Chapter 5

Symbol Tables, Declarations and References

Symbols are used for various different purposes within a compiler. There are symbols that represent constants, both generated internally and ones explicity specified by the programer (such as with the const declaration in Plat). There are symbols that represent types, such as the data types integer or float. There are symbols that represent memory locations, such as variables stored in global memory or on the activation record. Symbols can describe functions. And finally symbols can represent fields in a record or a class.

As we noted in Chapter ??, there are two major reasons a compiler must make use of a symbol table. First, the symbol table must carry information from the point where a symbol is first introduced, typically in a declaration statement for a variable or a function definition, to the point where the symbol is used, such as a variable being used in an assignment statement. Second, the symbol table then provides the way to interpret the meaning of the symbol, which generally means indicating how to create the intermediate representation that corresponds to the address of a variable or function.

In this chapter we will first consider the role of symbols in the abstract. This will be followed by a discussion of the various different types of symbol tables, and the way they are used to create the internal representation of variables.

5.1 Symbols

Symbols will be maintained internally by an instance of class Symbol. The description of this class is shown in Figure 5.1. As can be seen by the class definition, all symbols have a *name*, a string handle by which they are identified. This name field is set by the constructor, and cannot be modified once set. The name field for a symbol can be retrieved via the method name. Symbols can be compared for equality, and are assumed to be equal if their name

```
class Symbol {
    private String nm;

    public Symbol (String n)  { nm = n; }

    public String name () { return nm; }

    public Ast buildAddr (Ast base) { return null; }

    public boolean equals (Object two) {
            // symbols are equal if equal in name
        if (two instanceof String) return nm.equals(two);
        if (two instanceof Symbol) return two.equals(nm);
        return false;
        }
}
```

Figure 5.1: Class Definition for Symbol

fields match. In comparing for equality, the argument passed to the the method equals can be either a string value, or another symbol. Anything other than a string or symbol is considered to be not equal.

Most symbols (but not all) are used to describe memory locations of one sort or another. These symbols have the ability to create an address expression, through the invocation of the method buildAddr. An address expression is simply an abstract syntax tree that describes the address of the memory location associated with the symbol. Symbols that do not denote memory locations will simply return a null value in response to this method.

Because the vast majority of addresses for symbols are known only as a relative offset, an expression is passed to the method buildAddr that represents the "base" for the address. The meaning of this argument depends in part on the type of symbol table in which the symbol appears. This will be described in more detail when we discuss the various types of symbol tables.

5.1.1 Constant Symbols

The simplest type of symbol is a constant. A constant is nothing more than an identifier that stands in place of an expression. Most languages, including Plat, require that the value of this expression be a literal value. In Plat constants are created by the const declaration, as in the following:

```
const Limit = 100;
```

Some languages allow more general expressions, if they can be evaluated at compile time. We can permit either of these alternatives by simply storing in the symbol table an abstract syntax tree for the expression:

```
class ConstantSymbol extends Symbol {
    private Ast value;

    public ConstantSymbol (String name, Ast v)
        { super(name); value = v; }

    public Ast buildAddr (Ast base) { return value; }
}
```

Like all symbols, a constant has a name. In addition, constants have a value. Both these quantities are set by the constructor. When the AST representation of a constant is requested, the value is returned. In Section 5.5.2 an example will be presented that illustrates how constants are added to a symbol table.

5.1.2 Type Symbols

A number of different symbol categories need to maintain a field of type Type (see Chapter 4 for a description of the class Type). To simplify the development of these abstraction, we create a common parent class, named TypedSymbol. A typed symbol requires both a name and a type, and allows the type field to be accessed using the method type():

```
class TypedSymbol extends Symbol {
    protected Type ty;

    public TypeSymbol (String name, Type t)
        { super (name); ty = t; }

    public Type type() { return ty; }
}
```

The simplest form of typed symbol is a type symbol. A type symbol does not denote a memory location, but instead maintains a value that describes a type. Type symbols arise in a number of different ways. The primitive types, integer, real and so on, are represented by type symbols in the symbol table associated with the global scope. Some languages allow the programmer to introduce new type names. We could extend Plat, for example, to permit declarations such as the following:

```
type matrix = [1 : 5][1:5] integer;
```

This would indicate that the name matrix was a synonym for a two dimensional array of integer values. Subsequent declarations of the type matrix would then be equivalent to a declaration formed using the array notation. There is no memory associated with the definition of matrix, although there would (of course) be memory assigned to values created using this type.

Because the information stored by a TypeSymbol can be inherited from TypedSymbol, it is only necessary to implement the constructor function.

```
class TypeSymbol extends TypedSymbol {
    public TypeSymbol (String name, Type t)
        { super (name, t);}
}
```

The two classes are necessary because we will subsequently define other symbols that use a type field, but are not themselves type symbols. Testing the class of a symbol to see whether or not it is TypeSymbol will tell whether or not it is a type symbol, whereas testing against TypedSymbol would not provide this information (see Section 5.5.4).

