Chapter 7 Interpreting a Program by Performing a Treewalk

If we wish to interpret a program with control structures, such as loops, we need to build an abstract syntax tree, then perform a treewalk to "interpret" the program.

Interpreting Control Structures

The following example represents an interpreter for a language with assignment statements, if statements, while statements, and compound statements. Expressions can involve identifiers, constants, and operators. (Refer INTERP3.)

The lexical analyser

The lexical analyser is fairly standard.

```
package grammar;

import java.io.*;
import java_cup.runtime.*;

%%

%public
%type          Symbol
%char

%{
     public Symbol token( int tokenType ) {
          System.err.println( "Obtain token "
               + sym.terminal_name( tokenType )
               + " \"" + yytext() + "\"" );
          return new Symbol( tokenType, yychar,
               yychar + yytext().length(), yytext() );
          }

%}

number         =    [0-9]+
ident          =    [A-Za-z][A-Za-z0-9]*
space          =    [\ \t]
newline        =    \r|\n|\r\n

%%

"="            { return token( sym.ASSIGN ); }
"+"            { return token( sym.PLUS ); }
"-"            { return token( sym.MINUS ); }
"*"            { return token( sym.TIMES ); }
"/"            { return token( sym.DIVIDE ); }
"("            { return token( sym.LEFT ); }
")"            { return token( sym.RIGHT ); }
"<"            { return token( sym.LT ); }
"<="           { return token( sym.LE ); }
">"            { return token( sym.GT ); }
">="           { return token( sym.GE ); }
"=="           { return token( sym.EQ ); }
"!="           { return token( sym.NE ); }
"if"           { return token( sym.IF ); }
"then"         { return token( sym.THEN ); }
```

```
"else"          { return token( sym.ELSE ); }
"while"         { return token( sym.WHILE ); }
"do"            { return token( sym.DO ); }
"{"             { return token( sym.LEFTCURLY ); }
"}"             { return token( sym.RIGHTCURLY ); }
";"             { return token( sym.SEMICOLON ); }
{newline}       { }
{space}         { }

{number}        { return token( sym.NUMBER ); }
{ident}         { return token( sym.IDENT ); }

.               { return token( sym.error ); }

<<EOF>>         { return token( sym.EOF ); }
```

