1.3.2 What is a Programming Language?

The programming paradigms provide a good range of choices to the developer. Yet, the percentage of successful projects is alarmingly low. Success in its simplest form means delivered on time, within budget and meeting the quality criteria adopted by the developer. The top reasons cited are all related to managing the process of software development.

The most difficult problem area is acknowledged to be the definition of the system. Elic stakeholder requirements has never been easy. The stakeholder is usually not fully aware of needs and evolves during the development of the project. As a result, as much as 50–60% of the tem definition is likely to change during the development. There are some semi-formal methods help the developer in this area. User-centered design is another strategy that attempts to minim the impact of changing definition of the system on development of the code. In this approach, stakeholders are involved in as many activities of problem solving as possible. Ability to wor interdisciplinary teams and excellent communication skills are imperative needs to address activity of problem solving.

The real task of the software developer is to translate the ideas elicited from the stakeholder i code. However, a treacherous chasm yawns early in the software development process. On one s of this gap is the natural language used to describe the needs of the stakeholders and subs requirements. On the other side is the formal programming languages used in software developm for analysis, design, and programming. Bridging this gap has not been easy despite inventing sev programming paradigms. For nearly two decades, programming languages were expected to off large set of features to help the developer bridge the gap.

> Is a programming language a tool for instructing machines? A means for communicating between programmers? A vehicle for expressing high-level designs? A notation for algorithms? A way of expressing relationships between concepts? A tool for experimentation? A means for controlling computerized devices? My conclusion is that a general purpose programming language must be all of those to serve its diverse set of users. The only thing a language cannot be—and survive—is be a mere collection of "neat" features.
>
> —Bjarne Stroustrup, Designer of C++, 1994

Programming languages are essentially carefully designed notations. They are used to specify, orga and reason about the various aspects of problem solving. The designers of programming langua have twin goals:

- making computing convenient for people
- making efficient use of computing machines

The first goal takes the priority. Many obtuse languages emerged as a result of this appro Gradually the interest in inventing new languages gave way to developing tools to describe th These tools supported the development process using the chosen language. The programming pa digms provided an underlying model to verify and validate the program in a reliable manner.

"Von Neumann Model" or "Stored Program Concept" is one such model. This model prov the foundation for the structured programming paradigm. Conventional programming langua based on this model have grown in the direction of supporting a large set of features. They proved be fat and weak in expressing the solution. The act of programming is bifurcated into the "world expressions" and the "world of statements". Provisions are made to represent the real world data "data types" supported by the language. Computations using this set of data types happen as expressions and the sequence control is supported by a set of statements. The focus of th languages is to **implement the design**. There is little or no help offered by the language designer the other activities of problem solving.

sentation in a program, may be directly represented ... brought together by represented by several separate syntactic elements that are brought together lator to produce one virtual computer element.

1.4.2 Language Paradigms

When proponents of various languages meet, the discussions often take the form of political r Great debates ensue about the efficiency of the array declaration in C++ versus the array de tion in Java or the value of interpreting versus compiling a program. In truth, however, the arra larations and translation issues have very little to do with distinguishing between these langu There are often minor syntactic variations that simply reflect the wishes of the language des which have little concrete effect upon the programs written in those languages. We need to deeper to understand how languages are constructed.

There are four basic computational models that describe most programming today: imper applicative, rule based, and object oriented. We briefly describe each of these models.

Imperative languages. *Imperative or procedural languages* are command-driven or statem oriented languages. The basic concept is the machine state, the set of all values for all memory loca in the computer. A program consists of a sequence of statements, and the execution of each state causes the computer to change the value of one or more locations in its memory, that is, to en new state. The syntax of such languages generally has the form

$$\text{statement}_1;$$
$$\text{statement}_2;$$

Figure 1.4(a) describes this process. Memory consists of a collection of *marbles in boxes*, and the cution of a statement (e.g., adding together two variables to get a third) can be represented as acce the memory *locations* (the boxes), combining these values (the marbles) in some way, and storin

1.4.3 Language Standardization

What describes a programming language? Consider the following C code:

```
int i; i = (1 && 2) + 3;
```

Is this valid C? What is the value of i? How would you answer these questions?[1] The following three approaches are most often used:

1. Read the definition in the language reference manual to decide what the statement
2. Write a program on your local computer system to see what happens.
3. Read the definition in the language standard.

Option 2 is probably the most common. Simply sit down and write a two- or three-line progra
tests this condition. Therefore, the concept of a programming language is closely tied into t
ticular implementation on your local computer system. For the more scholarly, a language ref
manual, typically published by the vendor of your local C compiler, can also be checked. Becau
have access to the language standard, Option 3 is rarely employed.

Options 1 and 2 mean that a concept of a programming language is tied to a particular
mentation. But is that implementation correct? What if you want to move your 50,000-line
gram to another computer that has a compiler by a different vendor? Will the program still c
correctly and produce the same results when executed? If not, why not? Often language
involves some intricate details, and one vendor may have a different interpretation from a
yielding a slightly different execution behavior.

However, one vendor may decide that a new feature added to the language may enhance
fulness. Is this legal? For example, if you extend C to add a new dynamic array declaration, c
still call the language C? If so, programs that use this new feature on the local compiler will
compile if moved to another system.

To address these concerns, most languages have standard definitions. All implementati
should adhere to this standard. Standards generally come in two flavors:

1. *Proprietary standards.* These are definitions by the company that developed and ow
language. For the most part, proprietary standards do not work for languages tha
become popular and widely used. Variations in implementations soon appear wit
enhancements and incompatibilities.
2. *Consensus standards.* These are documents produced by organizations based on an
ment by the relevant participants. Consensus standards, or simply standards, are the
method to ensure uniformity among several implementations of a language.

Each country typically has one or more organizations assigned with the role of developing
dards. In the United States, that is the American National Standards Institute (ANSI). Progr
language standards are assigned to committee X3 of the Computer Business Equip
Manufacturers Association (CBEMA). The Institute of Electrical and Electronic Engineers (IE
also may develop such standards. In the United Kingdom, the standards role is assumed

[1] Note that the answers are "yes" and "4," respectively, for the curious.

British Standards Institute (BSI). International standards are produced by the International Standards Organization (ISO) with headquarters in Geneva, Switzerland.

In the United States, standards are voluntary. The National Institute of Standards and Technology (NIST), an agency of the United States government, develops federal standards. These standards are only requirements on vendors that wish to sell products to federal agencies. Private companies are free to ignore such standards. However, because of the size of the federal government, these standards often become adopted, and NIST, ANSI, IEEE, and ISO often work together to develop many of these standards.

Standards development follows a similar process in all of these organizations. At some point, a group decides that a language needs a standard definition. The standards body charters a working group of volunteers to develop that standard. When the working group agrees on their standard, it is voted on by a larger voting block of interested individuals. Disagreements are worked out, and the language standard is produced.

Although it sounds good in theory, the application of standards making is partially technical and partially political. For example, vendors of compilers have a strong financial stake in the standards process. After all, they want the standard to be like their current compiler to avoid having to make changes in their own implementation. Not only are such changes costly, but users of the compiler, which employ the features that have changed, now have programs that do not meet the standard. This makes for unhappy customers.

