Chapter 14 Tree Walking and the Interpreter

Representation of Nodes, and Treewalking

Most nodes have:

- Fields corresponding to their children.
- A constructor, to build the node from its children.
- A toString() method, for reprinting.
- A field called "env" representing its compile-time environment.
- A genEnv() method that takes a section kind (class, method, variable, or parameter) and compile-time environment as a parameter, which it passes down to its children. When the child corresponds to a local block, a new compile-time environment is created and passed down. When a declaration is processed, the declaration is added to the compile-time environment. However, lookup of type identifiers is usually left to the next pass.
- A getType() method that looks up type identifiers in the environment, and builds a data structure for each type construct, or a setType() method that sets the type for each declaration.
- A field called "type" representing the type of the construct, for expressions.
- A checkType() method, to perform type checking of expressions.
- A genOffset() method, to determine the offset of a declaration from the base of the block of memory the value is stored in at run time.
- An eval() or evalAddr() method, to evaluate the construct corresponding to the node, that takes a run-time environment as a parameter. This method builds local run-time environments for method invocations. For constructs such as loops, it may evaluate a child node multiple times, and for constructs such as if statements, it will only evaluate one child.
- A genCode() (declarations and statements), evalCode() (arithmetic expressions), evalAddrCode() (variables), or boolCode() (Boolean expressions) method to generate assembly language for the construct corresponding to the node.

Note that usually either the eval() method or the genCode() method is invoked, depending on whether we wish to interpret the program, or generate assembly language.

Reprinting of Nodes of the Tree

Each node has a toString() method to convert it to a String, that can then be printed.

The string generated includes formatting directives, %n, %+, %-, %%, to generate a line break, increase the indenting level, decrease the indenting level, or print a %. The FormattedOutput class is used to perform formatting.

```
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++ ) );
                  }
             }
        }
```

For example, the WhileStmtNode statement has a toString() method that indents the substatements.

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

Declaration/Statement Lists

For sequences of declarations and statements, the method for each pass just invokes the corresponding method for each child.

```
public class DeclStmtListNode extends Node {

     private int section;
     public int section() { return section; }

     private Vector list = new Vector();

     public DeclStmtListNode() {
          }

     public DeclStmtListNode( DeclStmtNode node ) {
          list.addElement( node );
          }

     public int size() { return list.size(); }

     public DeclStmtNode elementAt( int i ) {
          return ( DeclStmtNode ) list.elementAt( i );
          }

     public void prependElement( DeclStmtNode node ) {
          list.insertElementAt( node, 0 );
          }

     public void addElement( DeclStmtNode node ) {
          list.addElement( node );
          }
```

```
public String toString() {
     String result = "";
     for ( int i = 0; i < size(); i++ ) {
          DeclStmtNode declStmt = null;
          try {
               declStmt = elementAt( i );
               if ( i > 0 )
                    result += "%n";
               result += declStmt;
               }
          catch ( Error exception ) {
               Print.error().println( exception.getMessage() );
               list.setElementAt( new ErrorDeclStmtNode(), i );
               }
          }
     return result;
     }

public void genEnv( int section, Env env ) {
     super.genEnv( env );
     this.section = section;
     for ( int i = 0; i < size(); i++ ) {
          DeclStmtNode declStmt = null;
          try {
               declStmt = elementAt( i );
               declStmt.genEnv( section, env );
               }
          catch ( Error exception ) {
               Print.error().println( "Error: " + exception.getMessage()
                    + " at%+%n" + declStmt + "%-" );
               list.setElementAt( new ErrorDeclStmtNode(), i );
               }
          }
     }

public void setType() {
     for ( int i = 0; i < size(); i++ ) {
          DeclStmtNode declStmt = null;
          try {
               declStmt = elementAt( i );
               declStmt.setType();
               }
          catch ( Error exception ) {
               Print.error().println( "Error: " + exception.getMessage()
                    + " at%+%n" + declStmt + "%-" );
               list.setElementAt( new ErrorDeclStmtNode(), i );
               }
          }
     }

public void checkType() {
     for ( int i = 0; i < size(); i++ ) {
          DeclStmtNode declStmt = null;
          try {
               declStmt = elementAt( i );
               declStmt.checkType();
               }
          catch ( Error exception ) {
               Print.error().println( "Error: " + exception.getMessage()
                    + " at%+%n" + declStmt + "%-" );
```

```
                    list.setElementAt( new ErrorDeclStmtNode(), i );
                    }
               }
          }

     public void genOffset() {
          for ( int i = 0; i < size(); i++ ) {
               DeclStmtNode declStmt = elementAt( i );
               declStmt.genOffset();
               }
          }

     public void eval( RunEnv runEnv ) throws ReturnException {
          for ( int i = 0; i < size(); i++ ) {
               DeclStmtNode declStmt = elementAt( i );
               declStmt.eval( runEnv );
               }
          }

     public void genCode() {
          for ( int i = 0; i < size(); i++ ) {
               DeclStmtNode declStmt = elementAt( i );
               declStmt.genCode();
               }
          }
     }
```