The parser

The parser builds an abstract syntax tree. As in C and Java, semicolons are used to terminate simple statements, not newlines. It has a single ambiguity, related to the two kinds of if statement, which are resolved in the natural way. Note the use of the error symbol, to perform error recovery.

```
package grammar;

import node.*;
import node.stmtNode.*;
import node.exprNode.*;
import node.exprNode.prefixNode.*;
import node.exprNode.valueNode.*;
import node.exprNode.binaryNode.*;

import java.io.*;
import java_cup.runtime.*;

 parser code
     {:
     private Yylex lexer;
     private File file;

     public parser( File file ) {
          this();
          this.file = file;
          try {
               lexer = new Yylex( new FileReader( file ) );
               }
          catch ( IOException exception ) {
               throw new Error( "Unable to open file \"" + file + "\"" );
               }
          }
...
     :};

scan with
     {:
     return  lexer.yylex();
     :};

terminal LEFT, RIGHT, PLUS, MINUS, TIMES, DIVIDE, ASSIGN, SEMICOLON;
```

```
terminal LT, LE, GT, GE, EQ, NE, IF, THEN, ELSE, WHILE, DO, LEFTCURLY,
RIGHTCURLY;
terminal String NUMBER;
terminal String IDENT;

nonterminal StmtListNode StmtList;
nonterminal StmtNode Stmt;
nonterminal ExprNode BoolExpr, Expr, Term, Factor;

start with StmtList;

StmtList::=
         {:
         RESULT = new StmtListNode();
         :}
     |
         StmtList:stmtList Stmt:stmt
         {:
         stmtList.addElement( stmt );
         RESULT = stmtList;
         :}
     ;

Stmt::=
         IDENT:ident ASSIGN Expr:expr SEMICOLON
         {:
         RESULT = new AssignStmtNode( ident, expr );
         :}
     |
         IF BoolExpr:expr THEN Stmt:stmt1 ELSE Stmt:stmt2
         {:
         RESULT = new IfThenElseStmtNode( expr, stmt1, stmt2 );
         :}
     |
         IF BoolExpr:expr THEN Stmt:stmt1
         {:
         RESULT = new IfThenStmtNode( expr, stmt1 );
         :}
     |
         WHILE BoolExpr:expr DO Stmt:stmt1
         {:
         RESULT = new WhileStmtNode( expr, stmt1 );
         :}
     |
         LEFTCURLY StmtList:stmtList RIGHTCURLY
         {:
         RESULT = new CompoundStmtNode( stmtList );
         :}
     |
         error SEMICOLON
         {:
         RESULT = new ErrorStmtNode();
         :}
     |
         error RIGHTCURLY
         {:
         RESULT = new ErrorStmtNode();
         :}
     ;
```

```
BoolExpr::=
         Expr:expr1 LT Expr:expr2
         {:
         RESULT = new LessThanNode( expr1, expr2 );
         :}
     |
         Expr:expr1 LE Expr:expr2
         {:
         RESULT = new LessEqualNode( expr1, expr2 );
         :}
     |
         Expr:expr1 GT Expr:expr2
         {:
         RESULT = new GreaterThanNode( expr1, expr2 );
         :}
     |
         Expr:expr1 GE Expr:expr2
         {:
         RESULT = new GreaterEqualNode( expr1, expr2 );
         :}
     |
         Expr:expr1 EQ Expr:expr2
         {:
         RESULT = new EqualNode( expr1, expr2 );
         :}
     |
         Expr:expr1 NE Expr:expr2
         {:
         RESULT = new NotEqualNode( expr1, expr2 );
         :}
     ;

Expr::=
         Expr:expr PLUS Term:term
         {:
         RESULT = new PlusNode( expr, term );
         :}
     |
         Expr:expr MINUS Term:term
         {:
         RESULT = new MinusNode( expr, term );
         :}
     |
         MINUS Term:term
         {:
         RESULT = new NegateNode( term );
         :}
     |
         Term:term
         {:
         RESULT = term;
         :}
     ;
```

```
Term::=
         Term:term TIMES Factor:factor
         {:
         RESULT = new TimesNode( term, factor );
         :}
     |
         Term:term DIVIDE Factor:factor
         {:
         RESULT = new DivideNode( term, factor );
         :}
     |
         Factor:factor
         {:
         RESULT = factor;
         :}
     ;

Factor::=
         LEFT Expr:expr RIGHT
         {:
         RESULT = expr;
         :}
     |
         NUMBER:value
         {:
         RESULT = new NumberNode( new Integer( value ) );
         :}
     |
         IDENT:ident
         {:
         RESULT = new IdentNode( ident );
         :}
     ;
```

The Main class

The Main class creates the parser, performs a parse and builds an abstract syntax tree. It then performs two treewalks: one to reprint the tree, and another to “execute” the program.

```
import java.io.*;
import java_cup.runtime.*;
import runEnv.*;
import node.*;
import node.stmtNode.*;
import grammar.*;
import text.*;

public class Main {

     public static void main( String[] argv ) {
          String dirName = null;

          try {
               for ( int i = 0; i < argv.length; i++ ) {
                    if ( argv[ i ].equals( "-debug" ) ) {
                         Print.DEBUG = true;
                         }
                    else if ( argv[ i ].equals( "-dir" ) ) {
                         i++;
                         if ( i >= argv.length )
                              throw new Error( "Missing directory name" );
                         dirName = argv[ i ];
```

```
                    }
                else {
                    throw new Error(
                        "Usage: java Main [-debug] -dir directory" );
                    }
                }

            if ( dirName == null )
                throw new Error( "Directory not specified" );

            System.setErr( new PrintStream( new FileOutputStream(
                new File( dirName, "program.parse" ) ) ) );
            Print.setError( new File( dirName, "program.err" ) );
            Print.setReprint( new File( dirName, "program.print" ) );
            Print.setInterp( new File( dirName, "program.out" ) );

            parser p = new parser( new File( dirName, "program.in" ) );
            StmtListNode program = ( StmtListNode ) p.parse().value;
            Print.error().println( "Reprinting ... " );
            Print.reprint().println( program );
            Print.error().println( "Evaluate ... " );
            program.eval( new RunEnv() );

            }
        catch ( Exception e ) {
            Print.error().println( "Exception at " );
            e.printStackTrace();
            }
        }
    }
```