Therefore, as stated earlier, standards making is a consensus process. Not everyone gets their way, but one hopes that the resulting language is acceptable to everyone. Consider the following simple example. During the deliberations for the 1977 FORTRAN standard, it was generally agreed that strings and substrings were desirable features, since most FORTRAN implementations already had such features. However, there were several feasible implementations of substrings: If M = "abcdefg," then the substring "bcde" could be the string from the second to fifth character of M (M[2:5]) or could be the string starting at Position 2 and extending for four characters (M[2:4]). It could also be written M[3:n] by counting characters from the right. Because no consensus could be reached, the decision was simply to leave this out of the standard. Although not fulfilling most of the goals for a language as expressed by this chapter, it was the expedient solution that was adopted. For this reason, standards are useful documents, but the language definition can get colored by the politics of the day.

To use standards effectively, we need to address three issues:

1. *Timeliness*. When do we standardize a language?

2. *Conformance*. What does it mean for a program to adhere to a standard and for a compiler to compile a standard?

[illegible]

3. *Obsolescence*. When does a standard age, and how does it get modified? We consider each question in turn.

[illegible]

Timeliness. One important issue is when to standardize a language. FORTRAN was initially standardized in 1966 after there were many incompatible versions. This led to problems because each implementation was different from the others. At the other extreme, Ada was initially standardized in 1983, before there were any implementations; therefore, it was not clear when the standard was produced whether the language would even work. The first effective Ada compilers did not even appear until 1987, and several idiosyncrasies were identified by these early implementations. One would like to standardize a language early enough so that there is enough experience in using the language, yet not so late as to encourage many incompatible implementations.

3.1 PROGRAMMING LANGUAGE SYNTAX

Syntax, which is defined as "the arrangement of words as elements in a sentence to show their relationship,"[1] describes the sequence of symbols that make up valid programs. The C statement $X = Y + Z$ represents a valid sequence of symbols, whereas $XY + -$ does not represent a valid sequence of symbols for a C program.

Syntax provides significant information needed for understanding a program and provides much-needed information toward the translation of the source program into an object program. For example, almost everyone reading this text will interpret the expression $2 + 3 \times 4$ as having the value of 14 and not 20. That is, the expression is interpreted as if written $2 + (3 \times 4)$ and is not interpreted as if written $(2 + 3) \times 4$. We can specify either interpretation, if we wish, by syntax and hence guide the translator into generating the correct operations for evaluating this expression.

As with the ambiguous English sentence "They are flying planes," developing a language syntax alone is insufficient to unambiguously specify the structure of a statement. In a statement like $X = 2.45 + 3.67$, syntax cannot tell us whether Variable X was declared or declared as type real. Results of $X = 5$, $X = 6$, and $X = 6.12$ are all possible if X and + denote integers, X denotes an integer and + is real addition, and X and + denote real values, respectively. We need more than just syntactic structures for the full description of a programming language. Other attributes, under the general term semantics, such as the use of declarations, operations, sequence control, and referencing environments, affect a variable and are not always determined by syntax rules.

[1] Webster's New World *Dictionary*, Fawcett, 1978.

3.1.1 General Syntactic Criteria

The primary purpose of syntax is to provide a notation for communication between the programmer and the programming language processor. The choice of particular syntactic structures, however, is constrained only slightly by the necessity to communicate particular items of information. For example, the fact that a particular variable has a value of type *real number* may be represented in any of a dozen different ways in a program—through an explicit declaration as in C, through an implicit naming convention as in FORTRAN, and so on. The details of syntax are chosen largely on the basis of secondary criteria, such as readability, which are unrelated to the primary goal of communicating information to the language processor.

There are many secondary criteria, but they may be roughly categorized under the general goals of making programs easy to read, easy to write, easy to translate, and unambiguous. We consider some of the ways that language syntactic structures may be designed to satisfy these often conflicting goals.

Readability. A program is readable if the underlying structure of the algorithm and data represented by the program is apparent from an inspection of the program text. A readable program is often said to be *self-documenting*—that is, it is understandable without any separate documentation (although this goal is seldom achieved in practice). Readability is enhanced by such language features as natural statement formats, structured statements, liberal use of keywords and noise words, provision for embedded comments, unrestricted length identifiers, mnemonic operator symbols, free-field formats, and complete data declarations. Readability, of course, cannot be guaranteed by the design of a language because even the best design may be circumvented by poor programming. However, syntactic design can force even the best-intentioned programmer to write unreadable programs (as is often the case in APL). The COBOL design emphasizes readability most heavily, often at the expense of ease of writing and translation.

Readability is enhanced by a program syntax in which syntactic differences reflect underlying semantic differences so that program constructs that do similar things look similar and program constructs that do radically different things look different. In general, the greater the variety of syntactic constructs used, the more easily the program structure may be made to reflect different underlying semantic structures.

Languages that provide only a few different syntactic constructs in general lead to less readable programs. In APL or SNOBOL4, for example, only one statement format is provided. The differences among an assignment statement, a subprogram call, a simple goto statement, a subprogram return, a multiway conditional branch, and various other common program structures are reflected syntactically only by differences in one or a few operator symbols within a complex expression. It often requires a detailed analysis of a program to determine even its gross control structure.

Moreover, a single syntax error, such as a single incorrect character in a statement, may radically alter the meaning of a statement without rendering it syntactically incorrect. In LISP errors in matching parentheses cause similar problems; one of Scheme's extensions to LISP is to circumvent this problem.

Writeability. The syntactic features that make a program easy to write are often in conflict with those features that make it easy to read. Writeability is enhanced by use of concise and regular syntactic structures, whereas for readability a variety of more verbose constructs are helpful. C unfortunately has this attribute of providing for very concise programs that are hard to read, although it does have a full complement of useful features.

Implicit syntactic conventions that allow declarations and operations to be left unspecified make programs shorter and easier to write but harder to read. Other features advance both goals (e.g., the use of structured statements, simple natural statement formats, mnemonic operation symbols, and unrestricted identifiers usually make program writing easier by allowing the natural structure of the problem algorithms and data to be directly represented in the program).

A syntax is redundant if it communicates the same item of information in more than one way. Some redundancy is useful in programming language syntax because it makes a program easier to read and also allows for error checking during translation. The disadvantage is that redundancy makes programs more verbose and thus harder to write. Most of the default rules for the meaning of language constructs are intended to reduce redundancy by eliminating explicit statement of meanings that can be inferred from the context. For example, rather than require explicit declaration of the type of every function parameter, ML uses data type induction to derive the type of a function's argument. However, if there is an error in writing such a function, the translator will be unable to detect this error. Because of this effect of masking errors in programs, languages that lack all redundancy are often difficult to use.

Ease of verifiability. Related to readability and writeability is the concept of program correctness or *program verification*. After many years of experience, we now understand that understanding each programming language statement is relatively easy, but the overall process of creating correct programs is extremely difficult. Therefore, we need techniques that enable the program to be mathematically proved correct. We discuss this further in chapter 4.

Ease of translation. A third conflicting goal is that of making programs easy to translate into executable form. Readability and writeability are criteria directed to the needs of the human programmer. Ease of translation relates to the needs of the translator that processes the written program. The key to easy translation is regularity of structure. The LISP syntax provides an example of a program structure that is neither particularly readable nor particularly writeable but that is extremely simple to translate. The entire syntactic structure of any LISP program may be described in a few simple rules because of the regularity of the syntax. Programs become harder to translate as the number of special syntactic constructs increases. For example, COBOL translation is made extremely difficult by the large number of statement and declaration forms allowed, although the semantics of the language is not particularly complicated.

Lack of ambiguity. Ambiguity is a central problem in every language design. A language definition ideally provides a unique meaning for every syntactic construct that a programmer may write. An ambiguous construction allows two or more different interpretations. The problems of ambiguity usually arise not in the structure of individual program elements but in the interplay between the

structures.