Types

When processing declarations, we refer to types. We have to convert the type construct into a type. This is done by the getType() method.

```
public abstract class TypeNode extends Node {
     public abstract Type getType();
     }

public abstract class BasicTypeNode extends TypeNode {
     }

public class IntTypeNode extends BasicTypeNode {

     public IntTypeNode() {
          }

     public String toString() {
          return "int";
          }

     public Type getType() {
          return IntType.type;
          }
     }

public class ArrayTypeNode extends TypeNode {

     private int size;
     private TypeNode subTypeNode;

     public ArrayTypeNode( int size, TypeNode subTypeNode ) {
          this.size = size;
          this.subTypeNode = subTypeNode;
          }
```

```
    public String toString() {
        return "[" + size + " ]" + subTypeNode;
        }

    public void genEnv( Env env ) {
        subTypeNode.genEnv( env );
        }

    public Type getType() {
        Type subType = subTypeNode.getType();
        return new ArrayType( size, subType );
        }
    }
```

We can declare class types, and refer to them by name. Notice that getType() converts the classType name to a classInstanceType.

```
public class TypeIdentNode extends TypeNode {

    private String ident;

    public TypeIdentNode( String ident ) {
        this.ident = ident;
        }

    public String toString() {
        return ident;
        }

    public Type getType() {
        Decl decl = env().searchEnv( ident );
        if ( decl == null )
            throw new Error( "Undeclared Identifier " + ident );
        if ( decl.section() != Decl.DECL_CLASS )
            throw new Error( ident + " must be class" );
        return decl.type().instanceType();
        }
    }
```

Declarations

For variable declarations, we get the type, check that the identifiers have not already been declared in the local block, and declare them. It is essentially the same for field declarations, and parameters.

```
public class VarDeclNode extends DeclStmtNode {

    private TypeNode typeNode;
    private DeclaratorListNode declrList;

    public VarDeclNode( TypeNode typeNode, DeclaratorListNode declrList ) {
        this.typeNode = typeNode;
        this.declrList = declrList;
        }

    public String toString() {
        return typeNode + " " + declrList + ";";
        }

    public void genEnv( int section, Env env ) {
        super.genEnv( section, env );
        typeNode.genEnv( env );
```

```
          declrList.genEnv( section, env );
          }

     public void setType() {
          Type type = typeNode.getType();
          declrList.setType( type );
          }

     public void checkType() {
          declrList.checkType();
          }

     public void genOffset() {
          declrList.genOffset();
          }

     public void eval( RunEnv runEnv ) throws ReturnException {
          declrList.eval( runEnv );
          }

     public void genCode() {
          Code.enter();
          declrList.genCode();
          Code.exit();
          }
     }

public abstract class DeclaratorNode extends OperandNode implements DeclNode {

     protected String ident;
     public String ident() { return ident; }
     private int section;
     protected Type type;
     protected Decl decl;
     public Decl decl() { return decl; }

     public DeclaratorNode( String ident ) {
          this.ident = ident;
          }

     public void genEnv( int section, Env env ) {
          this.section = section;
          super.genEnv( env );
          decl = env().declare( ident, section, this );
          }

     public void setType( Type type ) {
          this.type = type;
          decl.setType( type );
          }

     public abstract void checkType();

     public void genOffset() {
          decl.genOffset();
          }

     public abstract void eval( RunEnv runEnv ) throws ReturnException;

     protected Register saveReg( int regType, Usage usage ) {
          return null;
```

```
          }

     public void genDeclCode() {
          decl.allocateSpace();
          }

     public abstract void genCode();

     }

public class InitDeclaratorNode extends DeclaratorNode {

     private ExprNode expr;

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

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

     public void genEnv( int section, Env env ) {
          super.genEnv( section, env );
          expr.genEnv( env );
          }

     public void checkType() {
          Type exprType = expr.checkType();
          expr = expr.castTo( Type.CAST_IMPLICIT, type );
          }

     public void eval( RunEnv runEnv ) throws ReturnException {
          runEnv.elementAt( decl ).setValue( expr.eval( runEnv ) );
          }

     public void genCode() {
          expr.evalCode( null );
          Register sourceReg = expr.loadReg( expr.usage() );
          decl.store( sourceReg );
          }
     }

public class UninitDeclaratorNode extends DeclaratorNode {

     public UninitDeclaratorNode( String ident ) {
          super( ident );
          }

     public String toString() {
          return ident;
          }

     public void checkType() {
          }

     public void eval( RunEnv runEnv ) throws ReturnException {
          switch ( decl.env().envKind() ) {
               case Env.ENV_GLOBAL:
               case Env.ENV_METHOD:
```

```
                case Env.ENV_CLASS:
                     PtrValue address = runEnv.elementAt( decl );
                     decl.type().initAddress( runEnv, address );
                     break;
                default:
                     throw new Error(
                          "Invalid environment of "
                          + Env.envKindText( decl.env().envKind() )
                          + " for identifier " + decl.ident() );
                }
          }

     public void genCode() {
          switch ( decl.env().envKind() ) {
                case Env.ENV_GLOBAL:
                case Env.ENV_METHOD:
                case Env.ENV_CLASS:
                     setAddress( decl.address() );
                     Displacement varAddress = loadAddress( new Usage( env() ) );
                     Code.loadA( SpecialReg.newInstPtr, varAddress );
                     decl.type().initAddressCode();
                     break;
                default:
                     throw new Error(
                          "Invalid environment of "
                          + Env.envKindText( decl.env().envKind() )
                          + " for identifier " + decl.ident() );
                }
          }
     }
```