Reprinting the program

Each node has a toString() method. The toString method returns a string containing formatting directives. %n means go to a newline at the same indenting level, %+ and %- increase and decrease the indenting level.

```
package text;

import java.io.*;

public class FormattedOutput {

    private static int INC = 4;
    private int indent;
    private int column;

    private PrintWriter printWriter;

    public FormattedOutput( File file ) throws Error {
        try {
            OutputStream outputStream = new FileOutputStream( file );
            printWriter = new PrintWriter( outputStream );
            }
        catch ( IOException exception ) {
            throw new Error( "Unable to open file " + file );
            }
        }
```

```
    public void println() {
        printWriter.println();
        printWriter.flush();
        column = 0;
        }

    public void print( char c ) {
        while ( column < indent ) {
            printWriter.print( ' ' );
            column++;
            }
        printWriter.print( c );
        column++;
        }

    private void print( String s ) {
        int i = 0;
        while ( i < s.length() ) {
            if ( s.charAt( i ) == '%' ) {
                i++;
                switch ( s.charAt( i ) ) {
                    case '+':
                        indent += INC;
                        break;
                    case '-':
                        indent -= INC;
                        break;
                    case 'n':
                        println();
                        break;
                    default:
                        print( s.charAt( i ) );
                    }
                i++;
                }
            else {
                print( s.charAt( i++ ) );
                }
            }
        }
...
    }
```

The run-time environment

The node classes have an eval method to evaluate the construct. All eval methods take a run-time environment as a parameter.

The RunEnv class contains information mapping identifiers to values. For this interpreter, it is implemented by a hash table, but in a more complex program with local blocks, it would need to have a much more complicated structure. For this interpreter, all identifiers represent integer variables.

```
public class RunEnv {
     private Hashtable table = new Hashtable();

     public void put( String ident, int value ) {
          table.put( ident, new Integer( value ) );
          }

     public int get( String ident ) {
          Integer value = ( Integer ) table.get( ident );
          if ( value != null )
               return value.intValue();
          else
               return 0;
          }
     }
```

The eval method for expressions returns an object as a result. For arithmetic expressions, it is of type Integer, for Boolean expressions, it is of type Boolean, etc.

The Cast support class is used to check that values are of an appropriate type.

```
public class Cast {

     public static boolean booleanValue( Object object ) {
          if ( object instanceof Boolean )
               return ( ( Boolean ) object ).booleanValue();
          else
               throw new Error( "Can't cast to Boolean" );
          }

     public static int intValue( Object object ) {
          if ( object instanceof Integer )
               return ( ( Integer ) object ).intValue();
          else
               throw new Error( "Can't cast to Integer" );
          }

     public static String stringValue( Object object ) {
          if ( object instanceof String )
               return ( String ) object;
          else
               throw new Error( "Can't cast to String" );
          }
     }
```

The Node classes

For operators, we have to evaluate the operands, then compute and return the result.

```
public class PlusNode extends ArithNode {

     public PlusNode( ExprNode left, ExprNode right ) {
          super( left, right );
          precedence = PREC_ADD;
          operator = "+";
          }

     public Object eval( RunEnv runEnv ) {
          int leftValue = Cast.intValue( left.eval( runEnv ) );
          int rightValue = Cast.intValue( right.eval( runEnv ) );
          return new Integer( leftValue + rightValue );
          }
     }
```

```
public class GreaterEqualNode extends RelationNode {

     public GreaterEqualNode( ExprNode left, ExprNode right ) {
          super( left, right );
          precedence = PREC_REL;
          operator = ">=";
          }

     public Object eval( RunEnv runEnv ) {
          int leftValue = Cast.intValue( left.eval( runEnv ) );
          int rightValue = Cast.intValue( right.eval( runEnv ) );
          return new Boolean( leftValue >= rightValue );
          }

     }
```

The node for identifiers has to search the run-time environment to obtain the value of the variable.

```
public class IdentNode extends ExprNode {

     private String ident;

     public IdentNode( String ident ) {
          this.ident = ident;
          precedence = PREC_PRIMARY;
          }

     public String toString() {
          return ident;
          }

     public Object eval( RunEnv runEnv ) {
          return new Integer( runEnv.get( ident ) );
          }
     }
```