Finally, note that type symbols do not implement the method buildAddr, and thus cannot generate an internal expression.

5.1.3 Global Symbols

A global symbol represents the memory location for a value stored in global memory. In general, the only information we have conerning global values is a type and an internal name. Making the class a subclass of TypedSymbol will implicitly provide storage and access to the type field. From the type field we can, if necessary, determine the amount of memory used by the corresponding variable. The internal name field is used to hold the mangled name for the memory quantity, which may be different from the name used as key in the symbol table. (Recall the discussion on name mangling in Section 3.1.2). All data fields are set by the constructor, and can be accessed by the appropriate member functions. When the expression representing the address of the memory is required, a global node that refers to the mangled name is generated.

```
class GlobalSymbol extends TypedSymbol {
    private String mangledName;

    public GlobalSymbol (String name, Type t, String mn)
        { super(name, t); mangledName = mn; }

    public String internalName() { return mangledName; }
```

```
    public Ast buildAddr (Ast base)
        { return new GlobalNode(new AddrOfType(type()), mangledName); }
}
```

The following subtle point should be noted. If we let ty represent the type associated with a global symbol, then the the expression returned by buildAddr is marked as being type "address of type", and not simply type. We will return to this discussion of values versus addresses in Section 5.5.7.

5.1.4 Offset Symbols

Offset symbols represent memory locations that are known only as an offset from some base. The address of the base will be determined by the symbol table, as will subsequently be described. Offset symbols can represent parameters, local variables, or fields in a record or class. Offset symbols have a type and an offset, both set by the constructor.

```
class OffsetSymbol extends TypedSymbol {
    private int location;

    public OffsetSymbol (String name, Type t, int n)
        { super(name, t); location = n; }

    public Ast buildAddr (Ast base) {
        return new BinaryNode(new AddrOfType(type()), BinaryNode.plus,
            base, new IntegerNode(location));
        }
}
```

The expression tree built to represent the location of the symbol is an addition of the base and the integer location value. Note that, as with global symbols, this computation results in the address of the memory location. The type attached to this node is therefore an address type, not the type of the value held by the node.

5.1.5 Function Symbols

There are three types of function symbols. These correspond to global functions (that is, functions defined at the outermost program level), nested functions, and methods. The class definitions for these categories is as follows:

```
class FunctionSymbol extends TypedSymbol {
    protected Ast code;
```

```
    public functionSymbol (String name, Ast c, Type t)
        { super(name, t); code = c;  ty = t; }

    public Ast buildAddr (Ast base) {
        ... // build function symbol
        }
}

function NestedFunctionSymbol extends FunctionSymbol {

    public NestedFunctionSymbol
        (String name, Ast c, Type t)
            { super(name, c, t); }

    public Ast buildAddr (Ast base) {
        ...//save base as static chain
        }
}

function MethodSymbol extends FunctionSymbol {
    protected Ast surroundingContext;

    public NestedFunctionSymbol
        (String name, Ast c, Type t)
            { super(name, c, t); }

    public Ast buildAddr (Ast base) {
        ...// save base as receiver
        }
}
```

The code field of a function is the address used to access the beginning of the code segment for the function. Frequently this is just a global name. Nested functions and methods have, in addition, a surrounding context that is used in accessing non-local variables. How these values are used will be described in the next section.

5.2 Scopes

The *scope* of an identifier refers to the textual region of a program in which the identifier can legally appear. Most programming languages, Plat included, have several different