Class type declarations are not so different. However, this is one place where we set the type of the declaration in the genEnv() pass.

```
public class ClassDeclNode extends DeclStmtNode implements DeclNode {

     private String ident;
     private String extendsIdent;
     private DeclStmtListNode body;

     private ClassType classType;
     private Decl decl;
     private Decl extendedDecl;
     private Env localEnv;

     public String ident() { return ident; }
     public Decl decl() { return decl; }
     public DeclStmtListNode body() { return body; }

     public ClassDeclNode(
          String ident,
          String extendsIdent,
          DeclStmtListNode body ) {
          this.ident = ident;
          this.extendsIdent = extendsIdent;
          this.body = body;
          }

     public String toString() {
          String extendsText;
          if ( extendsIdent == null )
               extendsText = "";
```

```
        else
            extendsText = " extends " + extendsIdent;

        return "class " + ident + extendsText
            + "%+%nbegin%+%n"
            + body
            + "%-%nend%-";
        }

    public void genEnv( int section, Env env ) {
        super.genEnv( section, env );
        Env extendedEnv = null;
        if ( extendsIdent != null ) {
            extendedDecl = env.searchEnv( extendsIdent );
            if ( extendedDecl == null )
                throw new Error( "Undeclared class " + extendedDecl );
            else if ( extendedDecl.section() != Decl.DECL_CLASS )
                throw new Error( "extended declaration " +
                    extendedDecl + " should be a class" );
            ClassType extendedClass = ( ClassType ) extendedDecl.type();
            extendedEnv = extendedClass.env();
            }
        localEnv = new Env( Env.ENV_CLASS, env, extendedEnv );
        classType = new ClassType( ident, localEnv );
        decl = env.declare( ident, Decl.DECL_CLASS, this );
        decl.setType( classType );
        localEnv.setDecl( decl );
        body.genEnv( Decl.DECL_VAR, localEnv );
        }

    public void setType() {
        body.setType();
        }

    public void checkType() {
        body.checkType();
        }

    public void genOffset() {
        decl.genOffset();
        localEnv.setClassOffset();
        body.genOffset();
        localEnv.createMethodTable();
        }

    public void eval( RunEnv runEnv ) throws ReturnException {
        }

    ...
    }
```

Control Statements

Most control statements follow the same pattern.

