Chapter 4 - Names, Bindings, and Types

The fundamental language concept introduced in this chapter is that of bindings. In Section 4.1, the binding of an object to its name is discussed. Section 4.2 describes how a data object is bound to values, including expression evaluation, types as sets of permissible values, and issues involving type conversion and subtyping. Section 4.3 describes in detail issues related to the common types in programming languages. Execution units are defined in Section 4.4 in relation to their use as scopes of binding.

4.1 Bindings and names

4.1.1 Bindings

The abstraction that is central to the imperative model is the data object. We view a data object as a four-tuple, \((L,N,V,T)\) where \(L\) is the location, \(N\) is the name, \(V\) is the value, and \(T\) is the type of the object. We call the determination of one of these components a binding as in the binding of a component to a specific object. The implication here is that different bindings can be initialized and changed at different times for a given data object.

Figure 4.1 shows a visualization of a data object and its bindings. The four bindings are all represented as lines from the data object to the corresponding component objects to which it is bound. The storage space, from which location bindings are selected, is the set of virtual storage locations available within the computer system on which the program will be executed. This space is completely invisible to the programmer, who does not need to know the specific location to which the data object is bound, but only if and when the binding takes place.

The time at which a binding takes place is often an important consideration. There are three times when bindings can typically occur:

1. **compile time** - when the program is being translated into machine language.
2. **load time** - when the machine language program generated by the compiler is being assigned to particular locations within the storage space of the computer.
3. **run time** - when the program is being executed.

Usually the location binding occurs at load time. This is a natural time for this binding to occur since it is the time when memory locations are assigned to the program’s instructions and data. We will see later that in some circumstances bindings to locations can occur during run time as well.

4.1.2 Identifiers

The identifier space of a language is the collection of all possible names that can be given to data objects. In addition to data objects, program units are also bound to names selected from the same identifier space. We consider this binding later.

In this text, we highlight the Java implementation of language components in boxes to provide examples of language concepts as they are introduces. The boxes also provide you with a template for expressing these components for new languages that you might encounter. The laboratory exercises frequently ask you to do this.

The accompanying box defines the identifier space for Java as the set of all character strings of length 1 or longer beginning with a non-digit and containing only letters, digits, dollar
Figure 4.1 Data object and bindings
signs, and underscores. Where possible, we use EBNF notation to express language components in order to avoid ambiguity.

Name binding for a data object typically occurs at compile time at the point where a declaration is encountered by the compiler.

Note that name binding becomes a more complex issue when one is dealing with aggregate data objects such as arrays and records. Such aggregate data objects, although bound to a single name, are bound to multiple locations. In addition, although the individual components are data objects themselves with type, value, and location bindings, they do not have a binding to a simple name in the identifier space. Rather, each component is identified by a compound binding of some form.

For example, if ITEM is an array variable, then the simple name ITEM is bound to several locations corresponding to the collection of component data objects of the aggregate object ITEM. Moreover, the compound name ITEM[0] is bound to a single data object. We ignore such complications in our present discussion by limiting our attention to simple, or scalar, data objects with simple name bindings.

Discuss: What are some considerations in choosing between a more restrictive or less restrictive identifier space?

Reinforce: Which of the following are valid Java identifiers? If invalid, give the reason.

a. x2c
b. A$$
c. _2C
d. a_very_long_name!
e. $20
f. $20$
g. $20.50

Reinforce: Suppose a language followed the same identifier naming convention as Java except the underscore character, when it occurred, must have at least one character before and after it. In other words, underscore cannot begin or end an identifier. What would the EBNF be for such an identifier rule?

Research: For a language or languages specified by your instructor, express its identifier space. Also, determine to what types of objects identifiers can be bound.
4.2 Data Types: Values and Operations

4.2.1 Types

The type space of a language is the set of all possible types that can be bound to a simple data object. Each type is, itself, a space of possible values to which a data object of that type can be bound and a set of operations that apply to objects of that type. Therefore, the type and value bindings are two phases of the same binding, with the type binding restricting the possible values to which an object can be bound and defining the set of operations that can be applied to the data object. Later, value bindings bind specific values within the type’s value space to the object. A data object is usually bound to its type at compile time through a type declaration in a programming language. Type declarations may be either explicit or implicit.

One interesting feature of final variables in Java is that they need not be assigned a value in the declaration. Such a variable is a blank final, can only be assigned a value once, and must be assigned a value before it is referenced.

To illustrate the effect of a declaration, consider the following sample declarations:

```java
int A;
int B = 1;
final int C = 1;
```

The effects of these three declarations on bindings at compile time are indicated in Figure 4.2. Double lines indicate bindings that will hold for the life of the data object. Single lines indicate bindings that may be modified at run time.

**Discuss:** Why must value binding occur after type binding?

**Reinforce:** For each of the following Java statements, indicate which are valid data object declarations. For those that are valid, draw the binding diagrams as in Figure 4.2. For those that are invalid, state why.

- a. `final int x = 5 + 1;`
- b. `int x, y, z;`
- c. `int a, b = 2;`
Figure 4.2 (a) Sample Declaration: `int A;`
Figure 4.2 (b) Sample Declaration: \texttt{int B = 1;}

![Diagram of storage space, type space, and identifier space with connections labeled Type Binding, Value Binding, Location, Name, Data Object, Integer, Operators, Storage Space, Identifier Space]
Figure 4.2 (c) Sample Declaration: final int C = 1;
d. int final a = 2*4;

Reinforce: The following Java declaration may or may not be valid depending on one circumstance. What is that circumstance?
final int a = b;

Laboratory: In Java, object declarations may occur anywhere in the program as long as they appear before the first reference to the object being declared. See if this is the case for some other language assigned to you by your instructor.

Research: How are constant identifiers (like Java final) declared in your assigned language?

