you, a program in machine language. You may request execution of this program, but a closer look at the manner of program execution might convince you that in fact the program is just data to the interpreter used by the executing computer. In the same vein we may always consider the input to any program equivalently as data to be processed or as a program to be executed.

This equivalence between programs and data can be taken a step further. In languages like C and FORTRAN, the storage representing the executable program is generally separated from the storage containing the data used by that program. But in other languages, such as Prolog and LISP, there is no real distinction. Programs and data are all intermixed, and it is only the execution process that keeps them separate.

2.2.4 Binding and Binding Time

Without attempting to be too precise, we may speak of the *binding* of a program element to a particular characteristic or property as simply the choice of the property from a set of possible properties. The time during program formulation or processing when this choice is made is termed the *binding time* of that property for that element. There are many different varieties of bindings in programming languages, as well as a variety of binding times. We will also include within the concepts of binding and binding time the properties of program elements that are fixed either by the definition of the language or by its implementation.

Classes of Binding Times

While there is no simple categorization of the various types of bindings, a few main binding times may be distinguished if we recall our basic assumption that the processing of a program, regardless of the language, always involves a translation step followed by execution of the translated program:

1. *Execution time* (*run time*). Many bindings are performed during program execution. These include bindings of variables to their values, as well as (in many languages) the binding of variables to particular storage locations. Two important subcategories may be distinguished:

   a. On entry to a subprogram or block. In most languages important classes of bindings are restricted to occur only at the time of entry to a subprogram or block during execution. For example, in C and Pascal the binding of formal to actual parameters and the binding of formal parameters to particular storage locations may occur only on entry to a subprogram.

   b. At arbitrary points during execution. Some bindings may occur at any point during execution of a program. The most important example here is the basic binding of variables to values through assignment, while some languages like LISP and ML permit the binding of names to storage locations to also occur at arbitrary points in the program.

2. *Translation time* (*compile time*). Three different classes of translation tim bindings may be distinguished:

a. Bindings chosen by the programmer. In writing a program, the programme consciously makes many decisions regarding choices of variable names, types fo variables, program statement structures, etc., that represent bindings during trans lation. The language translator makes use of these bindings in determining the fina form of the object program.

b. Bindings chosen by the translator. Some bindings are chosen by the language translator without direct programmer specification. For example, the relative loca tion of a data object in the storage allocated for a procedure is generally handled without knowledge or intervention by the programmer. How arrays are stored and how descriptors for the arrays, if any, are created are another decision made by the language translator. Different implementations of a given language may choose to provide these features in different ways.

c. Bindings chosen by the loader. A program usually consists of several subprograms that must be merged into a single executable program. The translator typically binds variables to addresses within the storage designated for each subprogram. However, this storage must be allocated actual addresses within the physical computer that will execute the program. This occurs during *load time* (also called *link time*).

3. *Language implementation time.* Some aspects of a language definition may be the same for all programs that are run using a particular implementation of a language, but they may vary between implementations. For example, often the details associated with the representations of numbers and of arithmetic operations are determined by the way that arithmetic is done in the underlying hardware computer. A program written in the language that uses a feature whose definition has been fixed at implementation time will not necessarily run on another implementation of the same language; even more troublesome, it may run and give different results.

4. *Language definition time.* Most of the structure of a programming language is fixed at the time the language is defined, in the sense of specification of the alternatives available to a programmer when writing a program. For example, the possible alternative statement forms, data structure types, program structures, etc. are all often fixed at language definition time.

To illustrate the variety of bindings and binding times, consider the simple assignment statement:

$$X := X + 10$$

Suppose that this statement appeared within some program written in a language L. We might inquire into the bindings and binding times of at least the following elements of this statement:

1. *Set of possible types for variable X.* The variable X in the statement usually has a data type associated with it, such as *real, integer,* or *Boolean.* The set of allowable types for X is often fixed at language definition time; e.g., only types

real, integer, Boolean, set, and *character* might be allowed. Alternatively, the language may allow each program to define new types, as in C, Pascal, and Ada, so that the set of possible types for X is fixed at translation time.

2. *Type of variable X.* The particular data type associated with variable X is often fixed at translation time, through an explicit declaration in the program such as *float* X, which is the C designation for a real data type. In other languages, such as Smalltalk and Prolog, the data type of X may be bound only at execution time through assignment of a value of a particular type to X. In these languages, X may refer to an integer at one point and to a string at a later point in the same program.

3. *Set of possible values for variable X.* If X has data type *real,* then its value at any point during execution is one of a set of bit sequences representing real numbers. The precise set of possible values for X is determined by the real numbers that can be represented and manipulated in the virtual computer defining the language, which ordinarily is the set of real numbers that can be represented conveniently in the underlying hardware computer. Thus the set of possible values for X may be determined at language implementation time; different implementations of the language may allow different ranges of possible values for X. Alternatively, it may be determined at load time depending upon the hardware that will execute the program.

4. *Value of the variable X.* At any point during program execution, a particular value is bound to variable X. Ordinarily this value is determined at execution time through assignment of a value to X. The assignment $X := X+10$ changes the binding of X, replacing its old value by a new one that is 10 more than the old one.