The code for if statements is:

```
public class IfStmtNode extends StmtNode {

    private ExprNode cond;
    private DeclStmtListNode thenPart;
    private ElseOptNode elsePart;
```

```
    public IfStmtNode(
         ExprNode cond,
         DeclStmtListNode thenPart,
         ElseOptNode elsePart ) {
         this.cond = cond;
         this.thenPart = thenPart;
         this.elsePart = elsePart;
         }

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

    public void genEnv( int section, Env env ) {
         super.genEnv( section, env );
         cond.genEnv( env );
         thenPart.genEnv( section, env );
         elsePart.genEnv( section, env );
         }

    public void checkType() {
         Type condType = cond.checkType();
         cond = cond.castTo( Type.CAST_IMPLICIT, BoolType.type );
         thenPart.checkType();
         elsePart.checkType();
         }

    public void eval( RunEnv runEnv ) throws ReturnException {
         if ( cond.eval( runEnv ).boolValue() )
              thenPart.eval( runEnv );
         else
              elsePart.eval( runEnv );
         }

    public void genCode() {
         Code.enter();
         Code.labelDefn( "if" );
         cond.boolCode( null,
              new Relative( "then" ), new Relative( "else" ), true );
         Code.labelDefn( "then" );
         thenPart.genCode();
         Code.br( new Relative( "end" ) );
         Code.labelDefn( "else" );
         elsePart.genCode();
         Code.labelDefn( "end" );
         Code.exit();
         }
    }
```

The code for while statements is:

```
public class WhileStmtNode extends StmtNode {

    private ExprNode cond;
    private DeclStmtListNode stmtList;

    public WhileStmtNode( ExprNode cond, DeclStmtListNode stmtList ) {
         this.cond = cond;
         this.stmtList = stmtList;
         }
```

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

    public void genEnv( int section, Env env ) {
        super.genEnv( section, env );
        cond.genEnv( env );
        stmtList.genEnv( section, env );
        }

    public void checkType() {
        Type condType = cond.checkType();
        cond = cond.castTo( Type.CAST_IMPLICIT, BoolType.type );
        stmtList.checkType();
        }

    public void eval( RunEnv runEnv ) throws ReturnException {
        while ( cond.eval( runEnv ).boolValue() )
            stmtList.eval( runEnv );
        }

    public void genCode() {
        Code.enter();
        Code.labelDefn( "while" );
        cond.boolCode( null,
            new Relative( "do" ), new Relative( "end" ), true );
        Code.labelDefn( "do" );
        stmtList.genCode();
        Code.labelDefn( "continue" );
        Code.br( new Relative( "while" ) );
        Code.labelDefn( "break" );
        Code.labelDefn( "end" );
        Code.exit();
        }
    }
```

The code for for statements is:

```
public class ForStmtNode extends StmtNode {

    private ExprNode initial;
    private ExprNode cond;
    private ExprNode increment;
    private DeclStmtListNode stmtList;

    public ForStmtNode(
        ExprNode initial,
        ExprNode cond,
        ExprNode increment,
        DeclStmtListNode stmtList ) {
        this.initial = initial;
        this.cond = cond;
        this.increment = increment;
        this.stmtList = stmtList;
        }

    public String toString() {
        return "for " + initial + "; " + cond + "; "
            + increment + " do%+%n" + stmtList + "%-%nend";
        }

    public void genEnv( int section, Env env ) {
        super.genEnv( section, env );
```

```
        initial.genEnv( env );
        cond.genEnv( env );
        increment.genEnv( env );
        stmtList.genEnv( section, env );
        }

    public void checkType() {
        Type initialType = initial.checkType();
        initial = initial.castTo( Type.CAST_IMPLICIT, VoidType.type );
        Type condType = cond.checkType();
        cond = cond.castTo( Type.CAST_IMPLICIT, BoolType.type );
        Type incrementType = increment.checkType();
        increment = increment.castTo( Type.CAST_IMPLICIT, VoidType.type );
        stmtList.checkType();
        }

    public void eval( RunEnv runEnv ) throws ReturnException {
        for (
            initial.eval( runEnv );
            cond.eval( runEnv ).boolValue();
            increment.eval( runEnv ) )
            stmtList.eval( runEnv );
        }

    public void genCode() {
        Code.enter();
        Code.labelDefn( "for" );
        initial.evalCode( null );
        Code.labelDefn( "while" );
        cond.boolCode( null,
            new Relative( "do" ), new Relative( "end" ), true );
        Code.labelDefn( "do" );
        stmtList.genCode();
        Code.labelDefn( "continue" );
        increment.evalCode( null );
        Code.br( new Relative( "while" ) );
        Code.labelDefn( "break" );
        Code.labelDefn( "end" );
        Code.exit();
        }
    }
```

Simple Statements

The code for an expression used as a statement just invokes the code for the expression, and ignores the result.