4.2.2 Operations, Functions, and Expressions

Another important consideration is the syntax a language uses to specify the operations performed on data objects. In general, operators are of two types, monadic and dyadic. Monadic operators have one operand, whereas dyadic operators have two. The standard format for monadic operators is prefix form, with the operator preceding the operand:

<Monadic Operation> ::=  <operator> <operand>

Some monadic operators are postfix operators and take the form

<Postfix Monadic Operation> ::= <operand> <operator>

The dyadic operations are commonly expressed in infix form, with the operator between the two operands:

<Infix Dyadic Operation> ::= <operand> <operator> <operand>

Functions are another form of operation but can have any number of operands. The form of function calls is typically the prefix form, using parentheses. This is expressed by

<Function call> ::=  
  <function identifier>([<operand> {,<operand>}]])

This specifies that functions with no operands are called by their identifier followed by an empty parenthesis pair. Some languages specify a parameterless call by omitting the parentheses altogether.

A further consideration in the evaluation of expressions is the order in which operations are performed. For example, the expression

6 + 2 * 3

could be evaluated as 24 if the addition is performed first. Alternatively, it could be evaluated as
if the multiplication is performed first. The most common rules for determining order of evaluation in a language with, for example, operations +, −, *, /, and %, is

1. Operations inside parentheses are performed first.
2. Next, operations are performed in the following order:
   first: *, /, and %
   second: + and −
3. Operations at the same level in step 2 are performed from left to right.

When the operands in an expression are themselves expressions and when those operands have side-effects, the order in which the operand expressions are evaluated becomes a factor. For example, consider the following expression in Java when a is initially zero:

(a++) - a

If the right-hand operand is evaluated first, the result is 1, while if the left-hand operand is evaluated first, the result is 0. Languages will usually perform left-to-right evaluation of operands for a dyadic operator.

A similar consideration is the order of evaluation of function operands. Again, this is usually done from left-to-right, so \( f(a++, a--, a) \) uses operand values 6, 7, and 6 while \( f(a, a--, a++) \) uses operand values 6, 6, and 5.

These rules, which are summarized in the accompanying box, can be represented in EBNF as previously described in Chapter 2 but we do not include that detail here.

Expression Evaluation
Language: Java

Dyadic Operators: <operand> <operator> <operand>

Order determined by:
1. parentheses
2. precedence
3. left-to-right if equal precedence
Postfix Monadic Operators: <operand> <operator>
Prefix Monadic Operators: <operator> <operand>

Although this precedence convention is common among programming languages, it should be noted that some languages use other strategies for expression evaluation, such as prefix notation for dyadic operations, no precedence levels, or right-to-left evaluation among operators of equal precedence.

Discuss: Languages that use prefix form for dyadic operators do not require parentheses. Why is this so?

Laboratory: By experimentation, determine the rules your assigned language uses to determine order of operation application.
Laboratory: Using experimentation, determine the order that your assigned language evaluates function parameters.

Reinforce: The language APL considers all operators to be of equal precedence and evaluated right to left. What would such a scheme evaluate for the following expressions? Rewrite each one in equivalent postfix form.

(a) $6 \times 2 + 3$
(b) $2 + 3 \times 6$
(c) $4 - 2 - 1$
(d) $4 - 1 - 2$

Reinforce: Operators that are defined on more than one type are called overloaded operators. Identify the operators that are overloaded in Java.

4.2.3 Type binding and type checking

Although types are bound to data objects at compile time in most languages, it is possible for this binding to occur at run time as well. The languages APL and Smalltalk implement such dynamic binding.

Whereas declarations are used to bind data objects to types at compile time, dynamically typed languages often need no declarations. These data objects in a chameleon-like way, take on the type of the value to which they are currently bound. Therefore, whenever such a data object is bound to a different value, it takes on the type of that newly bound value at that time.

Type checking is the process of determining the type of a specified data object. Such checking, when automated, can be of great assistance in the detection and prevention of errors. Type checking can occur at compile time, at run time, or not at all. A language is said to be strongly typed if all type checking that can feasibly be done at compile time is done then, and all other type checking is done at run time. Pascal is nearly strongly typed but not quite because of two exceptional cases: subrange types and variant records.

Consider, for example, the following Pascal program:

type soft = record
  case test:boolean of
    true : (first:1..20);
    false : (second:char);
  end;

var x,y:soft;
  c:char;

begin
  ...
  c := x.second;
  y.first := 2 * x.first;
Pascal violates strong typing in this program for two reasons. First, when \texttt{x.second} is used, Pascal is unable to determine at compile time that the second variant part of \texttt{x} is in effect, and hence cannot guarantee type compatibility. Second, at the assignment statement to \texttt{y.first}, there will be a type violation if \texttt{x.first} is greater than 10, and it is not possible to determine this at compile time. In both of these cases a run time check could be made, but Pascal does not provide it. Ada, on the other hand, does provide for run-time checking in both of these situations and therefore is strongly typed.

There are two possible alternatives for those situations where types cannot be checked at compile time. First, the types might not be checked at all, which is what happens in the preceding cases for Pascal. This situation places the burden of type checking on the programmer and can lead to serious undetected errors. The second possibility is that the type may be checked at run time. Such dynamic type checking can be expensive in terms of execution time, because a check must be performed every time a data object is referenced. It is also more costly in its use of memory because a type indicator must be stored as a part of every data value.

**Dynamically typed languages** bind types to data objects at run time and permit those bindings to be changed. **Statically typed languages** bind types to data objects at compile time and hence can check types at either compile time or run time. Java is a dynamically typed language in ways that we will see later, but it is also strongly typed.

\textit{Laboratory: Investigate whether your assigned language is statically or dynamically typed.}

\textit{Laboratory: For the eight Java types (excepting reference), determine which types are subsets of which other types.}

\textit{Reinforce: Consider the following Ada-like declarations:}

\begin{verbatim}
    type T1 is INTEGER range 0..10;
    type T2 is T1;
    C,D : T2;
    E,F : T1;
\end{verbatim}

Which variables are of equivalent type using

\begin{verbatim}
a. Domain equivalence
b. Name equivalence
c. Declaration equivalence
\end{verbatim}

\subsection*{4.2.4 Type conversion}

Another major issue in dealing with types is the way a data object of one type is converted to another type if the two types are mixed in the evaluation of an expression. Operators require their operands to be of specified types. This includes the assignment operator whose left operand is the result variable and whose right operand is an expression that represents a value to be bound to the result data object. The types of the two operands are said to be consistent if there is a logical way that a value of the actual type can be converted to the type that the operator expects.
The two common strategies for converting an operand to a consistent type are implicit and explicit conversion. Implicit conversion is often called type coercion. Such coercions may occur automatically when certain type mixtures occur. For example, real and integer operand mixtures usually result in the integer operand being converted to a real of equivalent value, if possible. This result is, indeed, often possible because many integers have an equivalent real representation. The only exception to this case occurs when an integer variable is the left operand of an assignment. In this case, the real is truncated to an integer before assignment occurs, because an opposite coercion would require changing the type binding of the target data object, a change that is not normally permitted because the target object is a variable. Languages that permit implicit coercions provide a list of all pairs of types for which implied coercion is permitted. Non permissible pairs are flagged as errors.

