
        byte op = instr.op;
        short d = instr.d;

        // Execute this instruction ...
        switch (op) {
```

```
            case STOREop:  data[d] = ACC; break;
            case LOADop:   ACC = data[d]; break;
            case LOADLop:  ACC = d; break;
            case ADDop:    ACC += data[d]; break;
            case SUBop:    ACC -= data[d]; break;
            case JUMPop:   PC = d; break;
            case JUMPZop:  if (ACC == 0) PC = d; break;
            case HALTop:   status = HALTED; break;

            default:       status = FAILED;

        }

    } while (status == RUNNING);
}
```

This emulator has been kept as simple as possible, for clarity. But it might behave unexpectedly if, for example, an ADD or SUB instruction overflows. A more robust version would set status to FAILED in such circumstances. (See Exercise 8.1.)

□

When we write an interpreter like that of Example 8.1, it makes no difference whether we are interpreting a real machine code or an abstract machine code. For an abstract machine code, the interpreter will be the only implementation. For a real machine code, a hardware interpreter (processor) will be available as well as a software interpreter (emulator). Of these, the processor will be much the faster. But an emulator is much more flexible than a processor: it can be adapted cheaply for a variety of purposes. An emulator can be used for experimentation before the processor is ever constructed. An emulator can also easily be extended for diagnostic purposes. (Exercises 8.2 and 8.3 suggest some of the possibilities.) So, even when a processor is available, an emulator for the same machine code complements it nicely.

Table 8.1 Instruction set of the hypothetical machine Hypo.

Op-code	Instruction	Meaning
0	STORE *d*	word at address *d* ← ACC
1	LOAD *d*	ACC ← word at address *d*
2	LOADL *d*	ACC ← *d*
3	ADD *d*	ACC ← ACC + word at address *d*
4	SUB *d*	ACC ← ACC – word at address *d*
5	JUMP *d*	PC ← *d*
6	JUMPZ *d*	PC ← *d*, if ACC = 0
7	HALT	stop execution

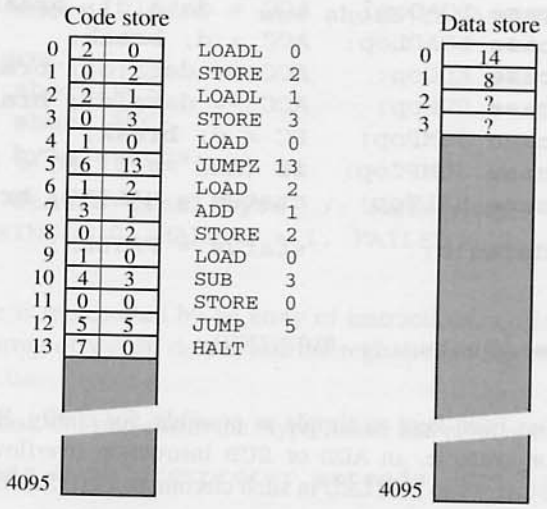

Figure 8.1 Hypo: code store and data store.

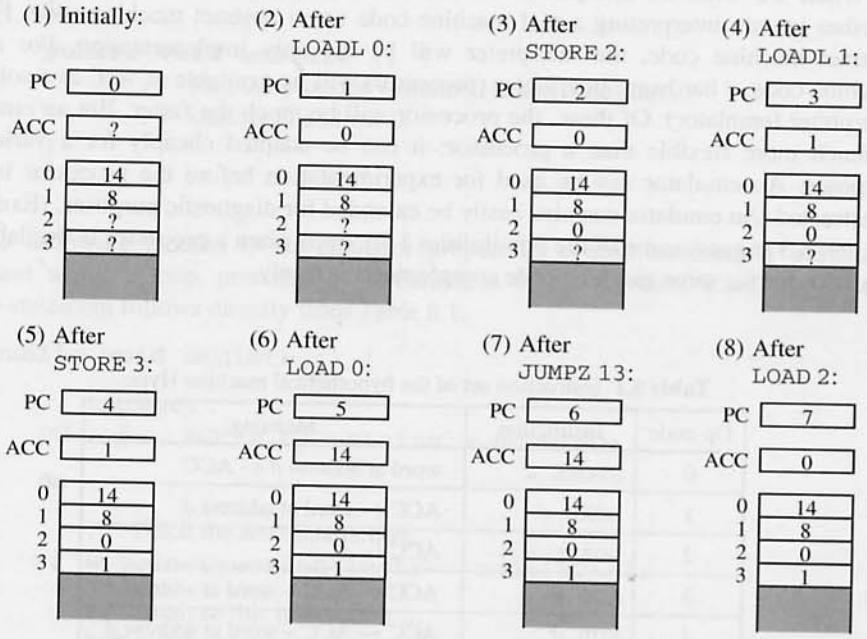

Figure 8.2 Hypo: state changes.

8.1.2 Iterative interpretation of command languages

Command languages (such as the UNIX *shell* language) are relatively simple languages. In normal usage, the user enters a sequence of commands, and expects an immediate response to each command. Each command will be executed just once. These factors suggest interpretation of each command as soon as it is entered. In fact, command languages are specifically designed to be interpreted. Below we illustrate interpretation of a simple command language.

Example 8.2 Interpreter for Mini-Shell

Consider a simple command language, *Mini-Shell*, that allows us to enter commands such as:

```
delete a b c
create f
list
edit f
/bin/sort f
print f 2
quit
```

The above is an example of a *script*, which is just a sequence of commands. Each command is to be executed as soon as it is entered.

Mini-Shell provides several built-in commands. In addition, any executable program (such as `/bin/sort`) can be run simply by giving the name of the file containing it. A command can be passed any number of arguments, which may be filenames or literals. The commands and their meanings are given in Table 8.2.

Table 8.2 Commands in Mini-Shell.

Command	Argument(s)	Meaning
`create`	filename	Create an empty file with the given name.
`delete`	$\text{filename}_1 \ldots \text{filename}_n$	Delete all the named files.
`edit`	filename	Edit the named file.
`list`	none	List the names of all files owned by the current user.
`print`	filename number	Print the given number of copies of the named file.
Run executable program	filename $\text{arg}_1 \ldots \text{arg}_n$	Run the executable program contained in the named file, with the given arguments.

The syntax of a script is as follows:

Script	::=	Command*	(8.1)
Command	::=	Command-Name Argument* end-of-line	(8.2)
Argument	::=	Filename	(8.3a)
	|	Literal	(8.3b)
Command-Name	::=	**create**	(8.4a)
	|	**delete**	(8.4b)
	|	**edit**	(8.4c)
	|	**list**	(8.4d)
	|	**print**	(8.4e)
	|	**quit**	(8.4f)
	|	Filename	(8.4g)

Production rules for Filename and Literal have been omitted here.

In the Mini-Shell interpreter, we can represent commands as follows:

```
public class MiniShellCommand {
    public String   name;
    public String[] args;
}
```

The following class represents the Mini-Shell state:

```
public class MiniShellState {

    // File store ...
    public ...;

    // Registers ...
    public byte status;   // RUNNING or HALTED or FAILED

    public static final byte   // status values
        RUNNING = 0, HALTED = 1, FAILED = 2;
}
```

There is no need for either a code store or a code pointer, since each command will be executed only once, as soon as it is entered.

The following class will implement the Mini-Shell interpreter:

```
public class MiniShell extends MiniShellState {

    public void interpret () {
        ... // Execute the commands entered by the user,
            // terminating on command quit.
    }
```

```
    public MiniShellCommand readAnalyze () {
        ... // Read, analyze, and return the next command entered by the user.
    }

    public void create (String fname) {
        ... // Create an empty file with the given name.
    }

    public void delete (String[] fnames) {
        ... // Delete all the named files.
    }

    public void edit (String fname) {
        ... // Edit the named file.
    }

    public void list (){
        ... // List names of all files owned by the current user.
    }

    public void print (String fname, String copies) {
        ... // Print the given number of copies of the named file.
    }

    public void exec (String fname, String[] args) {
        ... // Run the executable program contained in the named file, with
            // the given arguments.
    }

}
```

It will be convenient to combine fetching and analysis of commands. This is done by method `readAnalyze`.

The following method is the interpreter proper. It just reads, analyzes, and executes the commands, one after another:

```
public void interpret () {

    // Initialize ...
    status = RUNNING;

    do {
        // Fetch and analyze the next instruction ...
        MiniShellCommand com = readAnalyze();

        // Execute this instruction ...

        if (com.name.equals("create"))
            create(com.args[0]);

        else if (com.name.equals("delete"))
            delete(com.args);
```

```
            else if (com.name.equals("edit"))
               edit(com.args[0]);

            else if (com.name.equals("list"))
               list();

            else if (com.name.equals("print"))
               print(com.args[0], com.args[1]);

            else if (com.name.equals("quit"))
               status = HALTED;

            else  // executable program
               exec(com.name, com.args);