5. *Representation of the constant 10.* The integer 10 has both a representation as a constant in programs, using the string 10, and a representation at execution time, commonly as a sequence of bits. The choice of decimal representation in the program (i.e., using 10 for ten) is usually made at *language definition time,* while the choice of a particular sequence of bits to represent 10 at execution time is usually made at *language implementation time.*

6. *Properties of the operator +.* The choice of symbol + to represent the addition operation is made at *language definition time.* However, it is common to allow the same symbol + to be overloaded by representing *real addition, integer addition, complex addition,* etc., depending on the context. In a compiled language it is common to make the determination of which operation is represented by + at *compile time.* The mechanism for specifying the binding desired is usually the typing mechanism for variables: If X is type integer then the + in $X + 10$ represents integer addition; if X is type real, then the + represents real addition; etc.

The detailed definition of the operation represented by + may also depend upon the underlying hardware computer. In our example, if X has the value 2^{49}, then $X+10$ may not even be defined on some computers. In other words, when is the meaning of addition defined? The meaning is usually fixed at language implementation time and is drawn from the definition of addition used in the underlying hardware computer.

In summary, for a language like Pascal the symbol + is bound to a set of *addition operations* at language definition time, each addition operation in the set is defined at language implementation time, each particular use of the symbol + in a program is bound to a particular addition operation at translation time, and the particular value of each particular addition operation for its operands is determined only at execution time. This set of bindings represents one choice of possible bindings and binding times typical of a variety of programming languages. Note, however, that many other bindings and binding times are also possible.

Importance of Binding Times

In the analysis and comparison of programming languages in the following chapters, many distinctions are based on differences in binding times. We shall be continuously in the process of asking the question: Is this done at translation time or at execution time? Many of the most important and subtle differences among languages involve differences in binding times. For example, almost every language allows numbers as data and allows arithmetic operations on these numbers. Yet not all languages are equally suited for programming problems involving a great deal of arithmetic. For example, while both ML and FORTRAN allow one to set up and manipulate arrays of numbers, solving a problem requiring large arrays and large amounts of arithmetic in ML would probably be most inappropriate if it could also be done in FORTRAN. If we were to try to trace the reason for this by comparing the features of ML and FORTRAN, we ultimately would ascribe the superiority of FORTRAN in this case to the fact that in ML most of the bindings required in the program will be set up at execution time, while in FORTRAN most will be set up at translation time. Thus an ML version of the program would spend most of its execution time creating and destroying bindings, while in the FORTRAN version most of the same bindings would be set up once during translation, leaving only a few to be handled during execution. As a result, the FORTRAN version would execute much more efficiently.

On the other hand, we might turn around and ask a related question: Why is FORTRAN so inflexible in its handling of strings as compared to ML? Again the answer turns on binding times. Because most bindings in FORTRAN are performed at translation time, before the input data are known, it is difficult in FORTRAN to write programs that can adapt to a variety of different data dependent situations at execution time. For example, the size of strings and the type of variables must

be fixed at translation time in FORTRAN. In ML, bindings may be delayed during execution until the input data have been examined and the appropriate bindings for the particular input data determined.

A language like FORTRAN in which most bindings are made during translation, early in the processing of a program, is said to have *early binding;* a language with *late binding,* such as ML, delays most bindings until execution time.

The advantages and disadvantages of early binding versus late binding revolve around this conflict between efficiency and flexibility. In languages where execution efficiency is a prime consideration, such as FORTRAN, Pascal, and C, it is common to design the language so that as many bindings as possible may be performed during translation. Where flexibility is the prime determiner, as in ML and LISP, most bindings are delayed until execution time so that they may be made data-dependent. In a language designed for both efficient execution and flexibility, such as Ada, multiple options are often available that allow choices of binding times.

Binding Times and Language Implementations

Language definitions are usually permissive in specifying binding times. A language is designed so that a particular binding *may* be performed at, e.g., translation time, but the actual time at which the binding is performed is in fact defined only by the implementation of the language. For example, Pascal is designed to permit the type of variables to be determined at compile time, but a particular Pascal implementation might instead do type checking at execution time. Thus while the definition of Pascal permits compile-time type checking, it does not require it. In general a language design specifies the earliest time during program processing at which a particular binding is possible, but any implementation of the language may in fact delay the binding. However, usually most implementations of the same language will perform most bindings at the same time. If the language is designed to permit compile-time bindings, then to delay these bindings until execution time will probably lead to less efficient execution at no gain in flexibility, such as the above example of execution-time type checking in Pascal. It ordinarily is expedient to perform the bindings at the earliest possible moment.

One additional caution is needed, however. Often seemingly minor changes in a language may lead to major changes in binding times. For example, in FORTRAN 90 the change to allow recursion modifies many of the binding times of important FORTRAN features. Because binding times are implementation-dependent, we place emphasis on knowing the language implementation. In Part II a number of languages are analyzed. In each case a "typical" implementation of the language is assumed, and the binding times of the various language elements in the context of this implementation are discussed. When approaching your own local implementation of the same language, it is important to ask about the binding times in that implementation. Are they the usual ones, or have local modifications to the language caused the usual binding times to be modified?