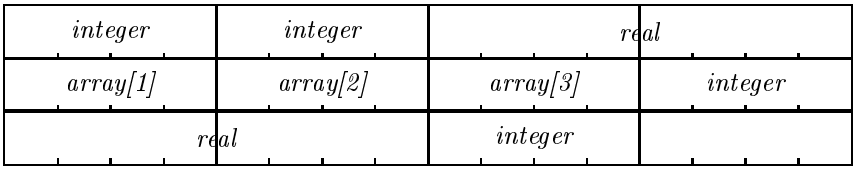
Chapter 4

Types

The concept of a *type* serves various purposes in a programming language. In one sense, a type identifies the set of values that a variable can assume. For example, a variable that is declared as an integer has one range of values, while a character variable has another. A type also identifies the set of permitted operations on a variable, in both syntax and semantics. It is illegal, for example, to subscript a variable that is not explicitly declared as an array. Types tell whether two values can be combined, as in an addition operation or an equality test. In object-oriented languages a parameter type is often used as a form of specification, serving to indicate that the associated parameter values must satisfy certain minimal functionality. Finally, a type helps determine how values are laid out in memory, and the extent of the storage they require.

All of these issues are important to both the programmer and the compiler writer. The last, however, is generally of more importance to the compiler writer than to the programmer. As we noted in Chapter 3, in statically typed languages all the local variables in a procedure are treated as a single block of storage, and each variable is assigned a fixed offset within the block. This is shown by the following picture, which illustrates the local storage for a procedure that contains a variety of values, including an array of three integer elements:



Knowing, at compile time, the size of every variable is a necessary prerequisit to assigning each variable a location. But the values a variable can hold are determined by the variable types. Therefore, each type must know how values are going to be laid out in memory, and how much storage the values will require.

```
interface Type {
        // number of bytes used to store in memory
    public int size();

        // test assignment compatibility
    public boolean canBeAssigned (Type right);

        // internal representation generation routines
    publib Ast pointerDeference (Ast base) throw parseException;
    public Ast fieldName (Ast base, String name) throw parseException;
    public Ast indexExpression (Ast base, Ast index) throw parseException;
}
```

Figure 4.1: The Type interface

The differing requiriements for types are reflected in the range of methods defined for the class Type, shown in Figure 4.1. The most basic method, size, simply indicates the amount of memory used by a value. The method canBeAssigned will be used to match the type of an expression and a target variable, and tell if they are compatible. There are three routines that are tied to the generation of internal representations. These match the syntactic operations in Plat of pointer dereferencing, subscripting, and record (or class) field names. Each expression takes an abstract syntax tree that represents a base and whatever addition information is required by the operation. Each will return a new syntax tree that describes the application of the operation.

Because many of these operations have a common implementation for a large number of types, it is useful to provide a parent class that describes a default behavior for these operations, to be used unless explicitly overridden. This default behavior is provided by the class SimpleType, shown in Figure 4.2. Here compatiblity of two types is determined by the equals method, inherited from class Object. By default, two values are equal if they are the same, however this can be changed by overriding equals. Deferences, subscripts and and field names all generate a syntax error.

4.1 Varieties of Types

In this section we will describe a few of the more common types that are found in programming languages. For each type, we describe how the type is laid out in memory, and how this structure is reflected in the type record maintained by the compiler.

```
abstract class SimpleType implements Type { // base class for simple types

    abstract public int size ( );

    public boolean canBeAssigned (Type right)
        { return equals(right); }

    public Ast pointerDerefence (Ast base) throw parseException
        {
        throw new parseException("cannot dereference this type");
        return null; // will never be executed
        }

    public Ast fieldName (Ast base, String name) throw parseException
        {
        throw new parseException("cannot use field name with this type");
        return null;
        }

    public Ast indexExpression (Ast base, Ast index) throw parseException
        {
        throw new parseException("cannot use subscript with this type");
        return null;
        }
}
```

Figure 4.2: Implementation of the Simple Type class

4.1.1 Primitive Types

For the most primitive types, such as character or integer, the only defining characteristic is a size. These can be represented by instances of the class PrimitiveType. Instances of this class know their size in bytes, and are considered equal only if they are identically the same type. The latter property is provided by the default implementation of the equals operator, so need not be specified.

```
class PrimitiveType extends SimpleType {
    private int sz;

        // static data fields for common types
    public static final Type IntegerType = new PrimitiveType(4);
    public static final Type BooleanType = new PrimitiveType(2);
    public static final Type CharacterType = new PrimitiveType(1);
    public static final Type RealType = new PrimitiveType(8);

    public PrimitiveType (int s) { sz = s; }

    public int size ( ) { return sz; }
}
```

Because the primitive types are frequently referenced from within the compiler itself, it is useful to maintain instances of the basic primitive types as static data fields. The class description illustrates the definition of static data fields for the types associated with integers, booleans, characters, and reals.