         } while (status == RUNNING);
      }
```

□

8.1.3 Iterative interpretation of simple programming languages

Iterative interpretation is also possible for certain programming languages, provided that a source program is just a sequence of primitive commands. The programming language must not include any composite commands, i.e., commands that contain subcommands.

In the iterative interpretation scheme, the 'instructions' are taken to be the commands of the programming language. Analysis of a command consists of syntactic and perhaps contextual analysis. This makes analysis far slower and more complex than decoding a machine-code instruction. Execution is controlled by the form of command, as determined by syntactic analysis.

Example 8.3 Interpreter for Mini-Basic

Consider a simple programming language, *Mini-Basic*, with the following syntax (expressed in EBNF):

Program	::=	Command*	(8.5)
Command	::=	Variable = Expression	(8.6a)
	|	**read** Variable	(8.6b)
	|	**write** Variable	(8.6c)
	|	**go** Label	(8.6d)
	|	**if** Expression Relational-Op Expression **go** Label	(8.6e)
	|	**stop**	(8.6f)
Expression	::=	primary-Expression	(8.7a)
	|	Expression Arithmetic-Op primary-Expression	(8.7b)

primary-Expression	::=	Numeral	(8.8a)
	|	Variable	(8.8b)
	|	**(** Expression **)**	(8.8c)
Arithmetic-Op	::=	**+** | **-** | ***** | **/**	(8.9a–d)
Relational-Op	::=	**=** | **\=** | **<** | **=<** | **>=** | **>**	(8.10a–f)
Variable	::=	**a** | **b** | **c** | ... | **z**	(8.11a–z)
Label	::=	Digit Digit*	(8.12)

A Mini-Basic program is just a sequence of commands. The commands are implicitly labeled `0`, `1`, `2`, etc., and these labels may be referenced in `go` and `if` commands. The program may use up to twenty-six variables, which are predeclared.

The semantics of Mini-Basic programs should be intuitively clear. All values are real numbers. The program shown in Figure 8.3 reads a number (into variable `a`), computes its square root accurate to two decimal places (in variable `b`), and writes the square root.

It is easy to imagine a *Mini-Basic abstract machine*. The Mini-Basic program is loaded into a code store, with successive commands at addresses 0, 1, 2, etc. The code pointer, CP, contains the address of the command due to be executed next.

The program's data are held in a data store of 26 cells, one cell for each variable. Figure 8.3 illustrates the code store and data store. Figure 8.4 shows how the abstract machine's state would change during the first few execution steps of the square-root program, assuming that the number read is 10.0.

We must decide how to represent Mini-Basic commands in the code store. The choices, and their consequences, are as follows:

(a) *Source text:* Each command must be scanned and parsed at run-time (i.e., every time the command is fetched from the code store).

(b) *Token sequence:* Each command must be scanned at load-time, and parsed at run-time.

(c) *AST:* All commands must be scanned and parsed at load-time.

Choice (a), illustrated in Figure 8.3, would slow the interpreter drastically. Choice (c) is better but would slow the loader somewhat. Choice (b) is a reasonable compromise, so let us adopt it here:

```
class Token {
   byte kind;
   String spelling;
}

class ScannedCommand {
   Token[] tokens;
}
```

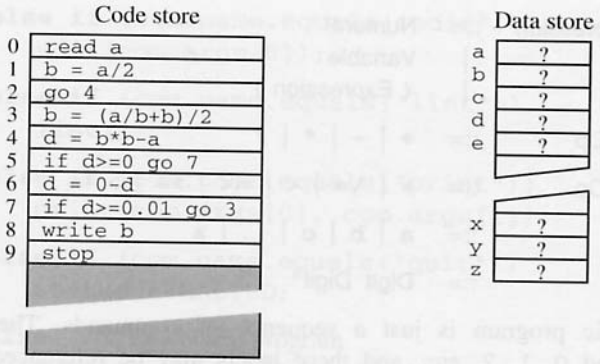

Figure 8.3 Mini-Basic abstract machine: code store and data store.

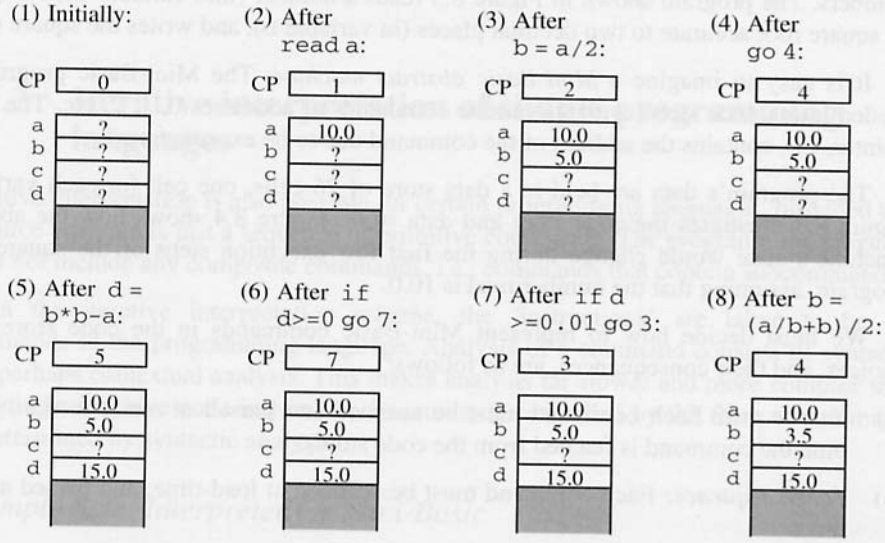

Figure 8.4 Mini-Basic abstract machine: state changes.

Although a sequence of tokens is a convenient way of representing a command in storage, it would be inconvenient for execution. Analysis of a scanned command should parse it and translate it to an internal form suitable for execution. In Mini-Basic all variables are predeclared, so there is no need to check for undeclared variables; and there is only one data type, so there is no need for type checking.

For this internal form let us adopt the command's AST. Note that only a single AST will exist at any one time, representing the current command to be executed. We introduce an abstract class for command ASTs, equipped with a method to execute a command AST:

```
public abstract class Command {
    // A Command object is an AST representing a Mini-Basic command.

    public void execute (MiniBasicState state);
        // Execute the command, using state.
}
```

And similarly for expression ASTs:

```
public abstract class Expression {
    // An Expression object is an AST representing a Mini-Basic
    //  expression.

    public float evaluate (MiniBasicState state);
        // Evaluate the expression, using state, and return its result.
}
```

Later we shall define concrete subclasses for particular forms of commands and expressions. These will implement the methods `execute` and `evaluate`, which we shall call ***interpreting methods***.

Note that we must allow the interpreting methods to access the state of the Mini-Basic abstract machine, hence their argument `state`. The following class will represent the abstract machine state:

```
public class MiniBasicState {

    public static final short CODESIZE = 4096;
    public static final short DATASIZE = 26;

    // Code store ...
    public ScannedCommand[] code =
              ScannedCommand[CODESIZE];

    // Data store ...
    public float[] data = new float[DATASIZE];

    // Registers ...
    public short CP;
    public byte status;

    public static final byte  // status values
          RUNNING = 0, HALTED = 1, FAILED = 2;
}
```

Here the code store is represented by an array of scanned commands, `code`. The data store is represented by an array of real numbers, `data`, indexed by variable addresses (using 0 for `a`, 1 for `b`, ..., 25 for `z`). The registers are represented by variables `CP` and `status`.

The following class will define the Mini-Basic interpreter:

```
public class MiniBasicInterpreter
                extends MiniBasicState {

    public void load () {
        ... // Load the program into the code store, starting at address 0.
    }

    public void run () {
        ... // Run the program in the code store, starting at address 0.
    }

    public static Command parse
                    (ScannedCommand scannedCom) {
        ... // Parse scannedCom, and return the corresponding
            // command AST.
    }
}
```

Note that we need a method, here called `parse`, to parse a scanned command an translate it to an AST.

The following method is the interpreter proper. It just fetches, analyzes, and execute the commands, one after another:

```
public void run () {

    // Initialize ...
    CP = 0; status = RUNNING;

    do {
        // Fetch the next instruction ...
        ScannedCommand scannedCom = code[CP++];

        // Analyze this instruction ...
        Command analyzedCom = parse(scannedCom);

        // Execute this instruction ...
        analyzedCom.execute((MiniBasicState) this);

    } while (status == RUNNING);
}
```

Now we must define how to represent and execute analyzed commands. We intr duce a subclass of `Command` for each form of command in Mini-Basic:

```
public class AssignCommand extends Command {
    byte V;                 // left-side variable address
    Expression E;           // right-side expression
```

```
    public void execute (MiniBasicState state) {
        state.data[V] = E.evaluate(state);
    }
}

public class GoCommand extends Command {
    short L;                // destination label

    public void execute (MiniBasicState state) {
        state.CP = L;
    }
}

public class IfCommand extends Command {
    Token R;                // relational-op
    Expression E1, E2;      // subexpressions
    short L;                // destination label

    public void execute (MiniBasicState state) {
        float num1 = E1.evaluate(state);
        float num2 = E2.evaluate(state);
        if compare(R, num1, num2)
            state.CP = L;
    }

    private static boolean compare
                (Token relop, float num1, float num2) {
        ... // Return the result of applying relational operator
            // relop to num1 and num2.
    }
}

public class StopCommand extends Command {

    public void execute (MiniBasicState state) {
        state.status = state.HALTED;
    }
}
```

(The `Command` subclasses `ReadCommand` and `WriteCommand`, and the various `Expression` subclasses, are omitted here. See Exercise 8.5.)

Study the object-oriented design of this interpreter. Once we decided to represent each command by an AST, we had to introduce the abstract class `Command`, and its subclasses `AssignCommand`, `GoCommand`, etc. We then found it convenient to equip each subclass of `Command` with an interpreting method, `execute`, allowing the interpreter to use dynamic method selection to select the right code to execute a particular command. However, these interpreting methods were outside the `MiniBasicInterpreter` class, so we had to pass the abstract machine state to them *via* their argument `state`.

The alternative to dynamic method selection would have been to make the interpreter test the subclass of each command before executing it, along the following lines:

```
// Execute this instruction ...
if (analyzedCom instanceof AssignCommand) {
   AssignCommand com = (AssignCommand) analyzedCom;
   data[com.V] = evaluate(com.E);
}
else if (analyzedCom instanceof GoCommand) {
   GoCommand com = (GoCommand) analyzedCom;
   CP = com.L;
}
else ...
```

But this would not be in the true spirit of object-oriented design!

□

8.2 Recursive interpretation

Modern programming languages are higher-level than the simple programming language of Example 8.3. In particular, commands may be composite: they may contain subcommands, subsubcommands, and so on.

It is possible to interpret higher-level programming languages. However, the iterative interpretation scheme is inadequate for such languages. Analysis of each command in the source program entails analysis of its subcommands, recursively. Likewise, execution of each command entails execution of its subcommands, recursively. Thus we are driven inexorably to a two-stage process, whereby the entire source program is analyzed before interpretation proper can begin. This gives rise to the ***recursive interpretation scheme***:

fetch and analyze the program
execute the program

where both analysis and execution are recursive.

We must decide how the program will be represented at each stage. If it is supplied in source form, 'fetch and analyze the program' must perform syntactic and contextual analysis of the program. A decorated AST is therefore a suitable representation for the result of the analysis stage. Therefore 'execute the program' will operate on the program's decorated AST.

Example 8.4 Interpreter for Mini-Triangle

Consider a recursive interpreter for the programming language Mini-Triangle of

Examples 1.3 and 1.8. Assume that the analyzed program is to be represented by a decorated AST. The source program will be subjected to syntactic and contextual analysis, and also storage allocation, before execution commences.

We must choose a representation of Mini-Triangle values. These include not only truth values and integers, but also *undefined* (which is the initial value of a variable). The following classes represent all these types of values:

```
public abstract class Value { }

public class IntValue extends Value {
   public short i;
}

public class BoolValue extends Value {
   public boolean b;
}

public class UndefinedValue extends Value {
}
```

We assume that each of these classes is equipped with a suitable constructor.

The following class will define the abstract machine state:

```
public class MiniTriangleState {

   public static final short DATASIZE = ...;

   // Code store ...
   Program program;  // decorated AST

   // Data store ...
   Value[] data = new Value[DATASIZE];

   // Register ...
   byte status;

   public static final byte  // status values
         RUNNING = 0, HALTED = 1, FAILED = 2;
}
```

Here we represent the data store, as usual, by an array. The 'code store' is just the decorated AST representing the Mini-Triangle program. We assume the class `AST`, and its subclasses `Program`, `Command`, `Expression`, `Declaration`, etc., defined in Example 4.19.

The following class will implement the Mini-Triangle interpreter. In particular, methods `fetchAnalyze` and `run` will implement the two stages of the recursive interpretation scheme.

```
public class MiniTriangleProcessor
      extends MiniTriangleState implements Visitor {
```

```
    public void fetchAnalyze () {
        ... // Load the program into the code store, after
            // subjecting it to syntactic and contextual analysis.
    }

    public void run () {
        ... // Run the program contained in the code store.
    }

    // Visitor/interpreting methods ...

    public Object visit...Command
                    (...Command com, Object arg);
        // Execute com, returning null (and ignoring arg).

    public Object visit...Expression
                    (...Expression expr, Object arg);
        // Evaluate expr, returning its result (and ignoring arg).

    public Object visit...Declaration
                    (...Declaration decl, Object arg);
        // Elaborate decl, returning null (and ignoring arg).

    // Other interpreting methods ...

    private Value fetch (Vname vname);
        // Return the value of the constant or variable vname.

    private void assign (Vname vname, Value val);
        // Assign val to the variable vname.

    // Auxiliary methods ...

    private static short valuation
                (IntegerLiteral intLit);
        // Return the value of intLit.

    private static Value applyUnary
                (Operator op, Value val);
        // Return the result of applying unary operator op to val.

    private static Value applyBinary
                (Operator op, Value val1, Value val2);
        // Return the result of applying binary operator op to val1 and val2.

    private static void callStandardProc
                (Identifier id, Value val);
        // Call the standard procedure named id, passing val as its argument.

}
```

This Mini-Triangle processor is a visitor object (see Section 5.3.2), in which the visitor methods act as interpreting methods.

The visitor/interpreting methods for commands are implemented as follows:

```
public Object visitAssignCommand
                (AssignCommand com, Object arg) {
   Value val = (Value) com.E.visit(this, null);
   assign(com.V, val);
   return null;
}

public Object visitCallCommand
                (CallCommand com, Object arg) {
   Value val = (Value) com.E.visit(this, null);
   callStandardProc(com.I, val);
   return null;
}

public Object visitSequentialCommand
               (SequentialCommand com, Object arg) {
   com.C1.visit(this, null);
   com.C2.visit(this, null);
   return null;
}

public Object visitIfCommand
                (IfCommand com, Object arg) {
   BoolValue val = (BoolValue) com.E.visit(this, null)
   if (val.b) com.C1.visit(this, null);
   else       com.C2.visit(this, null);
   return null;
}

public Object visitWhileCommand
                (WhileCommand com, Object arg) {
   for (;;) {
      BoolValue val = (BoolValue)
            com.E.visit(this, null);
      if (! val.b) break;
      com.C.visit(this, null);
   }
   return null;
}

public Object visitLetCommand
                (LetCommand com, Object arg) {
   com.D.visit(this, null);
   com.C.visit(this, null);
   return null;
}
```

The visitor/interpreting methods for expressions are implemented as follows:

```
public Object visitIntegerExpression
                (IntegerExpression expr, Object arg)
   return new IntValue(valuation(expr.IL));
}

public Object visitVnameExpression
                (VnameExpression expr, Object arg) {
   return fetch(expr.V);
}

public Object visitUnaryExpression
                (UnaryExpression expr, Object arg) {
   Value val = (Value) expr.E.visit(this, null);
   return applyUnary(expr.O, val);
}

public Object visitBinaryExpression
                (BinaryExpresion expr, Object arg) {
   Value val1 = (Value) expr.E1.visit(this, null);
   Value val2 = (Value) expr.E2.visit(this, null);
   return applyBinary(expr.O, val1, val2);
}
```

The visitor/interpreting methods for declarations are implemented as follows:

```
public Object visitConstDeclaration
                (ConstDeclaration decl, Object arg)
   KnownAddress entity = (KnownAddress) decl.entity;
   Value val = (Value) decl.E.visit(this, null);
   data[entity.address] = val;
   return null;
}

public Object visitVarDeclaration
                (VarDeclaration decl, Object arg) {
   KnownAddress entity = (KnownAddress) decl.entity;
   data[entity.address] = new UndefinedValue();
   return null;
}

public Object visitSequentialDeclaration
               (SequentialDeclaration decl,
                Object arg) {
   decl.D1.visit(this, null);
   decl.D2.visit(this, null);
   return null;
}
```

Finally, the auxiliary methods for value-or-variable-names are implemented as follows:

```
private Value fetch (Vname vname) {
    KnownAddress entity =
          (KnownAddress) vname.visit(this, null);
    return data[entity.address];
}

private void assign (Vname vname, Value val) {
    KnownAddress entity =
          (KnownAddress) vname.visit(this, null);
    data[entity.address] = val;
}
```

To fetch and analyze a Mini-Triangle program, we need the following:

- A *parser*. The class `Parser` of Example 4.12 exports a method `parse`, which parses a Mini-Triangle source program and returns the corresponding AST.
- A *contextual analyzer*. The class `Checker` of Example 5.11 exports a method `check`, which performs identification and type checking on a given AST and decorates it accordingly.
- A *static storage allocator*. Let us assume a class `StorageAllocator` that exports a method `allocateAddresses`. This method visits each constant and variable declaration in a given AST, allocates a suitable address to the declared constant or variable, and records that address in a `KnownAddress` entity description attached to the declaration. (See Example 7.13.)

```
public void fetchAnalyze () {
    Parser parser = new Parser(...);
    Checker checker = new Checker(...);
    StorageAllocator allocator = new StorageAllocator();
    program = parser.parse();
    checker.check(program);
    allocator.allocateAddresses(program);
}
```

To run the program, we simply visit the program command:

```
public void run () {
    program.C.visit(this, null);
}
```

This design for a Mini-Triangle recursive interpreter reuses the visitor design pattern already exploited in our Mini-Triangle contextual analyzer (Example 5.11) and code generator (Example 7.8). This design allows the interpreting methods to be located in the `MiniTriangleInterpreter` class, where they have direct access to the abstract machine state.

A design similar to that of Example 8.3 would be a reasonable alternative. That would entail equipping each `Command` subclass with an `execute` method, each `Expression` subclass with an `evaluate` method, each `Declaration` subclass with an `elaborate` method, and so on. Each of these interpreting methods would be passed the abstract machine state as an argument.

□

The kind of interpreter we have just illustrated analyzes the entire source program before execution commences. Thus it forgoes one of the usual advantages of interpretation, that is immediacy. This explains why recursive interpretation is not generally used for higher-level programming languages. For such languages, a better alternative is compilation of source programs to a simple intermediate language, followed by iterative interpretation of the intermediate language, as outlined in Section 2.4.

Some notable exceptions to the general rule, namely Lisp and Prolog, will be discussed in Section 8.4.

8.3 Case study: the TAM interpreter

The abstract machine TAM was outlined in Section 6.8, and is fully described in Appendix C. It is the target machine of the Triangle compiler.

TAM is implemented by an interpreter that is in most respects similar to other machine-code interpreters, such as that of Example 8.1. But of course it is more sophisticated, since TAM directly supports many features of high-level languages.

The `TAM` package includes the classes `Instruction`, `State`, and `Interpreter` outlined here.

The following class represents TAM instructions (Figure C.5):

```
public class Instruction {
    public byte  op;      // op-code (0 .. 15), from Table C.2
    public byte  r;       // register field (0 .. 15), from Table C.1
    public byte  n;       // length field (0 .. 255)
    public short d;       // operand field (–32767 .. +32767)
    ...
}
```

The following class represents the TAM state. The code and data stores are represented as usual by arrays. The registers are represented by variables (with read-only registers declared as **`final`**).