For example, both Pascal and ALGOL allow two different forms of conditional statement:

1. **if** *Boolean expression* **then** *statement*; **else** *statement*;
2. **if** *Boolean expression* **then** *statement*;

The interpretation to be given to each statement form is clearly defined. However, when the two forms are combined by allowing *statement*, to be another conditional statement, then the structure

if *Boolean expression*, **then if** *Boolean expression*, **then** *statement*, **else** *statement*,

termed a *dangling else*, is formed. This statement form is ambiguous, because it is not clear which the two execution sequences of Figure 3.1 is intended. FORTRAN syntax provides another example A reference to *A*(*I*,*J*) might be either a reference to an element of the two-dimensional array *A* or call of the function subprogram *A* because the syntax in FORTRAN for function calls and array element is the same. Similar ambiguities arise in almost every programming language.

The ambiguities in FORTRAN and ALGOL mentioned earlier have in fact been resolved in the languages. In the ALGOL conditional statement, the ambiguity has been resolved by changing the syntax of the language to introduce a required **begin** — **end** delimiter pair around the embedded conditional statement. Thus, the natural but ambiguous combination of two conditional statements has been replaced by the two less natural but unambiguous constructions, depending on the desired interpretation:

1. **if** *Boolean expression*, **then begin if** *Boolean expression*, **then** *statement*, **end else** *statement*
2. **if** *Boolean expression*, **then begin if** *Boolean expression*, **then** *statement*, **else** *statement*,

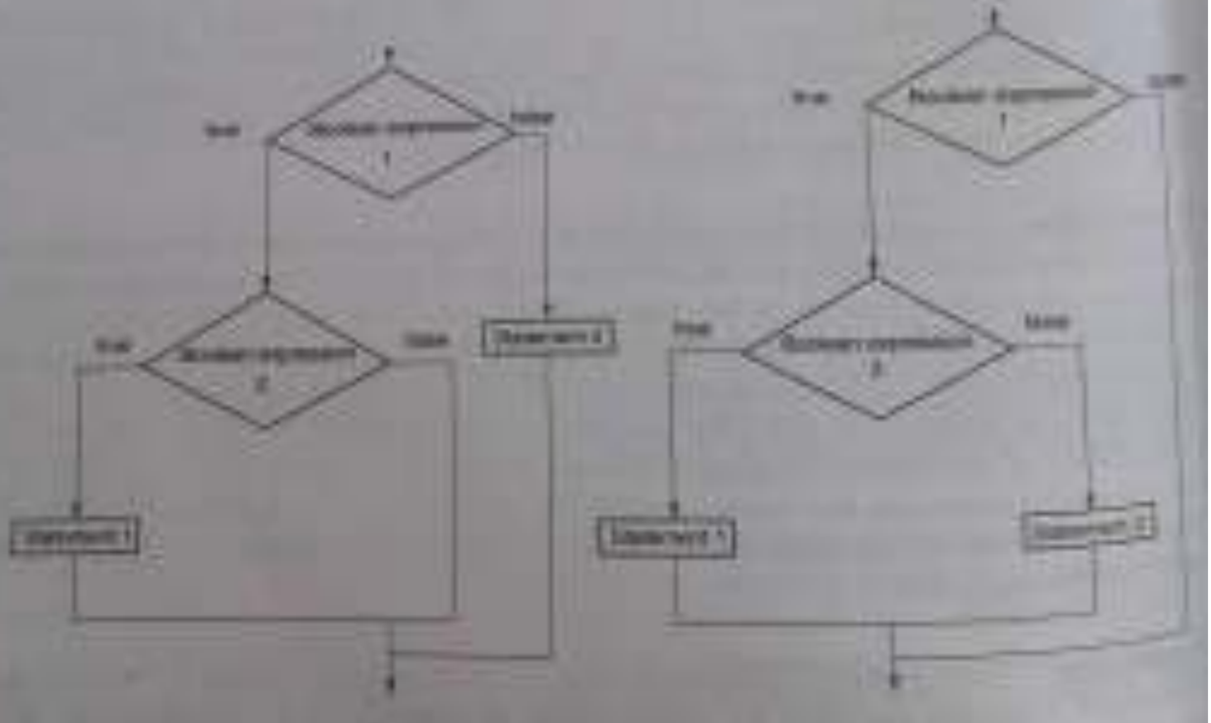


Figure 3.1. Two interpretations of a conditional statement

A simpler solution is used in Ada: Each **if** statement must end with the delimiter **end if**. In C and Pascal another technique is used to resolve the ambiguity. An arbitrary interpretation is chosen for the ambiguous construction—in this case, the final else is paired with the nearest **then** so that the combined statement has the same meaning as the second of the previous ALGOL constructions. The ambiguity of FORTRAN function and array references is resolved by the rule: The construct *A*(*I*, *J*) is assumed to be a function call if no declaration for an array *A* is given. Because each array must be declared prior to its use in a program, the translator may readily check whether there is in fact an array *A* to which the reference applies. If none is found, then the translator assumes that the construct is a call to an external Function *A*. This assumption cannot be checked until load time when all the external functions (including library functions) are linked into the final executable program. If the loader finds no Function *A*, then a loader error message is produced. In Pascal, a different technique is used to distinguish function calls from array references: A syntactic distinction is made. Square brackets are used to enclose subscript lists in array references (e.g., *A*[*I*, *J*]), and parentheses are used to enclose parameter lists of function calls (e.g., *A*(*I*, *J*)).

3.1.2 Syntactic Elements of a Language

The general syntactic style of a language is set by the choice of the various basic syntactic elements. We consider briefly the most prominent of these.

Character set. The choice of character set is one of the first to be made in designing a language syntax. There are several widely used character sets, such as the ASCII set, each containing a different set of special characters in addition to the basic letters and digits. Usually one of these standard sets is chosen, although occasionally a special nonstandard character set may be used, as, for example, in APL. The choice of character set is important in determining the type of input-output (I/O) equipment that can be used in implementing the language. For example, the basic C character set is available on most I/O equipment. However, the APL character set cannot be used directly on most I/O devices.

The general use of 8-bit bytes to represent characters seemed like a reasonable choice when the computer industry went from 6- to 8-bit characters in the early 1960s. Two hundred and fifty-six characters seemed like more than enough to represent the 52 upper and lower case letters, 10 digits, and a few punctuation symbols. However, today the computer industry is much more international. Not many (in fact, few) countries use the same 26 letters. Spanish adds the tilde (~), French uses accents (`), and other characters are present in some languages (e.g., ä, ß, ö). In addition, there are languages like Greek, Hebrew and Arabic with totally different character sets. Representing languages like Chinese or Japanese, where each ideograph represents a word or phrase, requires a character set on the order of 10,000 symbols. Single-byte characters clearly do not work in all such cases, and language implementors increasingly have to consider 16-bit (i.e., 65,536) representations for the character set.

Identifiers. The basic syntax for identifiers—a string of letters and digits beginning with a letter—is widely accepted. Variations among languages are mainly in the optional inclusion of special characters such as . or - to improve readability and in length restrictions. Length restrictions, such as the early BASIC restriction to a single letter and digit, force the use of identifiers with little mnemonic value in many cases and thus restrict program readability significantly.