On some machines certain primitive types are restricted in the memory locations they can occupy. For example, on many machines integers can only be stored beginning at memory locations that are a multiple of 4, and real numbers only at memory locations that are a multiple of 8. This is a slight annoyance, but not a significant problem. The way out of this difficulty will be investigated in an exercise at the end of the chapter.

4.1.2 Pointer Types

A pointer occupies a fixed amount of storage (here assumed to be four bytes) regardless of the type of object it points to. Two pointer types are considered equal if the type of objects they reference are equal. The method baseType can be used to access the type associated with the values the pointer references.

The action of dereferencing a pointer value is handled by the method pointerDereference. The argument must represent an abstract syntax tree that denotes a pointer value. A new node is made to represent the deference operator. Note that the type attached to this node is an address of a memory location that holds a value of the base type.

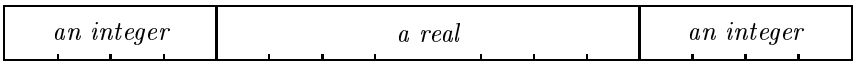
```
class PointerType extends PrimitiveType {
    protected Type base;

    public PointerType (Type t) { super(4); base = t; }

    public boolean equals (Object t) {
        if (! t instanceof PointerType)
            return false;
        PointerType pt = (PointerType) t;
        return base.equals(pt.baseType());
        }

    public Type baseType ( ) { return base; }

    public Ast pointerDereference (Ast base) throw parseException {
        if (! equals(base.type())) {
            throw new parseException
                ("pointer dereference on non pointer value");
            }
        return new UnaryNode (UnaryNode.dereference,
            new AddressType (base), result);
        }
}
```

As we will see if Chapter 5, the representation of a variable is most naturally thought of as the address of the corresponding memory location. That is, the internal representation is actually an expression that describes the address of the value in memory, not the memory value itself. The root node for every subtree in the internal representation must have a type field, and this node is no exception. We could just represent the type of this node with a pointer type, however it will be useful for type checking purposes to distinguish these internally generated addresses from user defined pointer types. For this reason we define a new variety of type node, which is an AddressType. However, from a practical standpoint the implementation of an address type is the same as a pointer type, and thus can inherit all of its behavior from the parent class.

```
class AddressType extends PointerType {
    public AddressType (Type t) { super(t); }
}
```

4.1.3 Record Types

The fields of a record are simply laid out end to end, one after another. For example, a record containing an integer, followed by a real, followed by another integer, could be envisioned as something like the following:



Much of the processing of a record type involves nothing more than keeping track of the field names. These activities involve making a new table of symbols, and are thus discussed in the next chapter when we develop the symbol table handling facilities. In particular, the processing of a field name will be handled by the method buildAddr. We defer further discussion of this routine until the next chapter.

```
class RecordType extends SimpleType {
    private RecordSymbolTable fields;

    public RecordType (RecordSymbolTable rt) { fields = rt; }

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

    public Ast fieldName (Ast base, String name) throw parseException {
        return fields.buildAddr(base, name);
        }
}
```

Structural versus Name Equivalence

There is, however, one issue involving record types that can be addressed here. That issue can be summed up by a question: Under what conditions should two different record types be considered as equal? To make the question more concrete, let us take a specific example, Suppose we have one record that is used to represent cartesian coordinates in a two dimensional space. These can be represented by a pair of integer values:

```
record Point {
    var x : integer;
    var y : integer;
    }
```

Imagine now that we have another class that is used to represent a size in two dimensional space. These can also be represented by a pair of integer values:

```
record Dimension {
    var height : integer;
    var width : integer;
    }
```

Should it be legal to assign, for example, a Point value to a Dimension variable? The semantics of most languages would say no, because the purpose and, more importantly, the *name* of the types do not match. This would correspond to inheriting the default implementation of the method canBeAssigned from class SimpleType. Some langauges, however, determine compatibility based on *structure*, rather than name. In these languages, the two values would be compatible, since they have similar structure. The definition of the canBeAssigned operator that corresponds to this semantic interpretation is explored as an exercise at the end of this chapter.

The issue of structural verus name equalivalence of types will arize once more when we discuss array types, and again in the discussion of function types.