```
public class ExprStmtNode extends StmtNode {

    private ExprNode expr;

    public ExprStmtNode( ExprNode expr ) {
        this.expr = expr;
        }

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

    public void genEnv( int section, Env env ) {
        super.genEnv( section, env );
        expr.genEnv( env );
```

```
        }

    public void checkType() {
        Type type = expr.checkType();
        expr = expr.castTo( Type.CAST_IMPLICIT, VoidType.type );
        }

    public void eval( RunEnv runEnv ) {
        expr.eval( runEnv );
        }

    public void genCode() {
        Code.enter();
        expr.evalCode( null );
        Code.exit();
        }
    }
```

Expressions

When type checking expressions, the checkType() method has to check the type of operands, cast them to the appropriate type, and return the type of the expression, for use by the parent.

```
public abstract class ExprNode extends OperandNode {

    public final static int PREC_ASSIGN   =    2;
    public final static int PREC_OR       =    4;
    public final static int PREC_AND      =    5;
    public final static int PREC_EQ       =    9;
    public final static int PREC_REL      =    9;
    public final static int PREC_ADD      =    12;
    public final static int PREC_MUL      =    13;
    public final static int PREC_CAST     =    15;
    public final static int PREC_PREFIX   =    15;
    public final static int PREC_POSTFIX  =    16;
    public final static int PREC_PRIMARY  =    17;

    protected int precedence;

    public String toString( int parentPrec ) {
        if ( precedence < parentPrec )
            return "( " + toString() + " )";
        else
            return toString();
        }

    public abstract Type checkType();

    public abstract RunValue eval( RunEnv runEnv );

    public ExprNode castTo( int castKind, Type postType ) {
        ExprNode exprNode = null;
        if ( ! postType.isCastable( castKind, type ) )
            throw new Error( "Can't cast " + type + " to " + postType );
        if ( type.equals( postType ) ) {
            return this;
            }
        else if ( postType instanceof VoidType ) {
            exprNode = new CastToVoidNode( this );
            }
        else if ( type instanceof IntType && postType instanceof CharType ) {
            exprNode = this;
```

```
            }
        else if ( type instanceof CharType && postType instanceof IntType ) {
            exprNode = this;
            }
        else if ( postType instanceof PtrType
            && postType.isCastable( Type.CAST_IMPLICIT, type ) ) {
            exprNode = this;
            }
        else {
            throw new Error( "Can't cast " + type + " to " + postType );
            }
        return exprNode;
        }

    public abstract void evalCode( OperandNode sibling );
    ...
    }
```

The code for operators is often in superclasses, because much of the code is in common.

```
public abstract class BinaryNode extends ExprNode {

    protected String operator;
    protected String opCode;
    protected Type operandType;
    protected ExprNode left, right;

    public BinaryNode( ExprNode left, ExprNode right ) {
        this.left = left;
        this.right = right;
        }

    public String toString() {
        return
            left.toString( precedence )
            + " " +  operator + " "
            + right.toString( precedence + 1 );
        }

    public void genEnv( Env env ) {
        super.genEnv( env );
        left.genEnv( env );
        right.genEnv( env );
        }

    public void evalCode( Node sibling ) {
        setSibling( sibling );
        left.evalCode( sibling );
        right.evalCode( left );
        left.loadReg( usage().plus( right.address() ) );
        right.loadReg( usage().plus( left.address() ) );
        setAddress( obtainReg( type, usage() ) );
        }
    }

public abstract class ArithNode extends BinaryNode {

    public ArithNode( ExprNode left, ExprNode right ) {
        super( left, right );
        }

    public Type checkType() {
        Type leftType = left.checkType();
```

```
          Type rightType = right.checkType();
          operandType = IntType.type;
          left = left.castTo( Type.CAST_IMPLICIT, operandType );
          right = right.castTo( Type.CAST_IMPLICIT, operandType );
          type = operandType;
          return type;
          }
     }

public class TimesNode extends ArithNode {

     public TimesNode( ExprNode left, ExprNode right ) {
          super( left, right );
          precedence = PREC_MUL;
          operator = "*";
          opCode = "mulq";
          }

     public RunValue eval( RunEnv runEnv ) {
          RunValue leftValue = left.eval( runEnv );
          RunValue rightValue = right.eval( runEnv );
          return new IntValue( leftValue.intValue() * rightValue.intValue() );
          }

     public void evalCode( OperandNode sibling ) {
          super.evalCode( sibling );
          Code.instrn( opCode, left.address(), right.address(), address() );
          }
     }
```

The code for "+" is a little more complex, because the operator can be used for both integer addition, and pointer plus integer.