Operator symbols. Most languages use the special characters + and − to represent the two basic arithmetic operations, but beyond that there is almost no uniformity. Primitive operations may be represented entirely by special characters, as is done in APL. Alternatively identifiers may be used for all primitives, as in the LISP PLUS, TIMES, and so on. Most languages adopt some combination, utilizing special characters for some operators, identifiers for others, and often also some character strings that fit in neither of these categories (e.g., the FORTRAN .EQ. and **, for equality and exponentiation, respectively).

Keywords and reserved words. A keyword is an identifier used as a fixed part of the syntax of a statement (e.g., *if* beginning a C conditional statement or *for* beginning a C iterative statement). A keyword is a reserved word if it may not also be used as a programmer-chosen identifier. Most languages today use reserved words, thus improving the error-detecting capabilities of the translator. Most statements begin with a keyword designating the statement type: READ, IF, WHILE, and so on.

Syntactic analysis during translation is made easier by using reserved words. FORTRAN syntactic analysis, for example, is made difficult by the fact that a statement beginning with DO or IF may not actually be an iteration or conditional statement. Because DO and IF are not reserved words, a programmer may legitimately choose these as variable names. COBOL uses reserved words heavily, but there are so many identifiers reserved that it is difficult to remember them all; as a result, one often inadvertently chooses a reserved identifier as a variable name. The primary difficulty with reserved words, however, comes when the language needs to be extended to include new statements using new reserved words (e.g., as when COBOL is periodically revised to prepare an updated standard). Addition of a new reserved word to the language means that every old program that uses that identifier as a variable name (or other name) is no longer syntactically correct, although it has not been modified.

Noise words. Noise words are optional words that are inserted in statements to improve readability. COBOL provides many such options. For example, in the **goto** statement, written GO TO label, the keyword GO is required, but TO is optional; it carries no information and is used only to improve readability.

Comments. Inclusion of comments in a program is an important part of its documentation. A language may allow comments in several ways: (1) Separate comment lines in the program, as in the BASIC REM statement; (2) delimited by special markers, such as the C /* and */ with no concern for line boundaries; or (3) beginning anywhere on a line but terminated by the end of the line, as the -- in Ada, // in C++ or ! in FORTRAN 90. The second alternative suffers from the disadvantage that a missing terminating delimiter on a comment will turn the following statements (up to the end of the next comment) into comments, so that, although they appear to be correct when reading the program, they are not in fact translated and executed.

Blanks (spaces). Rules on the use of blanks vary widely between languages. In C, the blanks are not significant anywhere except in literal character-string data. Other languages use blanks as separators so that they play an important syntactic role. In SNOBOL4, the primitive operation of concatenation is represented by a blank, and the blank is also used as a separator between elements of a statement (leading to much confusion). In C, blanks are generally ignored, but not always. In early versions of C, the symbol =+ was a single operator, whereas = + represented two operations. To prevent such errors, the current C definition uses the symbol += as the combined operator because + = would be a syntax error.

Delimiters and brackets. A *delimiter* is a syntactic element used simply to mark the beginning or end of some syntactic unit such as a statement or expression. *Brackets* are paired delimiters (e.g., parentheses or **begin** ... **end** pairs). Delimiters may be used merely to enhance readability or simplify syntactic analysis, but more often they serve the important purpose of removing ambiguities by explicitly defining the boundaries of a particular syntactic construct.

Free- and fixed-field formats. A holdover from the early punched-card era of computing is the *fixed field*. A syntax is *free field* if program statements may be written anywhere on an input line without regard for positioning on the line or for breaks between lines. A *fixed-field* syntax utilizes the positioning on an input line to convey information. Strict fixed-field syntax, where each element of a statement must appear within a given part of an input line, is most often seen in assembly languages. Fixed-field syntax is increasingly rare today, and free field is the norm.

Expressions. Expressions are functions that access data objects in a program and return some value. Expressions are the basic syntactic building block from which statements (and sometimes programs) are built. In imperative languages like C, expressions form the basic operations that allow for the machine state to be changed by each statement. In applicative languages like ML or LISP, expressions form the basic sequence control that drives program execution. We discuss expression building more fully in chapter 8.

Statements. Statements are the most prominent syntactic component in imperative languages, the dominant class of languages in use today. Their syntax has a critical effect on the overall regularity, readability, and writability of the language. Some languages adopt a single basic statement format, whereas others use a different syntax for each different statement type. The former approach emphasizes regularity, whereas the latter emphasizes readability. SNOBOL4 has only one basic statement syntax, the pattern-matching-replacement statement, from which other statement types may be derived by omitting elements of the basic statement. Most languages lean toward the other extreme of providing different syntactic structures for each statement type. COBOL is most notable in this regard. Each COBOL statement has a unique structure involving special keywords, noise words, alternative constructions, optional elements, and so on. The advantage of using a variety of syntactic structures, of course, is that each may be made to express in a natural way the operations involved.

A more important difference in statement structures is that between *structured* or *nested* statements and simple statements. A simple statement contains no other embedded statements. APL and SNOBOL4 allow only simple statements. A structured statement may contain embedded statements. The advantages of structured statements are discussed at length in chapter 8.

3.1.3 Overall Program-Subprogram Structure

The overall syntactic organization of main program and subprogram definitions is as varied as the other aspects of language syntax.

Separate subprogram definitions. C illustrates an overall organization in which each subprogram definition is treated as a separate syntactic unit. Each subprogram is compiled separately and the compiled programs linked at load time. Object orientation requires information to be passed among the separately compiled units. The inheritance of class definitions requires that the compiler process some of these separate subprogram issues before the program is loaded prior to execution.

Separate data definitions. An alternative model is to group together all operations that ma a given data object. For example, a subprogram might consist of all operations that address a spe data format within the program: operations to create the data record, operations to print the d record, and operations to compute with the data record. This is the general approach of the mechanism in languages like Java, C++, and Smalltalk.

Nested subprogram definitions. Nested subprogram definition was an important concept for bu modular programs during the early days of ALGOL, FORTRAN and Pascal, but the conce disappearing with the rise of object-oriented languages such as C++ and Java. Pascal illustra nested program structure in which subprogram definitions appear as declarations within the m program and may contain other subprogram definitions nested within their definitions to any d These nested subprogram definitions serve to provide a nonlocal referencing environme subprograms that is defined at compile time and that allows static type checking and compila efficient executable code for subprograms containing nonlocal references.

Separate interface definitions. The structure of FORTRAN permits the easy compilation of sepa subprograms, but it has the disadvantage that data used across different subprograms may different definitions that the compiler will not be able to detect at compile time. However, Pa allows the compiler to have access to all such definitions to aid in finding errors. The disadvanta that the entire program, even if it is many thousands of statements long, must be recompiled time a single statement needs to be changed. C, ML, and Ada use aspects of both of these techn to improve compilation behavior.

In these languages, a program implementation consists of several subprograms that are to interact together. All such components, called *modules*, are linked together, as in FORTRAN create an executable program, but only any changed components need be recompiled each However, data passed among the procedures in a component must have common declarations Pascal, permitting efficient checking by the compiler. However, to pass information betwe separately compiled components, additional data are needed. This is handled by a *program spec* component. In C the approach is to include certain operating system file operations into the la by allowing the program to include files that contain these interface definitions. The C ".h" fil the specification components and the source program ".c" files form the implementation comp In Ada, the approach was to build such features directly into the language. Programs are de components called *packages*, which contain either the specification of the interface defini the source program implementations (in the package *body*) that will use these definitions.