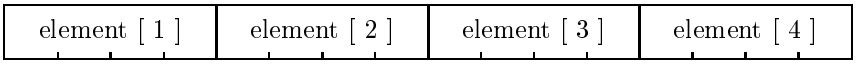
4.1.4 Array Types

An array type is characterized by three quantities. These are the smallest and largest legal index values, and the underlying base type. Each of these is represented by a data field in the array type record:

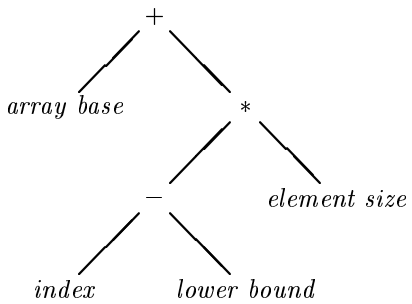
```
class ArrayType extends SimpleType {
    protected int lowerBound;
    protected int upperBound;
    protected Type baseType;

    public ArrayType (int lb, int up, Type bt)
        { lowerBound = lb; upperbound = ub; baseType = bt; }

    public int size ( )
        { return ((upperBound - lowerBound) + 1) * baseType.size(); }

    public Ast indexExpression (Ast base, Ast index) throw parseException {
        if (! index.equals(PrimitiveType.integer))
            throw new parseException
                ("Index expression must be integer type");
            // build the array calculation expression.
        Ast expr = new BinaryOperator(BinaryOperator.minus,
            PrimitiveType.integer,
            index, new IntegerNode(lowerBound)); // subtract lower bound
```

```
        expr = new  BinaryOperator(BinaryOperator.times,
            PrimitiveType.integer,
            expr, new IntegerNode(baseType.size())); // multiply by element size
        expr = new BinaryOperator(BinaryOperator.plus,
            new AddressType(baseType), base, expr); // add offset to base
        return expr;
        }

    public boolean equals (Object t) { ... }
}
```

The question of how array elements should be represented in memory is not as trivial as might at first seem. In the simplest scheme, elements of an array are laid end to end in memory, one after another. An array with four integer elements, indexed 1 to 4, could be envisioned as follows:



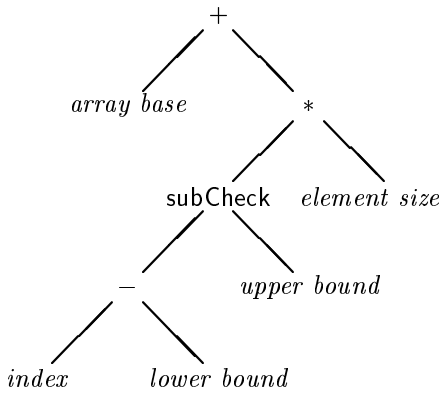
To locate a value in memory, an index value must be convered into an address expression. Doing so requires several steps. The first step is to subtract the lower bound from the index value. The resulting value represents the position of the requested value in sequence, where sequence numbers start with zero, rather than with the declared lower bound. Next, the sequence value is multipled by the size of each array element. This size can be determined by simply invoking the size method on the underlying element type. The result of this computation is a byte-offset for the requested position, relative to the beginning of memory for the array. Thus, to obtain the final memory address, it is sufficient to add this offset to the address of the start of the array. The following diagram illustrates the computation being performed, where base and index are abstract syntax trees that represent the address of the base for the array and the value of the index expression, respectively.



The code to perform this calculation is contained in the method indexExpression, shown earlier. An exercise at the end of the chapter investigates how this technique is extended to cover multiple dimensioned arrays.

Array Bounds Checking

The definition of Plat explicitly states that the validity of index calculations need not be checked at run time. However, many languages are more circumspect on this issue. If one wanted to check the legality of an subscript expression, a logical place to do this is after the lower bound has been subtracted, and before the result is multipled by the element size. The expression at this point represents a zero-based index. It should never be negative, and it should never be larger than the number of elements in the array. We can create a binary operator in our intermediate representation that checks these conditions. The binary operator, which we will call subCheck, takes two integer values. If the left value is negative, or larger than or equal to the right, then an exception will be raised at run time. Otherwise, the value returned by the operator is simply the value of the left argument. Using this operator, the abstract syntax tree representation of a subscript calculation can be given as follows:

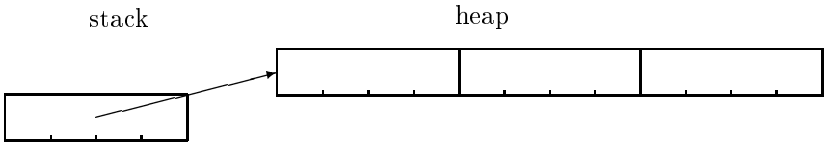


Alternative Array Layout Techniques

There are two major problems with the simple scheme for array layout described earlier. The first is that the size of the array must be fixed at compile time, and cannot be allowed to change during execution. The second is that the values in memory are not "self describing". By that we mean that an examination of the memory contents cannot tell you how many elements the array will hold. This difficulty becomes most relevant when we consider array values as parameters. We will consider this issue in detail in Section 4.2.