```
public class PlusNode extends ArithNode {

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

     public Type checkType() {
          Type leftType = left.checkType();
          Type rightType = right.checkType();
          if ( leftType instanceof PtrType ) {
               right = right.castTo( Type.CAST_IMPLICIT, IntType.type );
               type = leftType;
               return type;
               }
          else {
               operandType = IntType.type;
               left = left.castTo( Type.CAST_IMPLICIT, operandType );
               right = right.castTo( Type.CAST_IMPLICIT, operandType );
               type = operandType;
               return type;
               }
          }

     public RunValue eval( RunEnv runEnv ) {
          RunValue leftValue = left.eval( runEnv );
          RunValue rightValue = right.eval( runEnv );
```

```
          if ( type instanceof PtrType ) {
               PtrType ptrType = ( PtrType ) type;
               int size = ptrType.subType().size().interp();
               return leftValue.addressValue().elementAt(
                    size * rightValue.intValue() );
               }
          else
               return new IntValue(
                    leftValue.intValue() + rightValue.intValue() );
          }

     public void evalCode( OperandNode sibling ) {
          super.evalCode( sibling );
          Code.scaledAdd( type, left.address(), right.address(), address() );
          }
     }
```

The nodes for constants just return the constant.

```
public abstract class ValueNode extends ExprNode {

     public ValueNode() {
          precedence = PREC_PRIMARY;
          }
     }

public class IntValueNode extends ValueNode {

     private Integer value;

     public IntValueNode( Integer value ) {
          this.value = value;
          }

     public String toString() {
          return "" + value;
          }

     public Type checkType() {
          type = IntType.type;
          return type;
          }

     public RunValue eval( RunEnv runEnv ) {
          return new IntValue( value.intValue() );
          }

     public void evalCode( OperandNode sibling ) {
          setSibling( sibling );
          setAddress( new Literal( value.intValue() ) );
          }
     }
```

Variables

Some expressions, such as assignment operators, refer to variables, for example the left-hand side of the assignment operator, the operand of “++” or “--”, and “&”.

The code for assignment is

```
public class AssignNode extends ExprNode {
```

```
    private VariableNode variable;
    private ExprNode expr;
    private Type varType;

    public AssignNode( VariableNode variable, ExprNode expr ) {
        this.variable = variable;
        this.expr = expr;
        precedence = PREC_ASSIGN;
        }

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

    public void genEnv( Env env ) {
        super.genEnv( env );
        variable.genEnv( env );
        expr.genEnv( env );
        }

    public Type checkType() {
        varType = variable.checkType();
        if ( ! variable.addressable() )
            throw new Error( "Operand for = not addressable" );
        Type exprType = expr.checkType();
        expr = expr.castTo( Type.CAST_IMPLICIT, varType );
        type = VoidType.type;
        return type;
        }

    public RunValue eval( RunEnv runEnv ) {
        PtrValue address = variable.evalAddr( runEnv );
        RunValue value = expr.eval( runEnv );
        address.setValue( value );
        return VoidValue.value;
        }

    public void evalCode( Node sibling ) {
        setSibling( sibling );
        expr.evalCode( sibling );
        variable.evalAddrCode( expr );
        expr.loadReg( usage().plus( variable.address() ) );
        variable.loadAddress( usage().plus( expr.address() ) );
        setAddress( null );
        Code.store( varType, ( Register ) expr.address(), variable.address() );
        }
    }
```

The code for “&” (address of) is

```
public abstract class PrefixVarNode extends ExprNode {

    protected VariableNode right;
    protected String operator;
    protected Type operandType;
    protected String opCode;

    public PrefixVarNode( VariableNode right ) {
        this.right = right;
        precedence = PREC_PREFIX;
        }

    public String toString() {
```

```
          return operator + " " + right;
          }

     public void genEnv( Env env ) {
          super.genEnv( env );
          right.genEnv( env );
          }

     public void evalCode( OperandNode sibling ) {
          setSibling( sibling );
          right.evalAddrCode( sibling );
          }
     }

public class AddressNode extends PrefixVarNode {

     public AddressNode( VariableNode right ) {
          super( right );
          operator = "&";
          }

     public Type checkType() {
          Type rightType = right.checkType();
          if ( ! right.addressable() )
               throw new Error( "Operand for & not addressable" );
          type = new PtrType( rightType );
          return type;
          }

     public RunValue eval( RunEnv runEnv ) {
          return right.evalAddr( runEnv );
          }

     public void evalCode( OperandNode sibling ) {
          super.evalCode( sibling );
          Register register = obtainReg( type, usage() );
          Address address = right.loadAddress( usage() );
          Code.loadA( register, address );
          setAddress( register );
          }
     }
```

A VarExprNode is used when we use a variable for its value.