Data descriptions separated from executable statements. COBOL contains an early fo component structure. In a COBOL program, the data declarations and the executable statem all subprograms are divided into separate program *data divisions* and *procedure divisions*. A *environment division* consists of declarations concerning the external operating environme procedure division of a program is organized into subunits corresponding to subprogram bo all data are global to all subprograms, and there is nothing corresponding to the usual local d a subprogram. The advantage of the centralized data division containing all data declaration it enforces the logical independence of the data formats and the algorithms in the procedure d

The popular UNIX program *make* is used to determine which ".c" and ".h" files have been altered to changed components of a system.

building small throw-away programs that need to be executed a few times and then Language Summary 3.1 summarizes the BASIC language. Since the 1980s, however, successive versions of BASIC have become more complex, often reaching the structure of languages like Pascal in power, syntax, and complexity.

3.2 STAGES IN TRANSLATION

The process of translation of a program from its original syntax into executable form is central in every programming language implementation. The translation may be quite simple, as in the case of Perl, Prolog, or LISP programs, but more often the process can be quite complex. Most languages could be implemented with only trivial translation if one were willing to write a software interpreter and if one were willing to accept slow execution speeds. However, efficient execution is such a desirable goal that major efforts are made to translate programs into efficiently executable structures. The translation process becomes progressively more complex as the executable program form becomes further removed in structure from the original program. At the extreme, an optimizing compiler for a complex language like Ada may radically alter program structures to obtain more efficient execution.

Logically, we may divide translation into two major parts: the analysis of the input source program and the synthesis of the executable object program. In most translators, these logical stages are not clearly separate but instead are mixed so that analysis and synthesis alternate—often on a statement-by-statement basis. Figure 3.2 illustrates the structure of a typical compiler.

Translators are crudely grouped according to the number of passes they make over the source program. A simple compiler typically uses two passes. The first analysis pass decomposes the program into its constituent components and derives information, such as variable name usage, from the program. The second pass typically generates an object program from this collected information.

If compilation speed is important (such as in an educational compiler), a one-pass strategy may be employed. In this case, as the program is analyzed, it is immediately converted into object code. Pascal was designed so that a one-pass compiler could be developed for the language. However, if execution speed is paramount, a three- (or more) pass compiler may be developed. The first pass analyzes the source program, the second pass rewrites the source program into a more efficient form using various well-defined optimization algorithms, and the third pass generates the object code.

As our knowledge of compiler technology has improved, the relationship between number of passes and compiler speed is no longer clear. What is more important is the complexity of the language rather than the number of passes needed to analyze the source program.

3.2.1 Analysis of the Source Program

To a translator, the source program appears initially as one long undifferentiated sequence of symbols composed of thousands or tens of thousands of characters. Of course, a programmer seeing such program almost instinctively structures it into subprograms, statements, declarations and so forth. To the translator, none of this is apparent. An analysis of the program's structure must be laboriously built up character by character during translation.

Lexical analysis (scanning). The initial phase of any translation is to group this sequence of characters into its elementary constituents: identifiers, delimiters, operator symbols, numbers, keywords, noise words, blanks, comments, and so on. This phase is termed *lexical analysis*, and the basic program units that result from lexical analysis are termed *lexical items* (or *tokens*). Typically the lexical analyzer (or scanner) is the input routine for the translator, reading successive lines of input program, breaking them down into individual lexical items, called lexemes, and feeding these lexemes to the later stages of the translator to be used in the higher levels of analysis. The lexical analyzer must identify the type

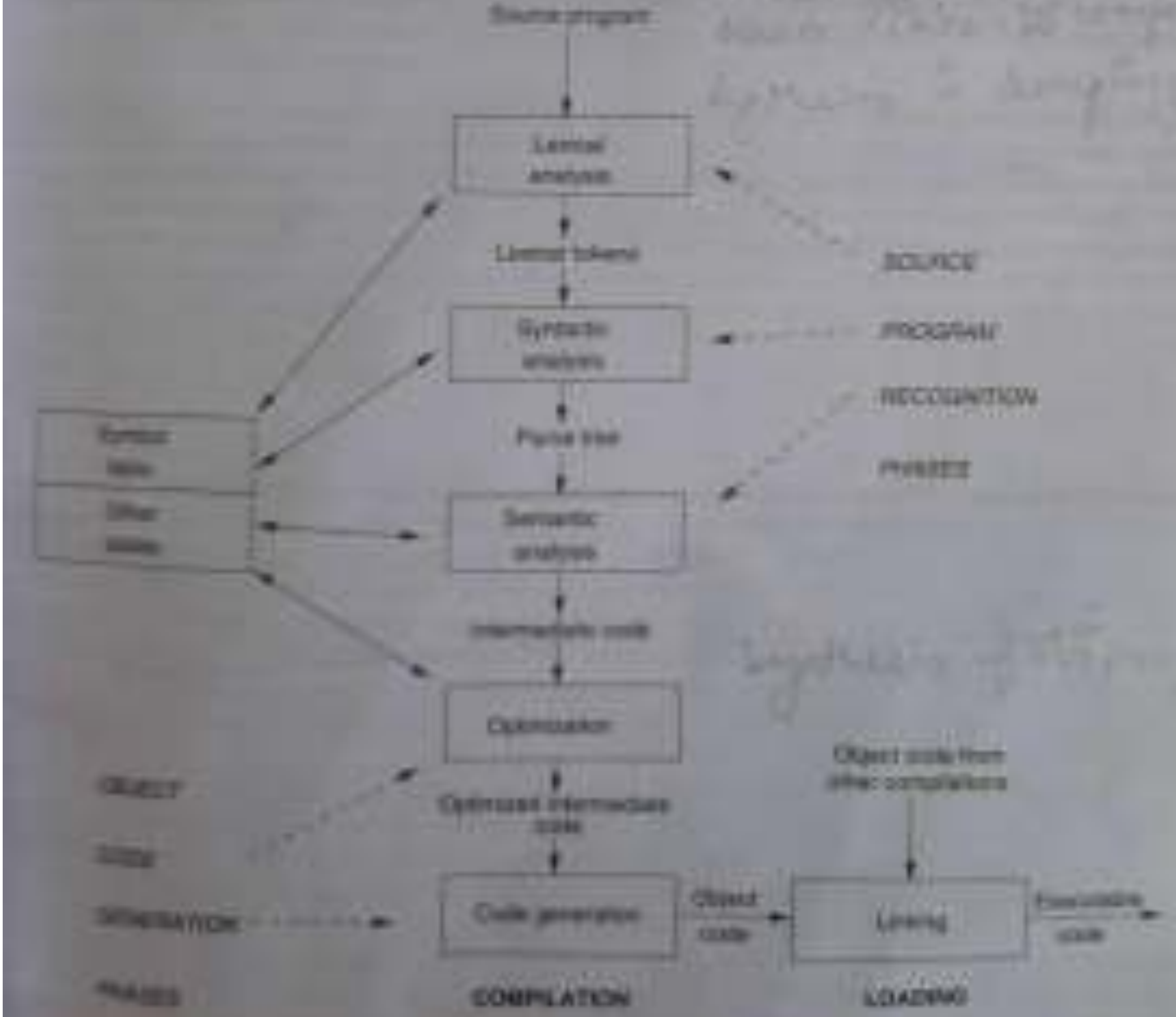


Figure 3.2. Structure of a compiler

of each lexeme (number, identifier, delimiter, operator, etc.) and attach a type tag. In addition, conversion to an internal representation is often made for items such as numbers (converted to internal binary fixed- or floating-point form) and identifiers (stored in a symbol table and the address of the symbol table entry used in place of the character string). The formal model used to design lexical analyzers is the *finite-state automaton*, which is briefly described in Section 3.3.2.