The solution to the first problem is to use one layer of indirection. Instead of storing array elements directly in memory, the local memory allocated for an array will be a pointer. This

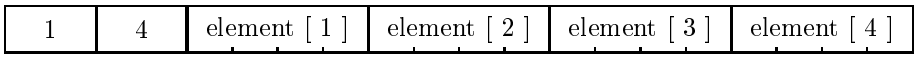
pointer will, at run time, be assigned to the address of the memory elements. The elements themselves will reside in more dynamic storage, typically on the heap. Since the space required in the local storage area for the array value is simply a pointer, the size is easily known at compile time.



Array values in Java are stored using this system, which is why array elements in Java must be explicitly allocated, in addition to declaring the array as a data type:

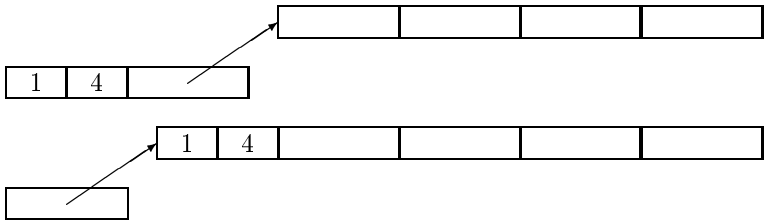
```
int anArray [ ]; // declare array storage, on stack
anArray = new int [ n ];  // allocate array memory, in heap
```

The second problem, making arrays more "self describing", can be solved by maintaining additional information concerning the array value. Depending upon the language being compiled, this information could either be the lower and upper bounds (if both can be specified by the programmer), the lower bound and the number of elements, or simply the number of elements (if the language does not permit the lower bound to be specified). Using the lower bound, number of elements system, an array of four elements might be described in memory as follows:



Array values in Java use this approach. This can be seen by noting that length is a data field recognized by objects of type Array. The length field is stored at the beginning of the array, and encodes the number of elements held by the array.

These two solutions to two different problems are independent of each other, and can be combined in different ways:



In Section 4.2 we will investigate the reason why a compiler writer might want to one of these alternative memory layouts.

Type Compatibility

As we did with records, we can ask when two array types should be considered to be equal. And, as with records, language designers can elect to use either a structural equivalance rule, or a name equivalence rule. Structural equalivalance would say that two arrays are equal if their bounds are equal and their element types are equal. Name equivalance is more restrictive, and says that two arrays are equivalant only if they are exactly the same type.

When types are considered as parameters, the issue of type compatibility becomes more complex. In many languages, such as C and Java, an array type can be specified without describing the array bounds or size. In the following procedure, for example, the array argument values does not have a specified extent:

```
double average (double [ ] values) {
    double sum  = 0.0;
    for (int i = 0; i < values.length; i++)
        sum += values[i];
    return sum / values.length;
    }
```

The value passed as argument to average, on the other, may have a known size. The two types are judged to be equivalent on the basis of their common element types, ignoring the bounds values. Again, we will explore this issue in more detail in Section 4.2.

4.1.5 Class Types

The concept of a *class* extends the idea of a record in several important regards. In addition to data fields (as in a record), a class can also hold function values. A class can be linked to another class by subclassing. And finally, a variable declared as one class can hold values derived from a subclass. (For example, a variable declared as Type in our compiler can actually hold a value that is a PointerType).

The structure of the ClassType reflects these issues. Each ClassType will hold a pointer to a parent class, which is null if the class does not have any parent. A class variable can be assigned another class type if either the types match exactly, or if the argument type is a subclass of the target type. The latter condition can be verified by moving up the parent chain until either a match is found, or until a class is reached that has no parent class.

```
class ClassType extends SimpleType {
    private ClassType parent;
    private ClassSymbolTable fields;

    public ClassType ( ) { parent = null; }
    public ClassType (ClassType p) { parent = p; }
```

```
    public setFields (ClassSymbolTable f) { fields = f; }

    public int size ( ) // return pointer size
        { return 4; }

    public boolean canBeAssigned (Type r) {
        if (! r instanceof ClassType)
            return false;
            // can be assigned if class
            // or subclass
        cr = (ClassType) r;
        while (cr != null) {
            if (equals(cr))
                return true;
                // try parent field
            cr = cr.parent;
            }
        return false;
        }

    public Ast fieldName (Ast base, String name) Throw parseException {
        base = new UnaryNode(UnaryNode.dereference, null, base);
        return fields.buildAddr(base, name);
        }
}
```

The question of how class values are represented in memory is subtle. We will defer the discussion of this point until Section 4.3.

4.1.6 Function Types

When viewed as a type, a function is characterized by the type of values it takes as arguments, and the type of value it returns as its result. These are stored as data fields in the type record.

```
class  FunctionType extends SimpleType {
    private Type returnType;
    private ArgumentSymbolTable arguments;

    public FunctionType (Type rt, ArgumentSymbolTable a)
        { returnType = rt; arguments = a; }
```

```
    public int size ()
    {
        return 8;  // one or two pointer values
    }

    public boolean equals (Object x)
    {
        if (! x instanceof FunctionType)
            return false;
        ft = (FunctionType) x;
            // compare return types
        if (! returnType.equals(x.returnType)
            return false;
            // compare argument types
        ...
            // everything matches
        return true;
    }

}
```

Unlike records, compatibility between function types is usually defined by structure, rather than by name. That is, two functions are compatible if their argument types match, and if their return types match.