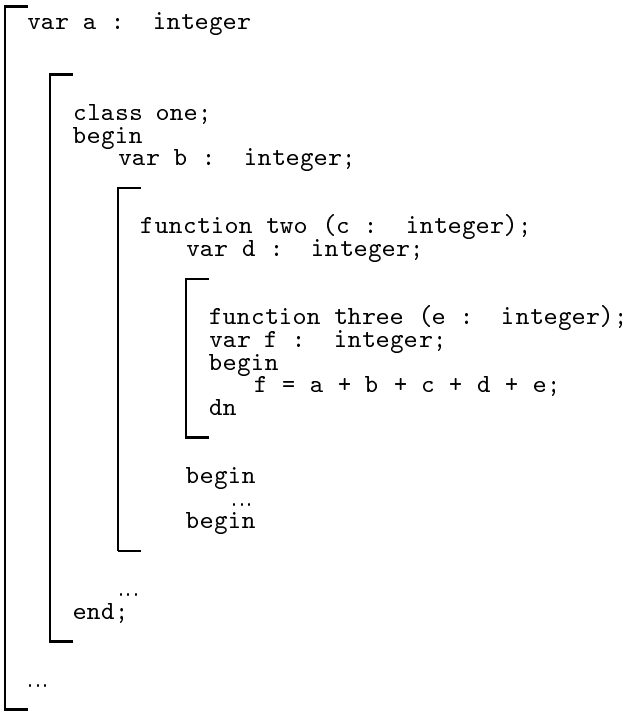

Figure 5.2: A program with many nested scopes

mechanisms for creating new scopes. The global scope includes symbols that can appear anywhere in a program, once their declaration has been processed. Function definitions introduce a new scope, containing local variables and parameter names. These symbols can be used within the function body, but are meaningless (or, at least, have different meanings) outside the function. Class definitions introduce yet another scope.

Scopes can be *nested* one within another. All scopes are nested within the global scope. When one function is nested within another, the scope for the inner function is nested within that of the outer. Functions defined as part of classes are nested within the scope for the class. This situation is illustrated by the Plat program shown in Figure 5.2. Here the identifier a is a global symbol. The class definition for one introduces a new scope. In this scope the variable b is declared. The name b, by itself, can only be used within the bounds of the class definition.[1] There is also a function, named two, defined inside of one.

[1] As we will describe in the next chapter, the name can be used as a field name in association with an object of type one, but that is a different issue.

The function definition introduces a new name scope, in which the parameter c and the local variable d are defined. Also defined within two is a nested function named three. This function, as well, defines both a parameter value and a local variable. Within the innermost definition; that is, within the function three, all the identifier names from surrounding scopes can be used. This is illustrated by the assignment statement in the body of this function.

To facilitate this, a separate symbol table will be created for each different name scope. These tables will be linked, so that at compile time the layout of the symbol tables will mirror the nesting of the scopes within the program. This is shown by Figure 5.3, which shows the linked series of symbol tables that would exist at the time the innermost function in Figure 5.2 was being processed. The locations of the symbols associated with the various identifiers in the program is also shown.

When an identifier is found in the body of a program, the symbol tables are searched to determine the interpretation to be placed on the symbol. This search proceeds from the innermost symbol table, the most current table, outwards towards the global symbol table. The first entry found that matches the text of the symbol determines the meaning of the symbol. As an artifact of this process a local variable can, for example, hide a global variable with the same name. Within the scope of the procedure in which the variable is declared, the search on the symbol table will always find the local symbol first.

5.3 Accessing Non-local Variables

The addresses of many values are easy to determine. Global variables are addressed using their symbolic name. As we saw in Chapter 3, local variables and parameters are easily accessed as a constant offset relative to the frame pointer, which is usually maintained in a machine register. It is the values that are in the intermediate symbol tables in Figure 5.3, those values that are not local but also not global, that are most difficult to access.

If we treat function names the same as any other symbol, the natural scoping of names described in the last section implies that a function can only call another function if the function being called is one of the following:

- The function being called has a more global scope than the caller.
- The function being called has the same scope level as the caller.
- The function being called is one (but no more than one) level lower than the caller.

This property will guarantee that any value accessible to a called procedure must still be stored somewhere on the activation record stack. The only difficulty is finding it. To see some of the problems that can occur, consider the set of functions shown in Figure 5.4. Imagine that we are executing the body of function one. Function one first calls function two. The first time two is executing, the activation record for one is immediately below it in the stack, as in the following picture:

picture

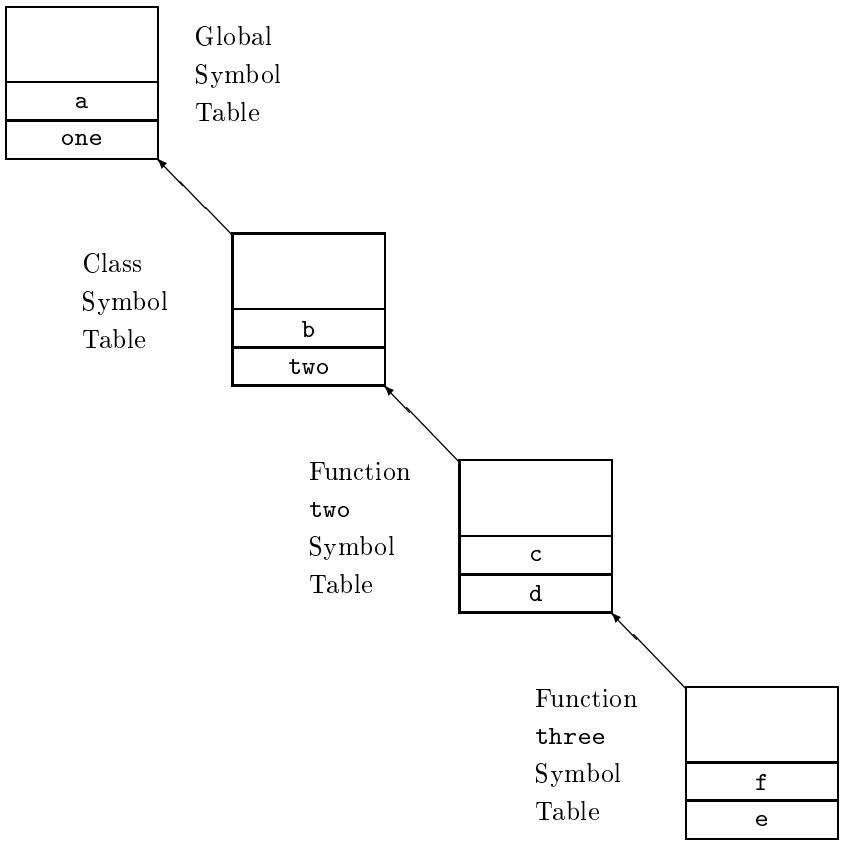

Figure 5.3: Symbol tables linked to show nesting

```
function one ();
var x : integer;

    function two();
    begin    print(x);    end;

    function three();
    begin
        if (...) three();
        else two();
    end;

    function four();
    begin    three();    end;

begin
    two(); four();
end;
```

Figure 5.4: Functions that Illustrate the need for static links

Recall that the dynamic link in each activation record points to the previous activation record. One might imagine that this value can be used to access the caller procedure. That is, to get hold of variable x from within two one could simply use a calculation such as the following:

```
    // DOESN'T WORK !!
    address of x = * (fp + dynamic link offset) + offset of x
```

To see that this is not true, consider next the activation record structure the second time that two is executed, after four has been called, and after three has recursed some unknown number of times.

picture

This time, we have no idea how many activation records lay between the activation record for two and the activation record for one in which the variable x is found. Thus, there is no way that the compiler can generate any expression that will calculate the address of x, using only the current frame pointer and the dynamic links.

To solve this problem, we need to introduce a new pointer, one that will at run time mirror the nested scoping of procedures, and the associated nested scoping of symbol tables shown in Figures 5.2 and 5.3. This pointer is termed the *static link*, since it reflects the static (compile time) layout of the program and not the dynamic (run-time) trace of execution.

The invariant maintained by the static link can be described as follows:

> *Within each activation record that represents a nested name scope, the static link is a pointer to the activation record for the surrounding scope.*

For example, the static links held in the activation records generated for procedures two, three and four in Figure 5.4 each point to the activation record for procedure one. Thus, to access the address of x from within two, the following calculation can be used:

```
address of x = * (fp + static link offset) + offset of x
```

This is true regardless of the number of activation records that may come between. For example, the following shows the static link values that are used the second time the procedure two is executed:

picture

Because a procedure can be called from many different levels, only the caller has sufficient information to determine the static link value (see Exercise 2). In practice the static link is therefore set by the caller, and stored by the called procedure. An easy way to do this, one that we will henceforth use, is to pass the static link as a "hidden" first parameter–that is, a parameter value that is not reflected in the original source program. Exactly how this is done will be explained after we discuss the structure and use of symbol tables.

5.4 Symbol Tables

A Java definition for SymbolTable appears in Figure 5.5. The definition has been given as an interface, rather than a class, so as to place emphasis on the operations provided by the data type, and not the implementation. The methods enterConstant, enterType, enterVariable and *enterFunction* create the different types of symbols. As we will see, various different types of tables handle these possibilities in a different fashion.

The method mangledPrefix returns the initial part of a mangled internal name.

As we will subsequently see, arguments for functions defined at the global level begin at one location, while arguments for functions that are nested or are member functions begin at a different offset relative to the activation frame. The function argStart returns the integer value that represents the location of the first parameter value.

The method nameDefined searches only the innermost symbol table to determine whether or not a given name has already been defined. This method is used to detect, for example, two local variables being declared with the same name. On the other hand, the methods findType and buildAddr search the entire linked list of symbol tables. The first returns the type value corresponding to the name (throwing a parse exception if there is no such record), while the second returns an expression tree that represents the address of the corresponding memory location.

```
interface SymbolTable {
        // methods to enter values into symbol table
    public void enterConstant (String name, AST value);
    public void enterType (String name, Type type);
    public void enterVariable (String name, Type type);
    public void enterFunction (String name, FunctionType ft);

        // information about current symbol table
    public String mangledPrefix();
    public int argStart();

        // methods to search the symbol table
    public boolean nameDefined (String name);
    public Type findType (String name) throw parseException;
    public Ast buildAddr (String name, Ast base) throw parseException;
}
```

Figure 5.5: Interface definition of SymbolTable

5.4.1 Symbol Table Implementation Techniques

A symbol table is basically just a dictionary, that is, a collection of key and value pairs. There are any number of different data structures that can be used to implement such containers. Examples include vectors, linked lists, and binary trees. The tradeoffs and benefits of each of the various possiblities is a topic that will be discussed in any good book on data structures. We will here describe only one simple scheme, which is sufficient for our purposes.