The assignment statement has to store the mapping of the identifier to the value in the run-time environment.

```
public class AssignStmtNode extends StmtNode {

     private String ident;
     private ExprNode expr;

     public AssignStmtNode( String ident, ExprNode expr ) {
          this.ident = ident;
          this.expr = expr;
          }

     public String toString() {
          return ident + " = " + expr +";";
          }

     public void eval( RunEnv runEnv ) {
          Object value = expr.eval( runEnv );
          runEnv.put( ident, Cast.intValue( value ) );
          Print.interp().println( ident + " = " + value );
          }
     }
```

The code for evaluating a statement sequence invokes the code for each substatement.

```
public class StmtListNode {

     private Vector list = new Vector();

     public StmtListNode() {
          }

     public void addElement( StmtNode node ) {
          list.addElement( node );
          }

     public String toString() {
          String result = "";
          for ( int i = 0; i < list.size(); i++ ) {
               StmtNode stmt = ( StmtNode ) list.elementAt( i );
               result += "%n" + stmt;
               }
          return result;
          }

     public void eval( RunEnv runEnv ) {
          for ( int i = 0; i < list.size(); i++ ) {
               StmtNode stmt = ( StmtNode ) list.elementAt( i );
               try {
                    stmt.eval( runEnv );
                    }
               catch ( Error exception ) {
                    Print.error().println(
                         "Runtime Error " + exception.getMessage() );
                    }
               catch ( RuntimeException exception ) {
                    Print.error().println(
                         "Runtime Exception " + exception.getMessage() );
                    }
               }
          }
     }
```

Evaluating a compound statement just involves evaluating the substatements.

```
public class CompoundStmtNode extends StmtNode {

     private StmtListNode stmtList;

     public CompoundStmtNode( StmtListNode stmtList ) {
          this.stmtList = stmtList;
          }

     public String toString() {
          return "{%+" + stmtList + "%-%n}";
          }

     public void eval( RunEnv runEnv ) {
          stmtList.eval( runEnv );
          }
     }
```

Evaluating a control statement is similar. For loops, the condition and substatement may need to be evaluated multiple times. For if statements, only one of the substatements will be evaluated. Evaluating a substatement will have side effects, in that it will modify the values of variables.

```
public class WhileStmtNode extends StmtNode {

     private ExprNode cond;
     private StmtNode stmt;

     public WhileStmtNode( ExprNode cond, StmtNode stmt ) {
          this.cond = cond;
          this.stmt = stmt;
          }

     public String toString() {
          return "while " + cond + " do%+%n" + stmt + "%-";
          }

     public void eval( RunEnv runEnv ) {
          while ( Cast.booleanValue( cond.eval( runEnv ) ) )
               stmt.eval( runEnv );
          }
     }

public class IfThenStmtNode extends StmtNode {

     private ExprNode cond;
     private StmtNode stmt;

     public IfThenStmtNode( ExprNode cond, StmtNode stmt ) {
          this.cond = cond;
          this.stmt = stmt;
          }

     public String toString() {
          return "if " + cond + " then%+%n" + stmt + "%-";
          }

     public void eval( RunEnv runEnv ) {
          if ( Cast.booleanValue( cond.eval( runEnv ) ) )
               stmt.eval( runEnv );
          }
     }
```

```
public class IfThenElseStmtNode extends StmtNode {

     private ExprNode cond;
     private StmtNode stmt1;
     private StmtNode stmt2;

     public IfThenElseStmtNode(
          ExprNode cond,
          StmtNode stmt1,
          StmtNode stmt2 ) {
          this.cond = cond;
          this.stmt1 = stmt1;
          this.stmt2 = stmt2;
          }

     public String toString() {
          return "if " + cond + " then%+%n"
               + stmt1 + "%-else%+%n" + stmt2 + "%-";
          }

     public void eval( RunEnv runEnv ) {
          if ( Cast.booleanValue( cond.eval( runEnv ) ) )
               stmt1.eval( runEnv );
          else
               stmt2.eval( runEnv );
          }
     }
```