The second strategy, explicit type conversion, makes the implicit mixing of types illegal. Instead, explicit functions are required to specify the conversion from one type to another. These functions can be given unique names for each conversion, such as INTEGER_TO_REAL, or, as in Ada, may permit conversion by a function name that matches the name of the target type and that will accept operands of any conversion-compatible type. For example, in Ada, the function FLOAT converts from any allowable type to float type. Such conversions are normally allowed between derived types and their parent types or between various numeric types.

Another common method of explicit type conversion is casting. This method, used by Java, C, and C++, this permits conversion between compatible types by explicitly placing the target type name in parentheses before the expression that is to be converted. For example, if \( i \) is a variable of type int and \( f \) is a variable of type float, the following statement will convert the value of \( f \) to int and store it in \( i \).

\[
i = (\text{int})f;
\]

This conversion will truncate the value of \( f \) to the largest int less than or equal to \( f \).

Questions often arise as to what actually constitutes different types and when there is a need for type conversion. We will give our illustrations in Ada, as that language has great flexibility in the definition of types. Consider the following declarations in Ada:

```ada
type T1 is integer range 0..10; -- defines type T1
type T2 is integer range 0..10; -- defines type T2
A: T1; -- declares A to be of type T1
B, C: T1; -- declares B and C to be of type T1
D: T2; -- declares D to be of type T2
```

The question is, which of the variables \( A, B, C, D \) are considered to be of the same type? Another way of stating this is to ask which of the following assignments are legal in Ada, which always requires explicit type conversion.

```ada
A := B;
A := D;
B := C;
```

Three possible definitions of type equivalence are considered here in order of increasing
restrictiveness:

1. **Domain equivalence**: Two data objects are of equivalent type if they have the same domain of possible values associated with their types. This is also known as *structural equivalence*. Under domain equivalence, A, B, C, and D are all of equivalent type and can be operands that share the same operator without any type conversion.

2. **Name equivalence**: Two data objects are of equivalent type if they are typed by the same name. Under name equivalence, A, B, and C are all of equivalent type, but D is not, because it was bound to a type T2, which differs in name from T1, the type of the other three variables. This is the type equivalence implemented in Ada.

3. **Declaration equivalence**: Two data objects are of equivalent type if they are bound to their type in the same declaration. Under declaration equivalence, only B and C are of equivalent type, because they are bound to a type in the same declaration.

The previous discussion has ignored the issue of **anonymous types**, that is, those types associated directly with variables through declaration without being given a name. For example, in Ada we might declare

```plaintext
E, F : array (1..10) of INTEGER;
G   : array (1..10) of INTEGER;
```

The interpretation of these declarations using domain equivalence results in E, F, and G all being of equivalent type. Declaration equivalence causes E and F to be equivalent types, but G is of a non-equivalent type. However, the interpretation of name equivalence for anonymous types needs more careful definition. The rule used by Ada is no two objects of anonymous type are name-equivalent. This means E, F, and G are all considered to be of different types under name equivalence.

The type issues just discussed, as they are addressed in Java, are summarized in the following box.

**Type Issues**

**Language**: Java

**Types**: boolean, byte, char, double, float, int, long, short, reference

**Type conversion**: Implicit from subset type to its superset. Explicit from superset to subset via casting.

**Type naming**: none

Discuss: In what situations are location bindings made at run time?

Discuss: Which, if any, of the three definitions of type equivalence form equivalence relations in the mathematical sense? Prove your answers.

Laboratory: For your assigned language, determine the answers to the following questions:
a. Are there implicit type conversions? Between what pairs of types do such conversions exist?
b. Is type equivalence a relevant issue? If so, is it domain, name, or declarations equivalence?
c. Is type casting or explicit type conversion available? If so, between which types?

Reinforce: Consider the following Ada-like type declarations:

```ada
type S is INTEGER range 1..100;
type T is INTEGER range 1..100;
type U is INTEGER range 1..50;
A,B : T;
C,D : S;
E,F : INTEGER range 1..100;
G,H : U;
I,J : T
```

Assume variables of anonymous type are not considered type equivalent to any other variables. Which variables are of equivalent type if the language uses each of the following?

a. Domain equivalence
b. Name equivalence
c. Declaration equivalence

4.3 Scalar data types

Scalar data types are those data types that can be bound to atomic data objects in a language. An object is considered atomic if it cannot be broken down into component objects. Two classifications of scalar types are built-in and user-defined. Built-in types are included within the language definition and can be bound directly through the declaration of the data object. A user-defined type must be defined by a declaration of the type prior to its use in a data object declaration.

In the following sections, we examine the most common built-in scalar types by listing a number of their properties. The properties we include are the following

1. Parameters
   Some types have parameters that determine, for example, the precision of a real or the subrange of an integer.

2. Declaration Format
   The format of the declaration that binds a type to a data object includes the manner in which type parameters are specified.

3. Domain
   The set of possible values of a type is the domain of that type. This includes the syntax used to represent constants of that type and the interpretation of that syntax.

4. Operations
   Operations defined on the type include operators and functions that are part of the definition of the language.

5. Attributes
   Attributes differ from operators in that they are operations on a type whose results are
properties of the type. In contrast, operators are operations on data objects whose results are other data objects.

6. Conversion
This category includes the permissible conversions between this type and other types and the syntax for specifying those conversions.