Although lexical analysis is simple in concept, this phase of translation often requires a larger share of translation time than any other. This fact is in part due simply to the necessity to scan and analyze the source program character by character. It is also true that in practice it is sometimes difficult to determine where the boundaries between lexical items lie without rather complex context-dependent algorithms. For example, the two FORTRAN statements

DO 10 I = 1,5 and DO 10 I = 1.5

have entirely different lexical structures. The first is a DO statement and the second is an assignment, but this fact cannot be discovered until either the "," or "." character is read, because blanks are ignored in FORTRAN.

Syntactic analysis (parsing). The second stage in translation is *syntactic analysis* or *parsing*. Here the larger program structures are identified (statements, declarations, expressions, etc.) using the lexical items produced by the lexical analyzer. Syntactic analysis usually alternates with semantic analysis. First, the syntactic analyzer identifies a sequence of lexical items forming a syntactic unit such as an expression, statement, subprogram call, or declaration. A semantic analyzer is then called to process this unit. Commonly, the syntactic and semantic analyzers communicate using a stack. The syntactic analyzer enters in the stack the various elements of the syntactic unit found, and these are retrieved and processed by the semantic analyzer. Much research has centered on discovery of efficient syntactic-analysis techniques, particularly techniques based on the use of formal grammars (as described in Section 3.3.1).

Semantic analysis. Semantic analysis is perhaps the central phase of translation. Here the syntactic structures recognized by the syntactic analyzer are processed, and the structure of the executable object code begins to take shape. Semantic analysis is thus the bridge between the analysis and synthesis parts of translation. A number of other important subsidiary functions also occur in this stage, including symbol table maintenance, most error detection, expansion of macros, and execution of compile-time statements. The semantic analyzer may actually produce the executable object code in simple translations, but more commonly the output from this stage is some internal form of the final executable program, which is then manipulated by the optimization stage of the translator before executable code is actually generated.

The semantic analyzer is ordinarily split into a set of smaller semantic analyzers, each of which handles one particular type of program construct. The semantic analyzers interact among themselves through information stored in various data structures, particularly in the central symbol table. For example, a semantic analyzer that processes type declarations for simple variables may often do little more than enter the declared types into the symbol table. A later semantic analyzer that processes arithmetic expressions may then use the declared types to generate the appropriate type-specific arithmetic operations for the object code. The exact functions of the semantic analyzers vary greatly depending on the language and logical organization of the translator. Some of the most common functions may be described as follows:

1. *Symbol-table maintenance.* A *symbol table* is one of the central data structures in every translator. The symbol table contains an entry for each different identifier encountered in the source program. The lexical analyzer makes the initial entries as it scans the input program. The symbol-table entry contains more than just the identifier. It contains additional data concerning the attributes of that identifier: its type (simple variable, array name, subprogram name, formal parameter, etc.), type of values (integer, real, etc.), referencing environment, and whatever other information is available from the input program through declarations and usage. The semantic analyzers enter this information into the symbol table as they process declarations, subprogram headers, and program statements. Other parts of the translator use this information to construct efficient executable code.

   The symbol table in translators for compiled languages is usually discarded at the end of translation. However, it may be retained during execution (e.g., in languages that allow new identifiers to be created at run time or as an aid in debugging). ML, Prolog, and LISP implementations all utilize a symbol table initially *created* during translation as a central run-time system-defined data structure. *dbx* is a popular UNIX program that uses a run-time symbol table to debug C programs.

2. *Insertion of implicit information.* Often in the source program, information is implicit and must be made explicit in the lower level object program. Most of this implicit information goes under the general heading of *default conventions*, which are interpretations to be provided when the programmer gives no explicit specification. For example, a FORTRAN variable that is used but not declared is automatically provided with a type declaration depending on the initial letter of its name.

3. *Error detection.* The syntactic and semantic analyzers must be prepared to handle incorrect as well as correct programs. At any point, the lexical analyzer may send to the syntactic analyzer a lexical item that does not fit in the surrounding context (e.g., a statement delimiter in the middle of an expression, a declaration in the middle of a sequence of statements, an operator symbol where an identifier is expected). The error may be more subtle, such as a real variable where an integer variable is required or a subscripted variable reference with three subscripts when the array was declared to have only two dimensions. At each step in translation, a multitude of such errors might occur. The semantic analyzer must not only recognize such errors when they occur and produce an appropriate error message, but must also, in all but the most drastic cases, determine the appropriate way to continue with syntactic analysis of the remainder of the program.

4. *Macro processing and compile-time operations.* Not all languages include macro features or provision for compile-time operations. Where these are present, however, processing is usually handled during semantic analysis.

   A *macro*, in its simplest form, is a piece of program text that has been separately defined and that is to be inserted into the program during translation whenever an appropriate macro call is encountered in the source program. Thus, a macro is much like a subprogram except that rather than being separately translated and called at run time (i.e., the binding of the subprogram name with its semantics occurs at run time), its body is simply substituted for each call during program translation (i.e., the binding occurs at translation time). Macros may be just simple strings to be substituted (e.g., substitution of 3.1416 for PI whenever the latter is referenced). More commonly, they look much like subprograms with parameters that must be processed before the substitution for the macro call is made.

Where macros are allowed, the semantic analyzers must identify the macro call within the source program and set up the appropriate substitution of the macro body for the call. Often this task involves interrupting the lexical and syntactic analyzers and setting them to work analyzing the string representing the macro body before proceeding with the remainder of the source string. Alternatively, the macro body may have already been partially translated so that the semantic analyzer can process it directly, inserting the appropriate object code and making the appropriate table entries before continuing with analysis of the source program.

A compile-time operation is an operation to be performed during translation to control the translation of the source program. C provides a number of such operations. The C "#define" allows for constants or expressions to be evaluated before the program is compiled. The "#ifdef" (if-defined) construct allows for alternative sequences of code to be compiled depending on the presence or absence of certain variables. These switches allow the programmer to alter the sequence of statements that are compiled. For example, a common source file can be used to compile alternative versions of a program:

```
#define pc                  /* Set to PC or UNIX version of program */
...
ProgramWrite(...)
    #ifdef pc               /* If defined then PC code needed */
    ...                     /* Do PC version code */
    ...                     /* e.g., write Microsoft Windows output */
    #else
    ...                     /* Do UNIX version code */
    ...                     /* e.g., write Motif X Windows output */
    #endif
```

2.2.2 Synthesis of the Object Program

The final stages of translation are concerned with the construction of the executable program from the outputs produced by the semantic analyzer. This phase involves code generation necessarily and may also include optimization of the generated program. If subprograms are translated separately, or if library subprograms are used, a final linking and loading stage is needed to produce the complete program ready for execution.

Optimization. The semantic analyzer ordinarily produces as output the executable translated program represented in some intermediate code, an internal representation such as a string of operators and operands, or a table of operator-operand sequences. From this internal representation, the code generator may generate the properly formatted output object code. Before code generation, however, there is usually some optimization of the program in the internal representation. Typically, the semantic analyzer generates the internal program form piecemeal as each segment of input program is analyzed. The semantic analyzer generally does not have to worry about the surrounding code that has already been generated. In doing this piecemeal output, however, extremely poor code may be produced (e.g., a register may be stored at the end of one generated segment and immediately reloaded from the same location at the beginning of the next segment). For example,

$$A = B + C + D$$

Where macros are allowed, the semantic analyzers must identify the macro call withi source program and set up the appropriate substitution of the macro body for the call. O this task involves interrupting the lexical and syntactic analyzers and setting them to analyzing the string representing the macro body before proceeding with the remain the source string. Alternatively, the macro body may have already been partially tran so that the semantic analyzer can process it directly, inserting the appropriate obje and making the appropriate table entries before continuing with analysis of the s program.