The class SimpleSymbolTable, shown in Figure 5.6, stores the symbols in a Vector of values. The class Vector, provided in the Java run-time library, permits easy insertion and removal of values, as well as testing to see if a value is included. The vector is declared as protected, so as to be available to subclasses of the class.

The class SimpleSymbolTable has been declared as abstract, meaning that some of the operations have not been defined. These methods must be defined in the classes that inherit from this class

5.4.2 Global Symbol Tables

The simplest symbol table to understand is the global symbol table. Variables entered into the global table are represented by global symbols. The mangling algorithm for global symbols simply appends an underscore to the front of the name. Functions have their name mangled, and are represented by function symbols, which in turn referernce the code as a global node.

```
abstract class SimpleSymbolTable implements  SymbolTable {

    public SimpleSymbolTable () { table = new Vector(); }

    public void enterConstant (String name, AST value)
        { enterSymbol(new ConstantSymbol(name, value)); }

    public void enterType (String name, Type type)
        { enterSymbol (new TypeSymbol(name, type)); }

    abstract public void enterVariable (String name, Type type);

    abstract public void enterFunction (String name, FunctionType ft);

    public String mangledPrefix() { return ""; }

    public int argStart() { return 0; }

    public boolean nameDefined (String name)
        { return lookup (name) != null; }

    public Type findType (String name) throw parseException {
        Symbol sym = lookup(name);
        if ((sym == null) || ! (sym instanceof TypeSymbol))
            throw new ParseException("expecting type, found"+name);
        return ((TypedSymbol) sym).type(); }

    public Ast buildAddr (Ast base, String name) throw parseException {
        Symbol sym = lookup(name);
        if (sym == null)
            throw new ParseException("symbol not found"+name);
        return sym.buildAddr(base); }

        // the following are protected internal methods
    protected Vector table;

    protected void enterSymbol (Symbol sym) { table.addElement(sym); }

    protected Symbol lookup (String name) {
        int idx = table.indexOf(name);
        if (idx >= 0) return table.elementAt(idx);
        else return null; }

}
```

Figure 5.6: Definition of class SimpleSymbolTable

```
class GlobalSymbolTable extends SimpleSymbolTable {

    public String mangledPrefix() { return "_"; }

    public void enterVariable (String name, Type type)
        { enterSymbol (new GlobalSymbol(name, type, name)); }

    public void enterFunction (String name, FunctionType ft) {
        String internalName = mangledPrefix() + name;
        enterSymbol (new FunctionSymbol(name,
            new GlobalNode(ft, internalName), ft));
        }
}
```

Because the values stored in global symbol tables are global symbols, when an address is constructed from a value in the global symbol table, the "base" field is ignored, and a global value is returned.

5.4.3 Record Symbol Tables

A record symbol table is the simplest example of a symbol table that maintains values as offsets in a linear structure. A record symbol table would be created during the processing of the declaration of a record, or structure. Each field in a record is simply laid out in memory following the previous. In this fashion a record such as the following:

```
record
    age : integer;
    height: real;
    status: integer;
end
```

Would be stored in memory as follows:
picture

The record symbol table maintains an internal integer variable that describes the current record size. This value can also be accessed as the result of the method named size. As variables are entered into the symbol table, the values are given the current location as an offset, and the size of the record is increased.[2]

[2] As we noted in Chapter 4, on some systems certain types must be aligned on a given boundary. For example, real numbers must often be aligned on addresses that are divisible by 8. As we described in Chapter 4. this information could be stored as an additional field in the Type record. The record symbol table would then perform a calculation prior to inserting the symbol to ensure that the appropriate boundary

```
class RecordSymbolTable extends SimpleSymbolTable {
    protected int recordsize;

    public RecordSymbolTable () { recordsize = 0; }

    public RecordSymbolTable (int is) { recordsize = is; }

    public int size () { return recordsize; }

    public void enterVariable (String name, Type type) {
        enterSymbol (new OffsetSymbol(name, recordsize, type));
        recordsize = recordsize + type.size();
        }
}
```

Although record symbol tables will almost always begin their addresses at zero, other types of table we will subsequently construct from record symbol tables will sometimes begin addressing their values with another location. For this reason there are two constructors, one of which allows the recordsize to begin with a different quantity.