```
public class VarExprNode extends ExprNode {

     private VariableNode variable;
     private Type addressType;

     public VariableNode variable() { return variable; }

     public VarExprNode( VariableNode variable ) {
          this.variable = variable;
          precedence = PREC_PRIMARY;
          }

     public String toString() {
          return "" + variable;
          }

     public void genEnv( Env env ) {
          super.genEnv( env );
          variable.genEnv( env );
```

```
        }

    public Type checkType() {
        addressType = variable.checkType();
        type = addressType.formalType();
        return type;
        }

    public RunValue eval( RunEnv runEnv ) {
        if ( addressType instanceof ArrayType
            || addressType instanceof ClassInstanceType )
            return variable.evalAddr( runEnv );
        else
            return variable.evalAddr( runEnv ).getValue();
        }

    public void evalCode( OperandNode sibling ) {
        setSibling( sibling );
        variable.evalAddrCode( sibling );
        Address address = variable.loadAddress( usage() );
        if ( addressType instanceof ArrayType
            || addressType instanceof ClassInstanceType ) {
            if ( address instanceof Indirect ) {
                Indirect indirectAddress = ( Indirect ) address;
                setAddress( indirectAddress.base() );
                }
            else {
                Register register = obtainReg( type, usage() );
                Code.loadA( register, address );
                setAddress( register );
                }
            }
        else
            setAddress( address );
        }
    }
```

The VariableNode class is an abstract class representing a variable. The addressable() method indicates whether it is possible to assign to the variable or not. This is important, because some things that look like variables may not be. In this language, it is not possible to assign to an array or class instance. In other languages, it might not be possible to assign to a for loop variable or formal parameter.

```
public abstract class VariableNode extends OperandNode {
    protected Type type;
    protected Type addressType;
    public Type type() { return type; }
    public boolean addressable() {
        if ( type instanceof ArrayType || type instanceof ClassInstanceType )
            return false;
        else
            return true;
        }

    public VariableNode() {
        }

    public abstract Type checkType();
    public abstract PtrValue evalAddr( RunEnv runEnv );
    public abstract void evalAddrCode( OperandNode sibling );
    ...
    }
```

Variables can be

- Simple identifiers. Finally we get down to a node that actually makes use of the compile-time and run-time environments. Type checking includes checking that it is declared as a variable or parameter in the compile-time environment. Evaluation involves getting the object containing the value from the run-time environment.

```
public class Decl {
     ...
     public Address address() {
          switch ( env().envKind() ) {
               case Env.ENV_GLOBAL:
                    return new Indirect( new Literal( absoluteName() ) );
               case Env.ENV_CLASS:
                    //   Only works for fields
                    return new Displacement( absoluteName(), SpecialReg.instPtr );
               case Env.ENV_METHOD:
                    return new Displacement( declName(), SpecialReg.framePtr );
               default:
                    throw new Error(
                         "Invalid section of " + sectionText( section() ) );
               }
          }
     ...
     }

public class IdentVariableNode extends VariableNode {

     private String ident;
     private Decl decl;

     public IdentVariableNode( String ident ) {
          this.ident = ident;
          }

     public String toString() {
          return ident;
          }

     public Type checkType() {
          decl = env().searchEnv( ident );
          if ( decl == null )
               throw new Error( "Undeclared Identifier " + ident );
          switch ( decl.section() ) {
               case Decl.DECL_VAR:
               case Decl.DECL_PARAM:
                    type = decl.type();
                    return type;
               default:
                    throw new Error( ident + " must be variable or parameter" );
               }
          }

     public PtrValue evalAddr( RunEnv runEnv ) {
          return runEnv.elementAt( decl );
          }

     public void evalAddrCode( OperandNode sibling ) {
          setSibling( sibling );
          setAddress( decl.address() );
          }
     }
```

- Field selections. Field selection is not so different from simple identifiers, but the block to search comes from the type of the class instance being selected, not the compile-time environment. Evaluation involves determining the class instance being selected, then getting the field at the appropriate offset from this structure.