Closures

In languages that do not allow functions to be nested, the "value" of a function is simply the address of the first statement in the generated assembly language code. This is nothing more than a memory location, and can therefore be represented by a pointer. The size of a function value is thus simply a pointer size.

The situation is more complicated in languages that allow function nesting. Here a function as value must be able to access data fields that are declared outside the function, as they can potentially be used within the function. In the following code fragment, for example, the function, when executed, must have access to the variable `x` that is declared outside.

```
    var x : integer;
        ...
    function foo ( );
    begin
```

```
        print (x);
    end;
```

In such languages a function value is represented by two quanaities. The first is the pointer to the code. The second is a pointer to a data area, called the *context*. The combination of code and context is referred to as a *closure*. We will discuss contexts and closures in more detail in the next chapter, when we investigate symbol tables.

4.2 Array Types and Parameter Passing

An examination of a specific procedure can be used to illustrate some of the tradeoffs involved in selecting an array memory representation, the detection of programming errors, and the flexibility of language constructs for problem solving. The problem we will use for this purpose is a procedure to compute the average of an array of numeric values.

One extreme in the range of possibilities is illustrated by the language C. In C array values are stored in memory using the simple scheme we outlined earlier. When an array is passed as argument, only the address of the start of the array is actually transmitted as a value. Thus, array values are compact, and array operations extremely efficient.

```
double average ( double [ ] values, int n)
{
    double sum = 0.0;
    for (int i = 0; i < n; i++)
        sum += values[i];
    return sum;
}
```

The disadvantage of this scheme is illustrated by the fact that it was necessary to pass two arguments to this procedure, rather than one. An array value, by itself, does not "know" how many elements it contains. So the programmer must keep track of this information, and pass it along when necessary. An unfortunate consequence is that there is no way for the compiler to check the legality of operations. For example, the compiler does not know whether or not the value of `n` passed as argument really does reflect the actual size of the array. There is no safeguard against the programmer passing a value that is larger than the array, and consequently accessing elements outside the legal range of the array. Because the compiler lacks sufficient information, the validity of subscript expressions cannot be checked.

Errors of this sort are common. Most often they are a reflection of mistakes, however, sometimes they are intentional. The famous internet "worm" in 199X depended upon the use of illegal subscript values to overwrite memory locations with new values. The efficiency of our simple array scheme comes with a dear price, namely a significant lost of error detection and security.

The opposite extreme can be illustrated by the language Pascal. In this language the array type explicitly included the range and element values, even in parameters. A function to compute averages might be written as follows:

```
function average (values : array [ 1 .. 10 ] of real) : real;
    var
        sum : real;
        i : int;
begin
    sum := 0.0;
    for i := 1 to 10 do
        sum := sum + values[i];
    average := sum;
end
```

Security in this case can be easily assurred. The limits of the array are known, and thus the legality of the subscript operations can be checked. However, the cost has been a loss of flexibility. This procedure works only for an array with limits 1 and 10. One cannot use this procedure to compute the average of an array with limits 1 to 20, or 2 to 11, or anything other than 1 to 10. Thus, security has been purchased at the cost of flexibility.

The language C represents an emphasis on efficiency and flexibility, at the expense of security. The language Pascal represents an emphasis on security, at the expense of efficiency and flexibility. To compromise between these two issues involves finding a scheme that is flexible, yet for which security can still be verified. In Java, for example, as in C, the bounds on an array parameter need not be specified:

```
    double average (double [ ] values) {
        double sum  = 0.0;
        for (int i = 0; i < values.length; i++)
            sum += values[i];
        return sum / values.length;
        }
```

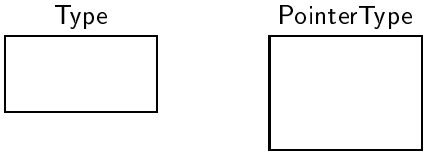
Yet the language definition requires that the validity of subscript index expressions must be checked at run time. Since the array size is not passed as argument, as it was in the C version, the array itself must "know" its own size. This is only possible if the internal representation of the array carries the array size, as well as the array element values.