A compile-time operation is an operation to be performed during translation to cont translation of the source program. C provides a number of such operations. The C "#d allows for constants or expressions to be evaluated before the program is compile "#ifdef" (if-defined) construct allows for alternative sequences of code to be co depending on the presence or absence of certain variables. These switches allo programmer to alter the sequence of statements that are compiled. For example, a co source file can be used to compile alternative versions of a program:

```
#define pc                    /* Set to PC or UNIX version of program */
    ...
ProgramWrite(...)
    #ifdef pc                 /* If defined then PC code needed */
    ...                       /* Do PC version code */
                              /* e.g., write Microsoft Windows output */
    #else
    ...                       /* Do UNIX version code */
                              /* e.g., write Motif X Windows output */
    #endif
```

3.2.2 Synthesis of the Object Program

The final stages of translation are concerned with the construction of the executable progran the outputs produced by the semantic analyzer. This phase involves code generation necessari may also include optimization of the generated program. If subprograms are translated separat if library subprograms are used, a final linking and loading stage is needed to produce the com program ready for execution.

Optimization. The semantic analyzer ordinarily produces as output the executable translated pro represented in some intermediate code, an internal representation such as a string of operator operands or a table of operator-operand sequences. From this internal representation, the generators may generate the properly formatted output object code. Before code genera however, there is usually some optimization of the program in the internal representation. Typ the semantic analyzer generates the internal program form piecemeal as each segment of input pro is analyzed. The semantic analyzer generally does not have to worry about the surrounding code has already been generated. In doing this piecemeal output, however, extremely poor code ma produced (e.g., a register may be stored at the end of one generated segment and immedia reloaded from the same location at the beginning of the next segment). For example, the statem

A = B + C + D

may generate the intermediate code

(a) Temp1 = B + C
(b) Temp2 = Temp1 + D
(c) A = Temp2

which may generate the straightforward, but inefficient, code

1. Load register with B (from (a))
2. Add C to register
3. Store register in Temp1
4. Load register with Temp1 (from (b))
5. Add D to register
6. Store register in Temp2
7. Load register with Temp2 (from (c))
8. Store register in A.

Instructions 3, 4, 6, and 7 are redundant because all the data can be kept in the register before storing the result in A. Often it is desirable to allow the generation of poor code sequences by the semantic analyzers and then during optimization replace these sequences by better ones that avoid obvious inefficiencies.

Many compilers go far beyond this sort of simple optimization and analyze the program for other improvements that can be made (e.g., computing common subexpressions only once, removing constant operations from loops, optimizing the use of registers, and optimizing the calculation of array accessing formulas). Much research has been done on program optimization, and many sophisticated techniques are known (see the references at the end of this chapter).

Code generation. After the translated program in the internal representation has been optimized, it must be formed into the assembly language statements, machine code, or other object program form that is to be the output of the translation. This process involves formatting the output properly from the information contained in the internal program representation. The output code may be directly executable, or there may be other translation steps to follow (e.g., assembly or linking and loading).

Linking and loading. In the optional final stage of translation, the pieces of code resulting from separate translations of subprograms are coalesced into the final executable program. The output of the preceding translation phases typically consists of executable programs in almost final form, except where the programs reference external data or other subprograms. These incomplete locations in the code are specified in attached *loader tables* produced by the translator. The *linking loader* (or *link editor*) loads the various segments of translated code into memory and then uses the attached loader tables to link them together properly by filling in data and subprogram addresses in the code as needed. The result is the final executable program ready to be run.

Bootstrapping

Often the translator for a new language is written in that language. For example, the initial Pascal compiler was written in Pascal and designed to execute on a P-code virtual machine. One problem

Elementary Data Types

Every program is a function that transforms a given set of inputs into an expected set of outputs. The set of inputs and the expected set of outputs are known to the programmer. The program must result in exactly the set of expected outputs. Anything more or less is not acceptable.

The set of inputs and outputs are commonly called data. Representing this data in binary form is a cumbersome exercise. This was done during the days of machine language and assembly language programming.

The input to the program is coming from the real world. Representing it in binary form makes the programmer lose the connection the data has with the real world. This makes the perception of relationships among data items very difficult. Programming relies on the correct perception of the relationships among various data items. Thus working with data needs the programmer to be closer to the real world and not to the binary world of the computers.

Language designers began to explore the nature of real world data. Real world data or entities take many forms. **Type theory** deals with classifying entities into sets called types. A **type** is a category of entities. Programming languages support some elementary data types. These data types are close to the real world entities and hence make it easy for the programmer to focus on the transformation to be done to produce the desired set of output. The compiler takes care of the conversion of the data type into machine recognizable form and vice-versa. This simple beginning of the data type concept in programming languages has paved way for higher forms of abstraction and notions of type checking.

A type system is a **tractable syntactic method** for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. A type system can be regarded as calculating a kind of **static approximation** to the run-time behaviors of the terms in a program.

Types are useful in:

1. Error detection: Early detection of common programming errors.
2. Safety: Well-typed programs do not go wrong.
3. Design: Types provide a language and discipline for design of data structures and program interfaces.
4. Abstraction: Types enforce language and programmer abstractions.
5. Verification: Properties and invariants expressed in types are verified by the compiler. This is a proven guarantee of correctness.

6. *Software evolution.* Support for orderly evolution of software by facilitating the tracing the consequences of changes.
7. *Documentation.* Types express programmer assumptions and are verified by compiler.

All programs specify a set of operations that are to be applied to certain data in a certain sequ Basic differences among languages exist in the types of data allowed, in the types of operations a able, and in the mechanisms provided for controlling the sequence in which the operations are appl to the data. These three areas—data, operations, and control—form the basis for much of the discus and comparison of the languages in this book. This chapter considers the data, types, and operations are usually built into languages. Chapter 6 extends this concept to programmer-defined extensions these basic data types—a concept called abstraction.

5.1 PROPERTIES OF TYPES AND OBJECTS

We first investigate the properties that define data objects in a programming language. We t discuss typical types in most programming languages in terms of features available from the hardw of the actual computer (i.e., elementary data types). In chapter 6 we consider data that are ma software simulated (i.e., structured data types).

5.1.1 Data Objects, Variables, and Constants

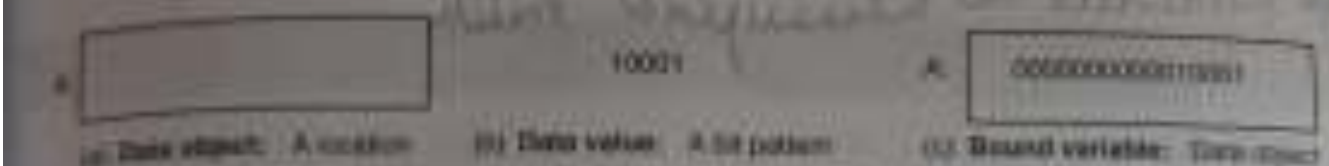
The data storage areas of an actual computer, such as the memory, registers, and external media, usu have a relatively simple structure as sequences of bits grouped into bytes or words. However, th storage of the virtual computer for a programming language tends to have a more complex orga zation, with arrays, stacks, numbers, character strings, and other forms of data existing at diffe points during execution of a program. It is convenient to use the term *data object* to refer to a run-ti grouping of one or more pieces of data in a virtual computer. During execution of a program, ma different data objects of different types exist. Moreover, in contrast to the relatively static organi of the underlying storage areas of an actual computer, these data objects and their interrelations change dynamically during program execution.