7. Predefined Constants
These constants are named constants of the specified type that are included in the language and need not be declared.

8. Implementation
Considerations in implementing a type include the representation of the type and the algorithms used in performing the operations.

4.3.1 Integer type

Some languages have only a single integer numeric type called INTEGER or int, which has as its domain a range of integers. Some languages, such as C, C++, and Java, provide several sizes of integers. These are often 8, 16, 32, and 64 bit representations. Other languages permit the specification of subtypes of the type INTEGER that limit the domain of the type to a subrange of the default integer domain.

The definition of a subrange of integers is so common that it is often included even in languages that contain no general subrange capability. The specification of the subrange type is usually accomplished by supplying parametric values for the lower and upper end of the subrange. For example, in Pascal, a possible subrange declaration is

```pascal
type POSITIVE = 1..MAXINT;
```

The underlying type for the subrange, INTEGER, is implied by the type of constants that make up the range parameters. This is the convention of Pascal, though other languages require the specification of the parent type INTEGER. Ada permits either approach.

The use of such subranges can allow better type checking. Two options are available for type checking on subrange types. The first is to ignore type checking altogether since such checking must be performed at run time when value bindings are made, and run-time checking is time-consuming. The second option is to perform a run-time check every time a new value binding is performed. Some languages give the programmer the option of enabling or disabling run-time subrange checks at the time of compilation.

A description of the properties of integers in Java is given in the accompanying box:
Integer Type
Language: Java

<table>
<thead>
<tr>
<th></th>
<th>byte</th>
<th>short</th>
<th>char</th>
<th>int</th>
<th>long</th>
</tr>
</thead>
<tbody>
<tr>
<td>size in bits</td>
<td>8</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>domain</td>
<td>-128 to 127</td>
<td>-2^{15} to 2^{15}-1</td>
<td>0 to 2^{16}-1</td>
<td>-2^{31} to 2^{31}-1</td>
<td>-2^{64} to 2^{64}-1</td>
</tr>
</tbody>
</table>

Operations:

<table>
<thead>
<tr>
<th>Dyadic</th>
<th>Dyadic</th>
<th>Monadic</th>
</tr>
</thead>
<tbody>
<tr>
<td>int or long result</td>
<td>boolean result</td>
<td>int or long result</td>
</tr>
<tr>
<td>+ addition</td>
<td>&gt; greater than</td>
<td>+ identity</td>
</tr>
<tr>
<td>- subtraction</td>
<td>&lt; less than</td>
<td>- negation</td>
</tr>
<tr>
<td>* multiplication</td>
<td>&gt;= gtr or equal</td>
<td></td>
</tr>
<tr>
<td>/ division</td>
<td>&lt;= lss or equal</td>
<td></td>
</tr>
<tr>
<td>% remainder</td>
<td>== equal to</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt; signed left shift</td>
<td>!= not equal to</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; signed right shift</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;&gt; unsigned rt shift</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp; bitwise and</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bitwise or</td>
<td></td>
</tr>
<tr>
<td>^ bitwise xor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Literals: Integer literals can be expressed in base 8, 10, or 16.
Literals with no suffix are of type int, those with suffix of type long.

<Decimal-Literal> ::= (0 | <NonZeroDigit> {<Digit>} ) [l | L]
<Hex-Literal> ::= 0 (x | X) <HexDigit> {<HexDigit>} [l | L]
<Oct-Literal> ::= 0 <OctDigit> {<OctDigit>} [l | L]
<HexDigit> ::= <Digit> | a | A | b | B | c | C | d | D | e | E | f | F
<Digit> ::= <NonZeroDigit> | 0
<NonZeroDigit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<OctDigit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

Conversion: Implicit conversion: byte→short→int→float→double
4.3.2 Real type

Two possible parameters of real type are the number of significant digits and the scaling factor designating the number of digits to the right of the decimal point. For example, real numbers with 10 significant digits and a scale factor of 2 digits would be able to represent all numbers from

-99999999.99 to 99999999.99

spaced apart by steps of 0.01. The representation with these two parameters is called fixed point, and when available in a language, it is usually provided in base 10 to permit exact decimal representations of quantities such as dollars and cents.

A more common real representation is floating point. This system includes the scale as a part of the value rather than as a parameter of the type. In this case, a pair of values is provided to represent a real value: the mantissa, whose precision is a parameter of the type, and the exponent, whose domain is another parameter of the type. Another important parameter in the building of a floating point type is the radix. This is the implicit base for the exponent. Floating point representation is usually stored with radix 2, taking advantage of the built-in floating-point commands of the machine on which the language is implemented. More specifically, we can define a floating point type by two parameters as follows:

\[
\text{FP}(m,e) \text{ where } m \text{ is the number of binary digits in the mantissa and } e \text{ is the number of binary digits in the exponent}
\]

A typical floating point type might be FP(24,8). This means the mantissa has precision of 23 binary digits plus one for the sign, and the exponent is expressed in 8 binary digits, permitting values of -128 to 127.

Many languages give the programmer no control over the values of the parameters of reals, usually tying them to the particular implementation. Programs written in these languages and use reals are therefore implementation dependent in the precision of their calculations.

Laboratory: For your assigned language, answer the following questions:

a. What is the range of allowable integers?

b. What different size integer types are built-in? Give their names and domains.

c. Does your language support subrange types for integers? If so, what strategy is used for subrange checking?

d. Is there a fixed-point type available in your language?

e. What is the number of bits in the mantissa for the floating point type? in the exponent? If there is more than one floating point type, answer this for all of them.