4.3 Polymorphic Variables

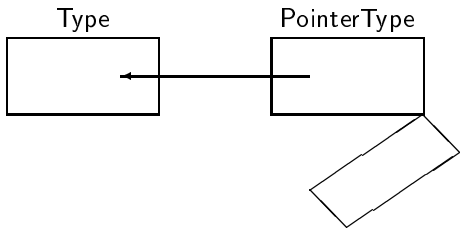
Object-oriented languages must, almost by definition, include the concept of a polymorphic variable. A polymorphic variable is a variable that is declared using one type, but in fact at run-time will maintain a value that is of a different type. Many languages (such as Java) restrict the type of the value to be a subclass of the declared type. We have used this concept repeatedly in the code for our compiler. For example, we have declared variables as maintaining a value of type Type, when in fact they may be holding a PointerType, or an ArrayType.

This difference between the declared and the actual types causes a significant problem for the compiler writer. In particular, polymorphism invariably conflicts with the goal of knowing, at compile time, the amount of storage that a value will require. This is because the subclass value can hold data fields that were declared in the child class, and did not appear in the parent class.

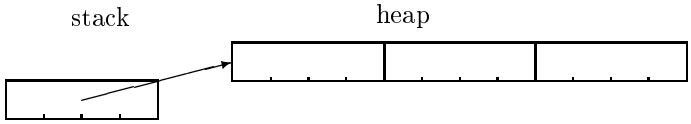
Languages have generally followed one of two approaches to address this difficulty. These two approaches can be described as the slicing technique, and the indirection technique.



C++ is the best example of a language that uses the slicing technique. In C++ the amount of storage allocated for a variable is based on the declared type. This type cannot be extended at run-time. If a value from a subclass is assigned, the "extra" fields in the subclass are simply sliced away. This is a simple approach, but it comes at a cost: the value "is" no longer an instance of the subclass type (since it has lost a significant portion of its "identity"). Thus, the value must be treated as an instance of the parent class. Any methods invoked using the value must come from the parent class, even if they have been overridden in the child class. This can have the effect of making the semantics of the language subtle and difficult to understand. This is all the more so in C++, since this explanation refers to values, and not to pointers or references. Pointers in C++ are polymorphic, and have the behavior one would expect in an object oriented language.



The other alternative is to use indirection. This is the approached followed by Java, as well as many other object-oriented langauges. Using this technique, all variables are represented internally by pointers. The declaration of an object variable only results in the allocation of a pointer space in the local memory area. Since pointers have the same size, regardless of the type of value they refer to, the memory assignment does not cause any difficulty for the compiler writer. At run time the actual storage for the variable is allocated on the heap, rather than in the local memory area. The local variable is then simply assigned the address of the heap storage.



Memory allocation is only one complicating factor raised by object-oriented languages. Another difficulty with inheritance is the ability to override methods, and give them new behavior that is different from the behavior of the parent class. In a later chapter we will explore how this issue is handled.

4.4 Semantic Actions

In this section we describe those semantic actions that involve types. 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.

4.4.1 Creating the Initial Types

The built-in types provided by the language are placed into the global symbol table before parsing begins. See Chapter 5 for a discussion of symbol tables. Because the primary types are used internally in the compiler, it is convenient to store an instance of these type records in a common global variable. This variable can be saved, for example, in a static data field

in class PrimitiveType. Using such values, the code that enters the global type names could be written as follows:

```
    // assume sym holds the global symbol table
sym.enterType( "int" , PrimitiveType.intType);
sym.enterType( "real" , PrimitiveType.realType);
```

Symbol tables nest one within another. By placing the values in the global symbol table, they will thereafter be accessible in all symbol contexts.

4.4.2 Processing the nonterminal type

The nonterminal type matches type expressions. In Plat these are either named types (built-in types or classes), pointer types, or array types. In a recursive descent parser, the synthasized attribute for this nonterminal is returned as the result of the function that parses instances of the nonterminal:

```
private Type type (SymbolTable sym) throws parseException {
    Type result = null;
    if (lex.isIdentifier()) {     // name type
        result = sym.findType(lex.tokenText());
        lex.nextLex();
        }
    else if (lex.match("*")) { // pointer type
        lex.nextLex();
        result = new PointerType(type(sym));
        }
    else if (lex.match("[")) { // array type
        int lowbound, highbound;
        // .. read lower bound, high bound, and brackets
        result = new ArrayType(lowbound, highbound, type(sym));
        }
    else throw new
        parseException("expecting type expression");
    return result;
}
```

If the token represenenting a type is a name, then the symbol table is searched to find an associated type record. The method findType for symbol tables will be explained in more detail in Chapter 5. If, on the other hand, a pointer type is being specified, then the type method will be called recursively, and a new pointer type created using the result. Finally, if an array type is specified, then the lower bound and high bound are read, and the type

method once more recursively called to find the base type, before a new array type record is generated.

Type records will also be created as part of the processing of class and function declarations. These actions will be described in Chapter 5.