When the abstract syntax tree for the address of a field is created, the "base" will be the base of the record structure. The method buildAddr inherited from SimpleSymbolTable will automatically add the offset to the base, and return the syntax tree that represents the address of a data field.

5.4.4 Argument Symbol Tables

The layout of arguments in memory is very similar to a record. For this reason, an ArgumentSymbolTable can usefully inherit from class RecordSymbolTable. However, instead of a positive offset, you will recall that arguments typically were accessed as a negative offset relative to the activation frame pointer. Furthermore, this value is the *start* of the associated field, which means the size of the structure is calculated *first*, before the entry is placed into the symbol table.

```
class ArgumentSymbolTable extends RecordSymbolTable {

    public void ArgumentSymbolTable (int is) { super(is); }

    public void enterVariable (String name, Type type) {
        recordsize = recordsize + type.size();
```

is used.

```
            // offset is negative of current location
        enterSymbol (new OffsetSymbol(name, - recordsize, type));
        }
}
```

5.4.5 Nested Symbol Tables

Both functions and class definitions introduce new scopes that are nested within other scopes. This naturally means that both types of symbol tables will have references to the symbol tables for the surrounding scope. We can abstract the common features into a class NestedSymbolTable:

```
class NestedSymbolTable extends RecordSymbolTable {
    protected String name;
    protected SymboTable surroundingContext;

    public NestedSymbolTable (int is, String n, SymbolTable sym)
        { super(is); name = n; surroundingContext = sym; }

    public String mangledPrefix ()
        { return surroundingContext.mangledPrefix + name + "_"; }

    public Type findType (String name) throw parseException {
        if (nameDefined(name))
            return super.findType(name);
        return surroundingContext.findType(name);
    }
}
```

A nested symbol table has a name, and a reference to another symbol table that represents the surrounding context. The name is used in creating a mangled prefix. The function findType is modifed so that all symbol tables will be searched for a type name, not just the current table. The other function used to access elements in a table buildAddr, is different in both class symbol tables and function symbol tables, so will be redefined in the separate subclasses.

5.4.6 Function Symbol Table

Just as arguments can be viewed as a record structure, the local variables held by a function are also a type of record structure. We can use this fact to simplify the implementation of a function symbol table. In addition to maintaining local variables, the function symbol table

will hold the argument symbol table used to access the argument values, and a reference to the symbol table for the surrounding context (see Figure 5.3).

```
class FunctionSymbolTable extends NestedSymbolTable {
    private final static int staticChainOffset = -4;
    private ArgumentSymbolTable argTable;

    FunctionSymbolTable (SymbolTable p, String n, ArgumentSymboltable a)
                // save space for bookkeeping information
        { super(12, n, p); argTable = a; }

    public void enterFunction (String newFunName, FunctionType ft) {
        String internalName = mangledPrefix() + newFunName;
        enterSymbol (new NestedFunctionSymbol(newFunName,
            new GlobalNode(ft, internalName), ft));
        }

    public int argStart() { return 4; }

    public void buildAddr (Ast base, String name) {
            // first see if it is an argument
        if (argTable.nameDefined(name))
            return argTable.buildAddr(base, name);
            // next try our own table
        if (nameDefined(name))
            return super.buildAddr(base);
            // else continue static link
        base = new UnaryNode(UnaryNode.deference, null,
            new BinaryNode(BinaryNode.plus, null,
            base, new IntegerNode(staticChainOffset)));
        return surroundingContext.buildAddr (base, name);
        }
}
```

Function symbol tables maintain a name field, used only in generating the mangled names for internal values, such as function names. When a nested function is entered into a function symbol table, the internal name used for the function is first mangled, which prepends the current function name.

To see how addresses for data fields are generated by the compiler symbol table, assume that the "base" argument given to buildAddr is initially simply a frame pointer node. We first examine the symbol table for the argument list. If the name is defined, then the address is simply the sum of the base and the offset, as will be produced by the argument symbol

table. Otherwise, we examine the table for local symbols, and if found then we generate the correct address.

If the symbol is not an argument nor a local, it still may be a symbol defined in the surrounding context. As we noted in Section 5.3, to access the value in the appropriate activation record we must unwind one link in the static chain. This is performed by adding the static chain offset to the current base, dereferencing the value found at that location, and passing the resulting expression as the base for the next level of symbol table. These intermediate expressions are not given any type value.