Laboratory: Using an object-oriented language, write a class for a fixed point type. The class should include a constructor that specifies as parameters the significant digits and the scale factor for the constructed object. All arithmetic operations should be provided as well as conversion from and to integer and floating point types.
Figure 4.3 Reference type bindings

Type Space

Storage Space

Type Binding

Value Binding

Location

Binding

Data Object 1

Type

Binding

Data Object 2

Identifier Space

Name

Binding

Value Binding

Location

Binding

Object Data Location Binding

Type Space

Integer

Operators
Real Type
Language: Java

<table>
<thead>
<tr>
<th></th>
<th>float</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>mantissa precision</td>
<td>24 binary digits</td>
<td>53 binary digits</td>
</tr>
<tr>
<td>Base 2 exponent</td>
<td>-149 to 104</td>
<td>-1075 to 970</td>
</tr>
</tbody>
</table>

Operations:

<table>
<thead>
<tr>
<th>Dyadic</th>
<th>Dyadic</th>
<th>Monadic</th>
</tr>
</thead>
<tbody>
<tr>
<td>float or double result</td>
<td>boolean result</td>
<td>float or double result</td>
</tr>
<tr>
<td>+ addition</td>
<td>&gt; greater than</td>
<td>+ identity</td>
</tr>
<tr>
<td>- subtraction</td>
<td>&lt; less than</td>
<td>- negation</td>
</tr>
<tr>
<td>* multiplication</td>
<td>&gt;= gtr or equal</td>
<td>++ increment</td>
</tr>
<tr>
<td>/ division</td>
<td>&lt;= lss or equal</td>
<td>-- decrement</td>
</tr>
<tr>
<td>% remainder</td>
<td>== equal to</td>
<td>!= not equal to</td>
</tr>
</tbody>
</table>

Discuss: When programming in a language with both fixed point and floating point types, what considerations would be important in deciding which to use for a given data object?

4.3.3 Boolean type

The simplest type is **Boolean type**. Its domain consists of two values, one representing true and one representing false. The operators on this type are the logical operators, including the dyadic operators AND and OR and the monadic operator NOT.

In some situations, it is convenient to use the short-circuit form of these Boolean operators. To understand the role of **short-circuit operators**, consider the following Java boolean expression:

\[(I \neq 0) \&\& (K/I > 6)\]

When the first condition is false, the second condition cannot be evaluated, because a division by zero would result. However, careful observation reveals that when the first condition is false, the second condition need not be evaluated, because the entire expression is guaranteed to be false anyway. A short-circuit AND operator is evaluated in the following way:
A AND B means if A then B else FALSE

This avoids evaluation of B in the case where A is false. Similarly, a short circuit OR can be defined by

A OR B means if A then TRUE else B

This case avoids evaluation of B if A is true, because a true A makes the expression true no matter what the value of B might be, even if it is not computable.

One way of making these short-circuit operators available is for the compiler to automatically generate code that evaluates AND and OR in this way. A better approach is to provide two new operators to carry out the short-circuit boolean evaluations. Java does this by providing the operators && and ||. This permits the programmer to explicitly specify the way boolean operators are to be performed.

Discuss: Why would a language provide both the standard AND and a short circuit AND? When might the standard AND have a different effect?

Laboratory: Determine if your assigned language does short circuit evaluation of logical operators.


4.3.4 Reference type

Thus far data objects of all types we have discussed have a name, a value, and a location bound to them. The reference data type complicates matters somewhat. The value of a reference data object is itself the location of another data object. The definition of a reference variable is illustrated in Figure 4.3.

Here, we see that two data objects are involved: the data object of reference type (data object 1) and the data object to which it refers (data object 2). Reference types (also known as pointer types) are similar to other types in that the data object is bound to a name and location. The value bound to a reference data object, however, is an element of the storage space. The value
Figure 4.4 Reference Assignments

In the diagram:

**Before**
- \( P := Q; \)
- \( P \text{ ALL} := Q \text{ ALL}; \)

**After**
- \( P := Q; \)
- \( P \text{ ALL} := Q \text{ ALL}; \)
space for a reference data object is, therefore, a storage space as shown in Figure 4.3. As with other types, the type and name bindings occur at compile time, the location binding at load time; and the value binding at run time.

The target data object (data object 2) differs quite a bit from those we have seen previously. First, we note that it has no name binding. It is not referenced by its name but rather through the reference data object that refers to it. Secondly, the location and type bindings of this object must occur at run time, because the data object itself does not exist until run time, when the reference data object is assigned a value. The binding of the reference data object to a value implies the existence of the target data object and the binding of this target object to its location.

Reference data objects are useful for constructing linked data structures for which storage is allocated dynamically at run time. This is especially useful for representing structured data whose size is not known beforehand, because it permits the structure to grow as the program is executed.

In most languages, the declaration of a data object of reference type will include as a parameter the type of the object to which it refers. This is not true for all languages, however, as a parameterless declaration is possible if the type of the target data object is specified or implied at run time when the reference data object is bound to a value (which is a location). This is known as dynamic type binding for references.

C++ and Ada both require the specification of a target type at compile time when declaring reference variables. For example, the declaration of a variable of reference type in C++ takes the form

```
TARGET_TYPE *P;
```

This indicates that the variable P will be a reference to data objects of type TARGET_TYPE. The same declaration in Ada is written

```
P : access TARGET_TYPE;
```

In Java, there is no explicit declaration of reference types. Rather, all array and class objects are of reference type, their values being the locations of their target objects. Therefore, Java declares objects of reference type implicitly every time a class or array is declared. We will see how this is done later.

The binding of a reference variable to a value is done in two ways: creation and assignment. Creation of a new target data object and the binding of its location as the value of the reference data object is accomplished by the keyword new in Ada and C++. For example, in C++

```
P = new TARGET_TYPE;
```

would create a new target object and bind P’s value to the location of the new object. In Ada this would be identical except for the use of := for the assignment. Ada also permits value initialization of the target object in the creation statement, whereas Pascal leaves that target object with an unbound value. For example, in Ada the access variable Q declared by

```
Q : access INTEGER;
```
could be initialized to point to an integer with value zero by the statement

\[ Q := \text{new INTEGER'(0)}; \]

Java also uses the keyword `new` for reference value creation. When the object created is a class member, a call on a constructor for the class follows the keyword `new`.

Assignment of reference objects occurs in the usual way, with the data object named on the left side of the assignment operator bound to the value resulting from evaluation of the right side.