4.4.3 Typing Expressions

As we noted in an earlier chapter, the purpose of the parser is to recognize a correct program, and generate the abstract syntax tree that is the internal representation of expressions. Every node in the abstract syntax tree has a type field. A major task of type checking is to ensure that these type fields are properly set.

picture

Constants have self-evident types. Identifiers are typed by their declaration statement, and that type is passed along when the identifier is used in an expression (see Chapter 5). As expressions involving operators are recognized, their internal representation nodes are also typed.

4.4.4 Checking Types

There are several places in the grammar for Plat where expressions of a given type are required. We can simplify the processing of these situations by defining procedures that do nothing more than check a type, issuing an error message if the type is inappropriate:

```
private void MustBeBoolean(Type t) throw parseException {
    if (t.equals(PrimitiveType.booleanType)) return;
    throw new parseException
        ("expecting boolean typed expression");
}

private void MustBeNumeric(Type t) throw parseException {
    if (t.equals(PrimitiveType.integerType)) return;
    if (t.equals(PrimitiveType.realType)) return;
    throw new parseException
        ("expecting numeric type expression");
}
```

The following illustrates the use of these, as well as how new nodes are typed as they are constructed:

```
private Ast term (SymbolTable sym) throw parseException {
    Ast result = null;
    ...
```

```
    else if (lex.match("not")) { // not operator
        lex.nextLex(); // get next token
        result = term(sym);
        MustBeBoolean(result.type());
        result = new UnaryNode(UnaryNode.not, // make new not operator node
            PrimitiveType.booleanType, result);
        }
    else if (lex.match("-")) {{ // unary negation
        lex.nextLex(); // get next token
        result = term(sym);
        MustBeNumeric(result.type());
        result = new UnaryNode(UnaryNode.negation,
            result.type(), result); // same type as argument
        }
    else ...
}
```

4.4.5 Type Coersions

There are many situations where a value of one type can be automatically converted, or coersed, into a different type. A common example is integer values, which will be automatically converted into real numbers if used in an arithmetic expression involving another real, or assigned to a variable declared as real. This situation occurs in several places in the grammar for Plat, so it is useful to create a small utility function that will automatically perform the coersion. The function takes as argument an expression and a type. Should the expression be integer, and the type real, a conversion node is created. Otherwise, the original expression is returned.

```
private Ast intCoercion (Type target, Ast value) {
    if (target.equals(PrimitiveType.realType) &&
            value.type().equals(PrimitiveType.integerType))
        return new UnaryNode(UnaryNode.makeReal,
            PrimitiveType.realType, value);
    return value;
}
```

4.4.6 Typechecking Assignments

According to the description of Plat, an integer value can legally be assigned to a variable declared as real. The assignment operator checks for this case, then checks that the type

of the value is compatible with the target, before finally generating code for the assignment statement.

```
private void assignOrProcedure (SymbolTable tab) throw parseException {
        // get type of left and value of right
    Type left;
    Ast right;
        // first, perform the integer coercison if called for
    right = intCoercison (left, right);
        // then check to see if they are now compatible
    if (! left.canBeAssigned(right.type())) {
        throw new parseException
            ("expression not compatible with target in assignment");
        }
    // then perform assignment
        ...
}
```

We will defer until Chapter ?? discussing the generation of code for assignments.

4.4.7 Typing Operator Expressions

The handling of arithmetic operators is performed in several steps. First, a check is made to insure that both values are numeric type. Then any coersions are applied, as necessary. Finally, a new node is created.

```
private Ast expression (SymbolTable sym) throw parseException {
    Ast left;
    Ast result;
    ... // read left argument
    if (lex.match("+")) { // handle an addition operator
        Ast right;
        ... // read right argument
        MustBeNumeric(left.type());
        MustBeNumeric(right.type());
        left = intCoercion(right.type(), left);
        right = intCoercion(left.type(), right);
        result = new BinaryNode(BinaryNode.plus, left.type(), left, right);
        }
    ... //
}
```

Other operators are handled in a similar fashion.

4.4.8 Checking Parameter Types

The result of the parameterList parsing routine will be a collection of nodes; stored, for example, in a Vector. These can be accessed one by one using an enumeration. From the function description (Section ??) we can obtain an enumeration of the expected argument types. We can check the two against each other using a procedure such as the following:

```
private void checkArguments (Enumeration arguments, Enumeration parameters)
    throw parseException {
    while (arguments.hasMoreElements()) {
        Ast arg = (Ast) arguments.nextElement();
        if (! parameters.hasMoreElements())
            throw new parseException
                ("argument/parameter count mismatch");
        Type typ = (Type) parameters.nextElement();
        if (! typ.canBeAssigned(arg.type()))
            throw new parseException
                ("argument/parameter type mismatch");
        }
    if (parameters.hasMoreElements()) // better be at end now
        throw new parseException
                ("argument/parameter count mismatch");
}
```

4.4.9 References and Dereferences

As we noted in Section ??, references differ from other expressions in that the type associated with a reference is an address of an expression, whereas the type associated with other nonterminals represents the value. The handling of many portions of the nonterminal reference, however, requires discovering the underlying base type for the expression. This can be handled by a small utility procedure, as follows:

```
private Type referenceType (Ast value) throw parseException {
    if (! value.type() instanceof AddressType) {
        throw new parseException
            ("reference expression on non reference");
        }
    AddressType t = (AddressType) value.type();
    return t.baseType();
```

```
}
```

In theory, it is not possible for a reference to hold anything except an address expression. Thus, the parse exception described here can never be seen. However, it costs little to check, even checking for "impossible" conditions.