```
public class State {

    public static final short CODESIZE = ...;
    public static final short DATASIZE = ...;
```

```
    // Code store ...
    public Instruction[] code =
          new Instruction[CODESIZE];

    // Data store ...
    public short[] data = new short[DATASIZE];

    // Registers ...
    public short final   CB = 0;
    public short         CT;
    public short final   PB = CODESIZE;
    public short final   PT = CODESIZE + 28;
    public short final   SB = 0;
    public short         ST;
    public short final   HB = DATASIZE;
    public short         HT;
    public short         LB;
    public short         CP;
    public byte          status;

    public static final byte  // status values
          RUNNING = 0, HALTED = 1, FAILED = 2;
}
```

The following class implements the TAM interpreter proper:

```
public class Interpreter extends State {

    public void loadProgram () {
        ... // Load the program into the code store, starting at address 0.
        ... // Set CT to the address of the last instruction + 1.
    }

    public void runProgram () {
        ... // Run the program contained in the code store,
            // starting at address 0.
    }

}
```

The interpreter proper is as follows. Its main control structure is a switch-statem within a loop. There is one case for each of the fifteen valid op-codes, and a default c for invalid op-codes:

```
public void runProgram () {

    // Initialize ...
    ST = SB;  HT = HB;  LB = SB;  CP = CB;
    status = RUNNING;
```

```
        do {
            // Fetch the next instruction ...
            Instruction instr = code[CP++];

            // Analyze this instruction ...
            byte  op = instr.op;
            byte  r  = instr.r;
            byte  n  = instr.n;
            short d  = instr.d;

            // Execute this instruction ...
            switch (op) {

               case LOADop:   ...
               case LOADAop:  ...
               case LOADIop:  ...
               case LOADLop:  ...
               case STOREop:  ...
               case STOREIop: ...
               case CALLop:   ...
               case CALLIop:  ...
               case RETURNop: ...
               case PUSHop:   ...
               case POPop:    ...
               case JUMPop:   ...
               case JUMPIFop: ...

               case HALTop:   status = HALTED;  break;

               default:       status = FAILED;
            }

        } while (status == RUNNING);
    }
```

The fact that TAM is a stack machine gives rise to many differences in detail from an interpreter for a register machine. Load instructions push values on to the stack, and store instructions pop values off the stack. For example, the TAM `LOADL` instruction is interpreted as follows:

```
case LOADLop:
    data[ST++] = d;
    break;
```

(Register ST points to the word immediately *above* the stack top, as shown in Figure C.1.)

Further differences arise from the special design features of TAM (outlined in Section 6.8).

Addressing and registers

The operand of a LOAD, LOADA, or STORE instruction is of the form '*d*[*r*]', where *r* is usually a display register, and *d* is a constant displacement. The displacement *d* is added to the current content of register *r*.

The display registers allow addressing of global variables (using SB), local variables (using LB), and nonlocal variables (using L1, L2, ...). The latter registers are related to LB by the invariants L1 = *content*(LB), L2 = *content*(*content*(LB)), and so on – see (6.25–27) in Section 6.4.2.

As explained in Section 6.8, it is not really worthwhile to have separate registers for access to nonlocal variables. The cost of updating them (on every routine call and return) outweighs the benefit of having them immediately available to compute the addresses of nonlocal variables. In the TAM interpreter, therefore, L1, L2, etc., are only *pseudo-registers*: their values are computed only when needed, using the above invariants. This is captured by the following auxiliary method in the interpreter:

```
private static short relative (short d, byte r) {
    // Return the address defined by displacement d relative to register r.
    switch (r) {
        ...
        case SBr:   return d + SB;
        case LBr:   return d + LB;
        case L1r:   return d + data[LB];
        case L2r:   return d + data[data[LB]];
        ...
    }
}
```

For example, the LOAD and STORE instructions (on the simplifying assumption that the length field n is 1) would be interpreted as follows:

```
case LOADop: {
    short addr = relative(d, r);
    data[ST++] = data[addr];
    break;
}

case STOREop: {
    short addr = relative(d, r);
    data[addr] = data[--ST];
    break;
}
```

The operand of a CALL, JUMP, or JUMPIF instruction is also of the form '*d*[*r*]', where *r* is generally CB or PB, and *d* is a constant displacement. As usual, the displacement *d* is added to the content of register *r*. The auxiliary method relative also handles these cases.

Primitive routines

Each primitive routine (such as *mult*, *lt*, or *not*) has a designated address within the code store's primitives segment, which is delimited by registers PB and PT. (See Figure C.1.) Thus the interpreter traps any call to an address within the primitives segment:

```
case CALLop: {
    short addr = relative(d, r);
    if (addr >= PB)
        ... // Execute the primitive operation at address addr.
    else
        ... // Call the code routine at address addr.
    break;
}
```

The interpreter performs the appropriate primitive operation itself. For example, *mult*, *lt*, and *not* are interpreted as follows:

```
case mult: {
    --ST;
    data[ST-1] *= data[ST];
    break;
}

case lt: {
    --ST;
    data[ST-1] = (data[ST-1] < data[ST]) ? 1 : 0;
    break;
}

case not: {
    data[ST-1] = 1 - data[ST-1];   // replacing 0 by 1, or 1 by 0
    break;
}
```

8.4 Further reading

Interpretation is still popular despite the obvious performance problems. Indeed, since an interpreter is easier to implement than a compiler, many programming languages rely on an interpreter for their first implementation (see Section 9.1.3).

The original Basic was one of the few 'high-level' programming languages for which interpretation was normal. A typical Basic language processor allowed programs to be entered, edited, and executed incrementally. Such a language processor could run on a microcomputer with very limited storage, hence its popularity in the early days of microcomputers. But this was possible only because the language was very primitive

indeed. Its control structures were more typical of a low-level language, making it unattractive for serious programmers. More recently, 'structured' dialects of Basic have become more popular, and compilation has become an alternative to interpretation.

Recursive interpretation is less common. However, this form of interpretation has long been associated with Lisp (McCarthy *et al.* 1965). A Lisp program is not just represented by a tree: it *is* a tree! Several features of the language – dynamic binding, dynamic typing, and the possibility of manufacturing extra program code at run-time – make interpretation of Lisp much more suitable than compilation. A description of a Lisp interpreter may be found in McCarthy *et al.* (1965). Lisp has always had a devoted band of followers, but not all are prepared to tolerate slow execution. A more recent successful dialect, Scheme (Kelsey *et al.* 1998), has discarded Lisp's problematic features in order to make compilation feasible.

It is noteworthy that two popular programming languages, Basic and Lisp, both suitable for interpretation but otherwise utterly different, have evolved along somewhat parallel lines, spawning structured dialects suitable for compilation!

Another example of a high-level language suitable for interpretation is Prolog. This language has a very simple syntax, a program being a flat collection of clauses, and it has no scope rules and few type rules to worry about. Interpretation is almost forced by the ability of a program to modify itself by adding and deleting clauses at run-time.

Exercises

8.1 Make the Hypo interpreter of Example 8.1 detect the following exceptional conditions, and set the status register accordingly:

(a) overflow;
(b) invalid instruction address;
(c) invalid data address.

(Assume that Hypo may have less than 4096 words of code store and less than 4096 words of data store, thus making conditions (b) and (c) possible.)

8.2 Make the Hypo interpreter of Example 8.1 display a summary of the machine state after executing each instruction. Display the contents of ACC and CP, the instruction just executed, and a selected portion of the data store.

8.3* Make the Hypo interpreter of Example 8.1 into an interactive debugger. Provide the following facilities: (a) execute the next instruction only (*single-step*); (b) set or remove a breakpoint at a given instruction; (c) execute instructions until the next breakpoint; (d) display the contents of ACC and CP; (e) display a selected portion of the data store; (f) terminate execution.

8.4** Write an emulator for a real machine with which you are familiar.

8.5 Fill in the missing details of the Mini-Basic interpreter of Example 8.3.

8.6 The Mini-Basic interpreter of Example 8.3 represents each stored command as a token sequence.

(a) Discuss in detail the advantages and disadvantages of this choice of representation, and of the other possible representations. Assume a typical Basic-style language processor, in which the user may interactively enter, edit, or delete any command.

(b) How does the choice of representation influence the method `analyze`, the method that loads a command into the code store, and the method that edits a stored command?

8.7* Extend the Mini-Basic interpreter of Example 8.3 to deal with the following extensions:

```
Command ::= ...
          | perform Label to Label
          | while Expression Relational-Op Expression
              do Command
```

The effect of '`perform` L_1 `to` L_2' is to execute the commands labeled L_1 through L_2 (where L_1 must not follow L_2). This is a kind of parameterless procedure call.

The effect of '`while` E_1 R E_2 `do` C' is to repeat the subcommand C as long as the comparison 'E_1 R E_2' yields *true*. C is restricted to be a primitive command (i.e., it may not itself be a while-command).

Do these extensions lead you to reconsider the choice of representation for a stored command (Exercise 8.6)?

8.8* Extend the Mini-Shell interpreter of Example 8.2 to deal with a new command, `call`. This takes a single argument, a filename. The named file is expected to contain a Mini-Shell script, whose commands are to be executed one after another.

8.9* Suppose that Mini-Basic (Example 8.3) is to be replaced by a structured dialect with similar expressive power. The syntax of commands is to become:

```
Command ::= Variable = Expression
          | read Variable
          | write Variable
          | if Expression Relational-Op Expression
              then Command* else Command* end
          | while Expression Relational-Op Expression
              do Command* end
          | stop
```

Expressions, operators, and variables are unchanged, but labels are removed. Write a recursive interpreter for this structured dialect.

8.10** The TAM interpreter (Section 8.3) sacrifices efficiency for clarity. For example, the fetch/analyze/execute cycle could be combined and replaced by a single switch-statement of the form:

```
switch ((instr = code[CP++]).op) {
  case LOADop:    ...
  ...
}
```

Another efficiency gain could be achieved by holding the top one or two stack elements in simple variables, and possibly avoiding the unnecessary updating of the stack pointer during a long sequence of arithmetic operations. (This is effectively turning TAM into a register machine!)

Consider these and other possible improvements to the TAM interpreter, and develop a more efficient implementation. Compare your version with the original TAM interpreter, and measure the performance gain.