A facility for deallocating a reference data object is also a standard feature of reference types. This results in the target locations being returned to the collection of available space for reassignment later in the program execution. In Pascal this is done by the procedure `DISPOSE` and in Ada by the procedure `FREE`, an instantiation of an Ada generic library procedure. Care must be taken in the deallocation of the target object, because such deallocation may leave some reference objects with value bindings to deallocated locations, that is, locations no longer bound to a target data object. Such references are said to be dangling references. A target data object that is no longer bound as a value to any reference data object but that has not been deallocated is known as garbage. Since the target object is not bound to a name, it is impossible to access such garbage elements. The philosophy of Java is to provide no deallocation specification in the language, but to include a garbage collection facility that will periodically identify and deallocate unreferenced storage. Java does provide an option to force garbage collection in addition to its automatic activation.

Dereferencing a reference object is the process of obtaining the value of the target data object through a reference to the name of the reference data object. For example, in C++ a reference variable `P` is dereferenced by writing `*P`, meaning the data object bound to the location that is bound as a value to `P`. Dereferencing in Ada is done by writing `P.ALL`. Therefore, if `P` and `Q` are two reference variables with the same target type,

\[ P := Q; \]

assigns \( P \) the same value as is bound to \( Q \), meaning they point to the same location. On the other hand,

\[ P.ALL := Q.ALL; \]

binds the target object of \( P \) to the same value as is bound to the target object of \( Q \). This difference is illustrated by Figure 4.4.

A final feature of reference variables is the existence of a constant of reference type that points to no location. In Ada and C++, this reference constant is named `NULL`, while in Java it is named `null`.

Reference Type
Language: Java
Reference type is implicit for all classes, interfaces, and arrays
Domain: set of locations of target type data objects
Operations: Dyadic-Result boolean
Allocation: new <constructor call> | new <component type> ↓<positive integer>↓
Dereferencing:
   <reference identifier> . <component name> for classes
   <reference identifier> ↓<index>↓ for arrays
Deallocation: implicit through garbage collection
Predefined constant: null

Laboratory: Does your assigned language support explicit deallocation? If so, what happens when a reference object is deallocated even though another reference object is referencing the same target object?

4.4 Execution units and scope of binding

In this section, we examine the role of units of execution. These units are divisions of the executable program that are grouped together for a number of reasons. The largest unit is the program itself, which is the fundamental executable unit. Programs are frequently divided into smaller units called blocks, and the smallest, indivisible execution unit of a program is called a statement.

Among the reasons that execution units are formed, the reason most closely related to the topic of this section is to identify the scope of bindings.

4.4.1 Statements

The fundamental unit of execution in a programming language is the statement. The statement specifies a single activity, making it roughly the counterpart of a sentence in a natural language. It is also considered to be the smallest translatable unit.

Some statements may contain other statements embedded within their structure. Conditional and iterative statements are examples of this, where the embedded statements constitute the domain of control, a concept we explore further in Chapter 6. We will call such statements nested statements.

In some languages, all statements have the same format. In essence, these languages have only one type of statement. Most languages, however, have different formats for each of several types of statements, permitting more flexibility in expressing different activities. Java, for example, has 13 different types of statements, each with its own syntax.

Statements are also the units of execution that can be separately labeled so they can be referenced from other statements to specify activities such as branching. Labels permit the binding of a name to a statement. The time at which this binding occurs can be either compile time or run time. Compile-time binding, which is the standard practice of Pascal and Ada, permits reference to the labels within the body of the program unit. This works in the same way as the binding of names to data objects and enables reference from elsewhere in the program unit. The alternative is binding at run time. This permits, for example, the binding of a label identifier to a label variable followed by a reference to that label variable. This binding is identical to the binding of data objects and their values. APL and SNOBOL allow this type of label binding.

A programming language may require that each statement have a label (BASIC) or labels may be optional (Pascal, Ada, Java). Labels may be selected from the space of integers (BASIC, FORTRAN, Pascal) or from the space of identifiers (Ada, Java). Labels may be explicitly declared
in order to specify their name binding (Pascal) or implicitly declared when attached to their statements (Ada). The syntax for including a label in a statement may identify the label by position or by punctuation. For example, FORTRAN reserves character positions 1 through 5 for the label of each statement, locating the label positionally. Pascal and Java, on the other hand, separate the label from the remainder of the statement by a colon.

Another major issue with statement syntax is the method used to delimit statements. Perhaps the simplest is to delimit statements by lines in the program text, with each line containing one and only one statement. This has proved to be too restrictive, however, and requires special consideration for nested statements. A more common approach is the use of a punctuation mark to delimit statements. Java, C++, and Pascal all use the semicolon for this purpose, though there is a subtle difference in the meaning of this punctuation in these languages. Java and C++ use the semicolon as a statement terminator, meaning the end of every statement is defined by the appearance of this punctuation mark. Pascal, on the other hand, uses the semicolon as a statement separator, which tends to be more confusing. In Pascal, statements that are not immediately followed by another statement do not require a semicolon--for example, as in

```
...                        ...
  x:=x+1                   x:=0
...                        ...
```

This leads to a great deal of confusion among novice Pascal programmers concerning the appropriate use of semicolons.

The following box provides a summary of these points with respect to Java.

**Laboratory:** Does your assigned language permit an unconditional branch?

**Reinforcement:** The following pseudocode uses a label variable. Determine what this program would print.

```
label home = first;
int i = 1;
first: print i;
if (i < 5)
  i = 2*i;
else
  ...
```
go to home;
}
else
{
  home = second;
  i = i / 2;
  go to first;
}

second: print i;

4.4.2 Blocks

The second level execution unit is the block, a collection of statements grouped together for a specific purpose. There are several reasons for forming blocks of statements:

1. **Scope of Control Structure** - In a conditional structure, it is necessary to form blocks of statements to indicate which statements are executed when a condition is true and which are executed when it is false. It is also necessary to block together statements that form the body of an iteration structure. Ways of expressing these blocks are discussed when control structures are introduced in Chapter 6.

2. **Scope of Procedural Abstraction** - A set of statements is frequently blocked together to form a module that carries out a specific process. Such blocks are commonly in the form of procedures or functions and are useful for implementing top-down design of programs. These blocks have the added advantage of providing reusability of the module. Such blocks will be further discussed in Chapter 7.