The use of this routine can be seen in the handing of the pointer dereference operator in the nonterminal reference. The routine pointerDereference was described earlier in Section 4.1.2.

```
private Ast reference (SymbolTable sym) throw parseException {
    Ast result = null;
    ... // handle identifier references
    if (lex.match("*")) { // pointer dereferernce
        lex.nextLex();
        result = referenceType(result).pointerDereference(result);
        }
    ... // process other types of references
}
```

Note that the invocation of the method pointerDereference does two things. First, it performs type checking, since an error message will be issued if the expression type is not pointer. Second, if the expression type is pointer, it will generate the abstract syntax tree for the dereference operator. Recall from section 4.1.2 that the type associated with the result will be the the address of the value referenced by the pointer.

4.4.10 Handing Array Expressions

To convert an array subscript expression into an address, the lower bound for the array must be subtracted from the index expression, and the result multiplied by the size of the base elements. This entire expression must then be added to the address that represents the start of the array. This computation is performed by the array type itself, in response to the method indexExpression. Note once again that the address type must be stripped away before we reach the type that represents the array itself.

```
private Ast reference (SymbolTable sym) throw parseException {
    Ast result = null;
    ... // handle other types of references
    if (lex.match("[")) { // subscript expression
        Ast index;
        ... // read the index value expression and closing bracket
        result = referenceType(result).indexExpression(result, index);
    }
```

```
    ... // process other types of references
}
```

If you refer back to section 4.1.4 you will note that the type attached to the resulting expression is an address of the base type, not a value of the base type.

4.4.11 Handing Field References

Field access expressions are permitted only with record or class types. The actual handling of the field is left to the type record itself. As we have seen, this involves trying to find the given field name in the symbol table associated with the base type.

```
private Ast reference (SymbolTable sym) throw parseException {
    Ast result = null;
    ... // handle other types of references
    if (lex.match(".")) { // field access expression
        lex.nextLex();
        if (! lex.isIdentifier())
            throw new parseException
                ("field expression not identifier");
            // go build the field access expression
        result = referenceType(result).fieldAccess(result, lex.tokenText());
        lex.nextToken();
        }
}
```

Study Questions

1. Need to write some.

Exercises

1. A multiple dimension array in Plat is represented as an array which has elements that are themselves an array. Describe the type records that would be constructed for the following declaration in Plat. Assume that integers require 4 bytes of memory. Show the values in the data fields held by each type record.

```
var values : [ 2 : 5 ] [ 1 : 4 ] int;
```

2. Using the declaration given in the previous question, describe how elements will be maintained in memory, assuming arrays are stored using the simple layout described in this chapter.

3. Assume i and j are integer variables. Using circles to represent the base address of values, and the abstract syntax three that represents the values of i and j, draw the abstract syntax representation of the expression:
   ```
   values[i][j]
   ```

4. Assume that one-dimensional arrays are stored using a prefix consisting of two integers, the lower bound and the number of elements in the array.

   (a) Give the new definition for the method size in the class ArrayType.

   (b) Show how memory values will be stored in memory for the array described in question 1.

   (c) Describe how the subscript address calculation expression must be changed.

5. In this exercise we will deal with the issue of alignment mentioned in Section 4.1.1. Assume we add a new method to the Type interface, as follows:
   ```
   interface Type {
       ...
       int align (); // return minimum alignment
   }
   ```
   The alignment value will represent the minimum permitted alignment for the associated type. That is, if a real number must be aligned on an address value that is a multiple of 8, then the align function for type real would return the value 8.

   (a) Modify SimpleType and define align as an abstract method.

   (b) Rewrite the PrimitiveType class, so that primitive types are characterized by both a size and an alignment. Implement the method align.

   (c) How will alignments change the way that array elements are stored? Modify the class ArrayType to reflect this.

   (d) Provide implementations of align for each of the other type records.

   In an exercise in the next chapter we will explore how alignments change the actions of the symbol table.

6. Suppose one wanted to define equality for record types using structural equality semantics (see Section 4.1.3). Using the fact that the RecordSymbolTable provides a method that returns an Enumeration of type fields, provide an implementation of the structural equality operator.