5.4.7 Class Symbol Tables

Classes differ from records in two regards. First, a class can be formed using inheritance from another class. Second, member functions, as well as data fields, can be defined as part of a class structure.

```
class ClassSymbolTable extends NestedSymbolTable {
    ClassSymbolTable parent = null;

    public ClassSymbolTable (String n, SymbolTable sym)
        { super(0, n, sym); }

    public ClassSymbolTable (String n, ClassSymbolTable p, SymbolTable sym)
        { super (p.size()); name = n; parent = p; surroundingContext = sym; }

    protected Symbol lookup (String name) {
        if (parent != null) {
            Symbol result = parent.lookup(name);
            if (result != null) return result;
            }
        return super.lookup(name);
        }

    public void enterFunction (String newFunName, FunctionType ft) {
        String internalName = mangledPrefix() + newFunName;
        enterSymbol (new MethodSymbol(newFunName,
            new GlobalNode(ft, internalName), ft));
        }
}
```

Like a function symbol table, a class symbol table has a name, used only in forming the mangled prefix for an internal function name. By redefining the method lookup so that it

examines both the current class table and the parent class table, both the methods nameDefined and buildAddr inherited from SimpleSymbolTable will work correctly with inherited fields.

As we noted in Chapter ??, there are several different techniques that can be used to represent methods. We here implement only the simplest scheme, which is to define a method as a global function, the name mangling reflecting the internal location of the symbol. Some of the other possible implementation techniques are investigated in the exercises.

5.5 Semantic Actions

In this section we describe how semantic actions tie the use of the symbol table to the analysis of a source program. We will assume for the purposes of our illustration that recursive descent is being used as the parsing technique, although similar approaches would be employed with other parsing techniques. We will also, where possible, tie the semantic actions to the grammar for Plat, which is described in Appendix A.

5.5.1 Creating the Global Symbol Table

The global symbol table represents the outermost scope, and is created before parsing begins. It contains those symbols that are characterized as built-in, and hence need not be defined in the program itself.

```
SymbolTable sym = new GlobalSymbolTable();
    // see PrimitiveType for descriptions of basic types
sym.enterType( "int", PrimitiveType.intType);
sym.enterType( "real", PrimitiveType.realType);
```

Thereafter, the current symbol table would be passed as an argument to each semantic routine. As functions and classes are defined, their symbol tables would be passed in place of the surrounding scope symbol tables:

```
void program (SymbolTable sym) throws parseException {
    // use sym to resolve symbol meanings
    ...
}
```

5.5.2 Processing Constant Symbols

The following shows a portion of the symantic routine that processes a constant expression in Plat:

```
void constantDeclaration (SymbolTable sym) throws parseException {
    String name;
        ... // find symbol name
    if (sym.nameDefined(name))
        throw new parseException("name redefinition " + name);
    Ast value;
        ... // find symbol value
        // then add new symbol to symbol table
    sym.enterConstant( name, value );
}
```

A check is first performed to ensure the name has not yet been defined in the current context. If not, then a new symbol table entry is created. A reference to the symbol name will subsequently be replaced by the symbol value (see Section 5.5.7).

5.5.3 Entering Variable Declarations

A nameDeclaration creates a new name/type association. As with constant declarations, a check is first performed to ensure the name is unique in the current context. Assuming the exception is not tossed, the creation of a new symbol is handled by invoking the enterVariable method:

```
void nameDeclaration (SymbolTable sym) throws parseException {
    String name;
        ... // find new identifier name
    if (sym.nameDefined(name))
        throw new parseException("name redefinition " + name);
    Type typ;
        ... // find type value
    sym.enterVariable (name, typ);
}
```

Note that the processing of the method enterVariable will differ depending upon the type of the symbol table passed as argument, which could be a global symbol table, an argument symbol table, a function symbol table, or a class symbol table.

5.5.4 Processing an Identifier as Type

When an identifier is used as a type, we must ensure that the associated symbol is a TypeSymbol. Note that the recognition routine for the grammar symbol type is here defined as a function, which returns a value of class Type.

```
Type type (SymbolTable sym) throws parseException {
    if (lex.isIdentifier()) {
        return sym.findType(lex.tokenText());
    }
    ...
}
```

5.5.5 Entering a new Function Definition

In the Plat grammar, a function definition consists of five parts, the keyword function, the identifier name, list of arguments, return type, and a function body. Once the keyword and identifier have been recogized, we can create the symbol table to be used in recording the argument values. A query on the current symbol table will tell us the starting location for the first argument, which is 0 for global functions, and -4 for nested functions and methods.

```
void FunctionDeclaration (SymbolTable sym) throws parseException {
    // read keyword
    String name;
        ...// read name
    if (sym.nameDefined(name))
        throw new parseException("name redefinition " + name);
        ...
    ArgumentSymbolTable atab = new ArgumentSymbolTable(sym.argStart());
    arguments (atab); // process arguments
    Type retType;
        ...// get return type
    FunctionType ftype = new FunctionType(atab, retType);
        // enter function type into current table
    sym.enterFunction (name, ftype);
        // then go parse function body
    FunctionSymbolTable ftab = new FunctionSymbolTable(sym, name, atab);
    functionBody (ftab);
}
```