3. **Compilation Units** - Statements are also blocked together to form compilation units. These blocks are compiled separately and then merged for execution. The capability of forming such blocks can be a useful tool in the program development process.

4. **Scope of Bindings** - This type of block is the one of interest in this section. It is the block of statements over which specific bindings hold. Such a scope of binding often occurs in conjunction with blocks that serve other purposes as well, such as procedural abstraction and compilation units. In the following sections, we focus on blocks that have the single purpose of defining the scope of bindings.

4.4.3 Scope of name binding

Blocks that define a scope of name binding usually contain two parts:

1. A declaration section that defines the bindings that hold inside the block.
2. An executable section, which contains the statements of the block over which the binding is to hold.

Some languages permit declarations that are embedded within the executable code, but others require that these two functions be found in entirely different sections of the block. We will first consider the case where there are two separate sections. Syntactically, this requires a marker for the beginning of the declaration section, a marker separating the declaration section and the statements, and a marker indicating the end of the block. We will develop a small pseudolanguage for expressing examples in this section. This pseudo-language will use BLOCK to begin the block, BEGIN to separate declarations from statements, and END to end a block. The only other relevant
statements in this pseudo-language are declarations of variables, accomplished by the DECLARE statement, with the scope of the declaration corresponding to the constructed block. The general structure of a block in our pseudolanguage is therefore

```
...  
BLOCK A;  
  DECLARE I;  
BEGIN A  
  ...       /*I from A*/  
END A;  
...  
```

The declare statement in this example is used to bind the name \( I \) to a data object upon entrance to block \( A \). Our pseudolanguage will not bind type or value, because those are irrelevant for the present discussion. The three dots immediately following BEGIN indicate the statements of block \( A \), and the comment to the right lists the bindings that hold in those statements. In this case, a reference to \( I \) is to the binding made in block \( A \). We will use this notation for all of our examples in this section.

Within any block, there are two kinds of bindings: local bindings are specified by the declarations within the block, and nonlocal bindings are specified by declarations outside of the current block. Such nonlocal bindings are commonly inherited from the containing block. Although some languages require a declaration for bindings that are inherited from the containing block, it is customary for a block to implicitly inherit bindings from the environment that directly contains it. This is particularly useful when we consider the possibility of nested blocks—for example,

```
PROGRAM P;  
  DECLARE X;  
BEGIN P  
  ...       /*X from P*/  
  BLOCK A;  
  ...       /*X from P*/  
  BLOCK A;  
  DECLARE Y;  
BEGIN A  
  ...       /*X from P, Y from A*/  
  BLOCK B;  
  DECLARE Z;  
BEGIN B  
  ...       /*X from P, Y from A, Z from B*/  
END B;  
...       /*X from P, Y from A*/  
END A;  
...       /*X from P*/  
BLOCK C;  
  DECLARE Z;  
BEGIN C  
  ...       /*X from P, Z from C*/  
END C;  
```
In this example, we see that each block inherits the bindings of its containing block (including the main program, considered as the outermost block) and adds the bindings of its own declarations. These constitute nonlocal and local bindings respectively.

The preceding scoping policy is known as **lexical scoping** or **static scoping**. It can be summarized as follows:

1. If a name has a declaration within a block, that name is bound to the object specified in the declaration.
2. If a name has no declaration within a block, the name is bound to the same object to which it was bound in the block containing the present block in the program text. If the block has no containing block or the name is not bound in the containing block, then the name is unbound in the present block.

A situation known as **hole-in-scope** arises when a block redeclares a name already bound in the containing environment. In this case, the local declaration overrides the nonlocal binding, making the nonlocally bound data object inaccessible in the present block, resulting in a hole in its scope.

Consider the following example:

```
PROGRAM P;
  DECLARE X, Y;
BEGIN P
  ...
  /*X from P, Y from P*/
  BLOCK A;
    DECLARE X, Z;
  BEGIN A
    ...
    /*X from A, Y from P, Z from A*/
  END A;
  ...
  /*X from P, Y from P*/
END P;
```

The hole-in-scope refers to the fact that even though the X bound in **PROGRAM P** exists during the execution of **BLOCK A**, it is not accessible in **BLOCK A** because the name X is bound by the local declaration there.

Some languages--Ada for example--avoid the hole-in-scope problem by providing a way to specify a nonlocally bound object even when there is an object bound locally to the same name. In the preceding example, although identifier X used within block A would specify the locally bound object, the Ada notation **P.X** could be used to specify the object bound to X by declaration in block (program) P. This feature makes a data object available in the entire block in which it is declared, including subblocks that rebind its name, thus eliminating hole-in-scope.

Now we consider the case where declarations are not specified in a separate section but rather are embedded within the executable part of the block. In this case, the bindings of a containing block are again inherited by its contained block, but those bindings may not be in force...
Upon Entrance to P

P

Upon Entrance to A

A

Upon Entrance to B

B

NULL

Upon Exit from B

A

Upon Exit from A

P

C

Upon Entrance to C

P

NULL

I

J

K

I

J

K
through the entire execution of the contained block. Instead, the new bindings will be in force
only after the declaration has been reached.

This approach can lead to an even more confusing hole-in-scope problem. Consider the
following example:

```plaintext
PROGRAM P;
BEGIN P
  DECLARE X;
  BLOCK A;
  BEGIN A
    ... /*X from P*/
    DECLARE X;
    ... /*X from A*/
  END A;
END P;
```

Here we see two different bindings for \( X \) within \( \text{BLOCK A} \), depending on whether the reference to
\( X \) is before or after the internal declaration. This can be avoided in several ways. One is by not
permitting the redeclaration of an identifier that is bound in the containing block, thus eliminating
hole-in-scope altogether. Another strategy is to permit the redeclaration of an identifier that is
bound within the containing block but consider that identifier to be unbound prior to reaching the
declaration. This strategy would result in replacing the comment /*X from P*/ by /*X
unbound*/ in the preceding example.

An alternative to lexical scoping is called **dynamic scoping**. In this case, names are bound to
objects at run time. We defer discussion of this strategy to Chapter 7.