```
public class MemberVariableNode extends VariableNode {

     private ExprNode object;
     private String ident;
     private Decl decl;

     public MemberVariableNode( ExprNode object, String ident ) {
          this.object = object;
          this.ident = ident;
          }

     public String toString() {
          return object.toString( ExprNode.PREC_PRIMARY ) + "." + ident;
          }

     public void genEnv( Env env ) {
          super.genEnv( env );
          object.genEnv( env );
          }

     public Type checkType() {
          Type oType = object.checkType();
          if ( ! ( oType instanceof PtrType ) )
               throw new Error( "Selection must be a reference" );
          PtrType ptrType = ( PtrType ) oType;
          Type sType = ptrType.subType();
          if ( ! ( sType instanceof ClassInstanceType ) )
               throw new Error(
               "Selection must be a reference to an instance of class type" );
          ClassInstanceType instanceType = ( ClassInstanceType ) sType;
          decl = instanceType.env().searchExtended( ident );
          if ( decl == null )
               throw new Error( "Undeclared field " + ident );
          if ( decl.section() != Decl.DECL_VAR )
               throw new Error( ident + " must be field" );
          type = decl.type();
          return type;
          }

     public PtrValue evalAddr( RunEnv runEnv ) {
          PtrValue address =
               object.eval( runEnv )
                    .addressValue().elementAt( decl.offset().interp() );
          return address;
          }

     public void evalAddrCode( OperandNode sibling ) {
          setSibling( sibling );
          object.evalCode( sibling );
          Register base = object.loadReg( usage() );
          setAddress( new Displacement( decl.absoluteName(), base ) );
          }
     }
```

- Indexed variables. Indexing is similar. Type checking involves checking we are indexing a pointer using an integer, and returning the subtype of the pointer type as the type of the whole

expression. Evaluation involves determining the pointer and index, then getting the variable at the appropriate index from this array. It is really very similar to addition involving a pointer.

```
public class IndexVariableNode extends VariableNode {

     private ExprNode pointer;
     PtrType pointerType;
     private ExprNode index;
     private Register resultReg;

     public IndexVariableNode( ExprNode pointer, ExprNode index ) {
          this.pointer = pointer;
          this.index = index;
          }

     public void genEnv( Env env ) {
          super.genEnv( env );
          pointer.genEnv( env );
          index.genEnv( env );
          }

     public Type checkType() {
          Type aType = pointer.checkType();
          if ( ! ( aType instanceof PtrType ) )
               throw new Error(
                    "Attempting to index a non-pointer type" + aType );
          pointerType = ( PtrType ) aType;
          Type indexType = index.checkType();
          index = index.castTo( Type.CAST_IMPLICIT, IntType.type );
          type = pointerType.subType();
          return type;
          }

     public String toString() {
          return pointer.toString( ExprNode.PREC_PRIMARY ) + "[ " + index + " ]";
          }

     public PtrValue evalAddr( RunEnv runEnv ) {
          PtrValue pointerAddress = pointer.eval( runEnv ).addressValue();
          RunValue indexValue = index.eval( runEnv );
          return pointerAddress.elementAt(
               type.size().interp() * indexValue.intValue() );
          }

     public void evalAddrCode( Node sibling ) {
          setSibling( sibling );
          pointer.evalCode( sibling );
          index.evalCode( pointer );
               pointer.loadReg( usage().plus( index.address() ) );
          index.loadReg( usage().plus( pointer.address() ) );
          resultReg = obtainReg( pointerType, usage() );
          Code.scaledAdd(
               pointerType, pointer.address(), index.address(), resultReg );
          setAddress( new Indirect( resultReg ) );
          }
     }
```

- Indirection. This is really the same as indexing, with an index of 0.

```
public class IndirectVariableNode extends VariableNode {
```

```
    private ExprNode address;

    public IndirectVariableNode( ExprNode address ) {
        this.address = address;
        }

    public String toString() {
        return address.toString( ExprNode.PREC_PRIMARY ) + "^";
        }

    public void genEnv( Env env ) {
        super.genEnv( env );
        address.genEnv( env );
        }

    public Type checkType() {
        Type pType = address.checkType();
        if ( ! ( pType instanceof PtrType ) )
            throw new Error(
                "Attempting to dereference a non pointer type " + pType );
        PtrType ptrType = ( PtrType ) pType;
        type = ptrType.subType();
        return type;
        }

    public PtrValue evalAddr( RunEnv runEnv ) {
        RunValue value = address.eval( runEnv );
        return value.addressValue();
        }

    public void evalAddrCode( OperandNode sibling ) {
        setSibling( sibling );
        address.evalCode( sibling );
        Register base = address.loadReg( usage() );
        setAddress( new Indirect( base ) );
        }
    }
```