Note that the function name is entered as a symbol in the current table, before the new symbol table for local variables is constructed. The argument symbol table is needed both as a component in the type definition (so that procedure calls can be checked against the definition) and as part of the function symbol table (so that argument locations can be properly determined while parsing the function body). At the time the function body is processed, we have a picture similar to the following:

picture

When we return from processing the function body, the symbol table for the function, the part containing entries for local variables, can be eliminated. The argument symbol table, however, continues to exist as part of the type description for the function.

picture

5.5.6 Entering a new Class Definition

Like a function, a class creates a new scope, and hence a new symbol table. A complicating factor is that some classes can have parent classes. The identifier that represents the parent class must represent a class type.

```
void classDeclaration (SymbolTable sym) throws parseException {
    ...
    String name; // read class name
    if (sym.nameDefined(name))
        throw new parseException("name redefinition " + name);
    ClassSymbolTable classSym;
    if (lex.match("extends")) { // have parent
            ... // read parent name
        Type ptype = sym.findType(lex.tokenText());
        if (! ptype instanceof ClassType)
            throw new parseException("expecting class name");
        ClassSymbolTable ptable = ((ClassType) ptype).symbolTable;
        classSym = new ClassSymbol(name, ptable, sym);
        }
    else {    // no parent class
        classSym = new ClassSymbolTable(name, sym);
        }
    ...
    sym.enterType (name, new ClassType(name, classSym));
    classBody (classSym);
}
```

Like a function, a class is both entered in the current symbol table as a part of the new class type name, and passed as argument to the recognizer for the class body.

picture

5.5.7 Processing References

Because of the way that symbol tables have been designed to create the internal representation of values, handing simple references is easy. The starting "base" for a reference is

simply the activation frame pointer. This is passed to the method buildAddr, which will return an abstract syntax tree that represents the address of the named quantity.

```
AST reference (SymbolTable sym) throws parseException {
    AST result;

    if (lex.isIdentifier()) {
        result = sym.buildAddr(new FramePointer(), lex.tokenText());
    }
    ... // handle other types of references
}
```

Regardless whether the name is a symbol, global, local, class data field, or intermediate value that requires a static chain to access, the process of looking up the name in the symbol tables will automatically create an abstract syntax three the represents the appropriate computation.

Lvalues and Rvalues

Note carefully the type attached to the expression returned in the method buildAddr by global or offset symbols. In these cases, the expression represents the *address* of the location being referred to, and not the *value* held in that location. Thus, the type field for the node is not assigned the type associated with the memory location, but is instead an AddrOfType node, which has as a base the type of the memory value. Programming languages often hide this distinction between addresses and values, or assume that the meaning will be made clear by context. Consider, for example, the following statement:

```
i = i + 1;
```

Notice that the variable i is here being used for two very different purposes. The i on the right hand side of the expression is intended to mean the value currently held by the variable, while the same symbol on the left hand side is being used to denote the memory location that we wish to modify. The former is termed the rvalue of the symbol, and the latter the lvalue. (You can remember these by remembering the left and right sides of an assignment statement).

The grammar can be used to determine which of the two possible intepretations for a symbol is intended. If you construct the parse tree for the assignment statement shown above using the Plat grammar described in Appendix A, you will note that the left side is generated directly by the nonterminal reference, while the right side goes through the expression sequence: expression, term, and then reference. In fact, any use of a variable as an expression must go through the nonterminal term. We can therefore use a symantic

action associated with this nonterminal to change an lvalue (returned by reference) into an rvalue.

```
public Ast term (SymbolTable sym) throws parseException {
    Ast result;
    ...
    else if (lex.isIdentifier()) {
        result = reference (sym);
        if (result.type() instanceof AddrOfType) {
                // create unary deref operator
            AddrOfType t = (AddrOfType) result.type();
            result = new UnaryNode(UnaryNode.deref,
                t.baseType(), result);
            }
        }
}
```

The unary operator "dereference" takes an address and converts it into the value stored at the address. In the vast majority of cases this operator is only a compile-time artifact, as it will disappear when we get around to generating code.

Note also that not all references represent addresses, as a reference value could equally well represent a symbolic constant. A conditional statement is therefore required to ensure that only lvalues get translated into rvalues.

Study Questions

1. Need to write some.

Exercises

1. An initialized global symbol represents a global value that has been given an initial value. A symbol for such a value would need to store both the type and the AST for the value. Write the class description for such a value, using inheritance from class GlobalSymbol.

2. Consider the three cases for callers. Show the activation records. Show that in each of these three cases the activation record for the surrounding scope is still found on the activation record stack. Give an example that shows that any other case might not work.