**Reinforce:** Consider the following block definitions. For each block, determine the
bindings of every bound name. Label each as local or nonlocal to the block where it is
declared.

```plaintext
Program P;
    BLOCK B1;
        DECLARE A,B,C;
    BLOCK B2;
        DECLARE C,D;
    BLOCK B3;
        DECLARE B,D,F;
    BEGIN B3
        ...
    END B3;
    ...
    END B2;
    BLOCK B4;
        DECLARE B,C,D;
    BEGIN B4
```
4.4.4 Scope of location binding

Up to this point, we have considered only the scope of the name binding within a programming language. The assumption we have made is that the data object is bound to its location at load time and remains bound throughout the entire execution of the program. A side effect of this assumption is that upon reentry to a block, the block's locally bound variables can be assumed to have retained their values from the block's preceding execution. If a new location binding is made upon each entry to the block, no such assumption can be made.

Consider the following example in our pseudolanguage:

PROGRAM P;
  DECLARE I;
BEGIN P
  FOR I:= 1 TO 10 DO
    BLOCK A;
    DECLARE J;
    BEGIN A
      IF I=1 THEN
        J:=1;               /*I from P, J from A*/
      ELSE
        J:=J*I;
      END IF
    END A;
  END FOR A;
END P;

In this program, BLOCK A is executed 10 times. In order for the calculation to make sense the last 9 times, we must assume that the variable J, which is local to BLOCK A, retains its value from one execution of BLOCK A to the next. The disadvantage of such a permanent location binding is that the storage for all blocks in a program must be reserved for the entire time the program is in execution.

The alternative is to bind the location as well as the name to the data object at run time upon entrance to a block, releasing that binding when the block is exited. This is called dynamic storage allocation, and the extent of such a binding is the period of time at run time when this location binding holds.

Let us reconsider the example program illustrating hole-in-scope.

PROGRAM P;
  DECLARE X,Y;
BEGIN P

... /*X from P, Y from P*/
BLOCK A;
  DECLARE X,Z;
BEGIN A
... /*X from A, Y from P, Z from A*/
END A;
...
/*X from P, Y from P*/
END P;

Assuming dynamic storage allocation inside block A, the name X is bound to the location of the
local variable. But upon completion of block A, the name X should revert back to its previous
location binding from P. In practice, there can be many levels of nesting, each declaring a binding
to the same name. Each binding must be recalled in reverse order as the blocks are exited.

The implementation of this nesting is accomplished through the use of activation
records. Activation records are records that contain the information about an execution unit that
is required to resume its execution after it has been suspended. We need only a simple activation
record to implement location binding in blocks. This simple record is augmented later to provide
additional capabilities.

For the purpose of location binding in blocks, the activation record must contain locations
only for all locally bound data objects plus a pointer to the activation record of the block's con-
taining block. As each block is entered, its activation record is placed on top of a stack. Similarly,
the activation record is removed from the top of the stack when the block is exited.

Consider, for example, the following program. The progression of activation record stacks
is shown in Figure 4.5.

PROGRAM P;
  DECLARE I,J;
BEGIN P
  BLOCK A;
  DECLARE I,K;
BEGIN A
  BLOCK B;
  DECLARE I,L:INTEGER;
BEGIN B;
... /*I from B, L from B, K from A, J from P*/
END B;
...
/*I from A, K from A, J from P*/
END A;
  BLOCK C;
  DECLARE I,N;
BEGIN C
... /*I from C, N from C, J from P*/
END C;
...
/*I from P, J from P*/
END P;
The use of the stack to locate data objects by name then proceeds as follows:

1. Look in the top activation record in the stack for an object bound to the given name.
2. If not found, look in the activation record pointed to by the containing block pointer for the top activation record, continuing the search through the list of activation records until the appropriate data object is found.

This search can be shortened, because the structure of the stack and of each activation record is known at compile time. Therefore, the compiler could represent the location of a variable by a pair of integers, $(i,j)$, where $i$ is the number of records up the list and $j$ is the displacement of the desired location from the start of the activation record.

For example, in the above program, block $B$ would represent each data object's location by the following pairs:

$I \ (0,0) \ \{0 \ records \ up \ the \ list, \ 0th \ element \ from \ start\}$
$J \ (2,1) \ \{2 \ records \ up \ the \ list, \ 1st \ element \ from \ start\}$
$K \ (1,1) \ \{1 \ record \ up \ the \ list, \ 1st \ element \ from \ start\}$

Finally, we conclude this section with a summary box of the block capabilities found in Java.

Blocks
Language: Java
Syntax: $\{ \{ \text{statement} \} \}$

- Variable Declarations may appear anywhere in the block.
- An identifier is unbound prior to its declaration.
- An identifier is bound as specified in the declaration in all statements following the declaration in the block.
- An identifier may never be redeclared within its block, thus preventing hole-in-scope.

Reinforce: Trace the contents of the stack of activation records for the following sample program written in our pseudolanguage. For each block, give the pair of integers used to locate each bound name.

PROGRAM P;
  DECLARE X,Y;
BEGIN P
  BLOCK A;
    DECLARE X,Y,Z;
  BEGIN A
    BLOCK B;
      DECLARE Y;
    BEGIN B

Page 33
BLOCK C;
    DECLARE X,Y;
BEGIN C
    ...
END C;
BLOCK D;
    DECLARE Z;
BEGIN D
    ...
END D;
    ...
END B;
END A;
BLOCK E;
    DECLARE Z;
BEGIN E
    BLOCK F;
        DECLARE X;
    BEGIN F
        ...
    END F;
    ...
END E;
    ...
END P;

Terms - Chapter 4

binding
data object
location binding
storage space
compile time
load time
run time
identifier space
type space
type declaration
monadic operator
dyadic operator
function
infix
prefix
precedence
static typing
dynamic typing
type checking
strongly typed
type conversion
type coercion
type equivalence
domain equivalence
name equivalence
declaration equivalence
anonymous type
scalar data type
integer type
real type
fixed point
floating point
mantissa
exponent
Boolean type
short circuit operators
reference type
dangling reference
garbage
dereference
enumerated type
statement
compound statement
label
block
local binding
non-local binding
hole-in-scope
static scope
static binding
dynamic scope
dynamic binding
extent
activation record