Some of the data objects that exist during program execution are *programmer defined* (i.e., th are variables, constants, arrays, files, etc.); the programmer explicitly creates and manipulates the declarations and statements in the program. Other data objects are *system defined* (i.e., they a data objects that the virtual computer sets up for housekeeping during program execution and a not directly accessible to the programmer, such as run-time storage stacks, subprogram activ records, file buffers, and free-space lists). System-defined data objects are ordinarily generated a matically as needed during program execution without explicit specification by the programme

A data object represents a *container for data values*—a place where data values may be st and later retrieved. A data object is characterized by a set of *attributes*, the most important of w is its *data type*. The attributes determine the number and type of values that the data object m contain and also determine the logical organization of those values.

A *data value* might be a single number, character, or possibly a pointer to another data obje A data value is ordinarily represented by a particular pattern of bits in the storage of a compute For now we consider two values to be the same if the patterns of bits representing them in stor are identical. However, this simple definition is insufficient for more complex data items. It is easy confuse data objects and data values and, in many languages, the distinction is not clearly made. Th [illegible] data values & class of elements of data [illegible]

[illegible]



(a) **Data object:** A location in computer memory with the name A.

(b) **Data value:** A bit pattern used by the translator whenever the number 17 is used in a program.

(c) **Bound variable:** Data object bound to data value 17.

Figure 5.1. A simple variable data object with value 17.

distinction is perhaps most easily seen by noting the difference in implementation. A data object is usually represented as storage in the computer memory. A data value is represented by a pattern of bits. To say that a data object *A* contains the value *B* means that the block of storage representing *A* is set to contain the particular bit pattern representing *B*, as shown in Figure 5.1.

If we observe the execution of a program, some data objects exist at the beginning of execution and others are created dynamically during execution. Some data objects are destroyed during execution; others persist until the program terminates. Thus, each object has a lifetime during which it may be used to store data values. A data object is *elementary* if it contains a data value that is always manipulated as a unit. It is a *data structure* if it is an aggregate of other data objects.

A data object participates in various bindings during its lifetime. Although the attributes of a data object are invariant during its lifetime, the bindings may change dynamically. The most important attributes and bindings are as follows:

1. *Type.* This associates the data object with the set of data values that the object may take.
2. *Location.* The binding to a storage location in memory where the data object is represented ordinarily is not directly modifiable by the programmer but is set up and may be changed by the storage management routines of the virtual computer, as discussed in chapter 10.
3. *Value.* This binding is usually the result of an assignment operation.
4. *Name.* The binding to one or more names by which the object may be referenced during program execution is usually set up by declarations and modified by subprogram calls and returns, as discussed in chapter 9.
5. *Component.* The binding of a data object to one or more data objects of which it is a component is often represented by a pointer value, and it may be modified by a change in the pointer, as discussed in Section 5.3.2.

Variables and Constants

A data object that is defined and named by the programmer explicitly in a program is termed a *variable*. A *simple variable* is an elementary data object with a name. We usually think of the value (or values) of a variable as being modifiable by assignment operations (i.e., the binding of data object to value may change during its lifetime). If there is no difference between upper and lower case letters in a name (e.g., *MYVARIABLE* and *myvariable* refer to the same object), the names are said to be *case insensitive*. If they are different objects, the names are *case sensitive*.

A *constant* is a data object with a name that is bound to a value (or values) permanently during its lifetime. A *literal* (or *literal constant*) is a constant whose name is just the written representation

EXAMPLE 5.1. C Simple Variables

A C subprogram may include the declaration

```
int N;
```

which declares a simple data object N of type integer. Subsequently in the subprogram, the assignment

```
N = 27;
```

may be used to assign the data value 27 to N. We would describe the situation somewhat mo completely as follows:

1. The declaration specifies an elementary data object of type integer.
2. This data object is to be created on entry to the subprogram and destroyed on exit f its lifetime is the duration of execution of the subprogram.
3. During its lifetime, the data object is to be bound to the name "N," through which it may referenced, as happens in the assignment statement above. Other names may be bou to the data object if it is passed as an argument to another subprogram.
4. No value is bound to the data object initially, but the assignment statement binds the va 27 to it temporarily until some later assignment to *N* changes the binding.
5. Hidden from the programmer are other bindings made by the virtual computer: the d object N is made a component of an activation record, a data object that contains all t local data for the subprogram, and this activation record is allocated storage in a run stack (another hidden data object) at the time that execution of the subprogram begi When the subprogram terminates, this storage is freed for reuse, and the binding of da object to storage location is destroyed (discussed in greater detail in chapter 6).

of its value (e.g., "21" is the written decimal representation of the literal constant that is a data obj with value 21). A *programmer-defined constant* or a *manifest constant* is a constant whose nam chosen by the programmer in a definition of the data object.

Because the value of a constant is bound to its name permanently during its lifetime, that val is also known to the translator. Therefore, if a programmer writes in C #define *MAX* 30, that in mation is known at translation time. The C compiler may make use of that information knowing th the value may not be altered. (For example, the assignment statement *MAX* = 4 would be an because *MAX* is always the constant 30; it makes as much sense as writing the assignment statem 30 = 4.) Sometimes the compiler can use information about constant values to avoid generating c for a statement or expression. For example, in the **if** statement

if (MAX < 2) {...}

the translator already has the data values for the constants *MAX* and 2, can compute that it is fa that 30 is less than 2, and hence can ignore completely any code for the **if** statement.

Persistence. Most programs today are still developed using the batch processing mod (Section 1.2.2). That is, the programmer assumes the following sequence of events

EXAMPLE 5.2. C Variables, Constants, and Literals

A C subprogram may include the declarations:

```
const int MAX = 30;
int N;
```

We may then write the assignments:

```
N = 27;
N = N + MAX;
```

N is a simple variable; MAX, 27, and 30 are constants. N, MAX, "27," and "30" are names for data objects of type *integer*. The constant declaration specifies that the data object named MAX is to be bound permanently (for the duration of execution of the subprogram) to the value 30. The constant MAX is a *programmer-defined constant* because the programmer explicitly defines the name for the value 30. The name "27," on the other hand, is a *literal* that names a data object containing the value 27. Such literals are defined as part of the language definition itself. The important but confusing distinction here is between the value 27, which is an integer represented as a sequence of bits in storage during program execution, and the name "27," which is a sequence of two characters "2" and "7" that represents the same number in decimal form as it is written in a program. C both has constant declarations, as in this example, and macro definitions such as #define MAX 30, which is a compile-time operation that causes all references to MAX in the program to be changed to the constant 30.

Note that in this example, the constant 30 has two names, the programmer-defined name "MAX" and the literal name "30," both of which may be used to refer to a data object containing the value 30 in the program.

One should also realize that

```
#define MAX 30
```

is a command, which the translator uses to equate MAX with the value 30, whereas the *const* attribute in C is a translator directive stating that variable MAX will always contain the value 30.

1. The program is loaded into memory.
2. Appropriate external data (e.g., tapes, disks) are made available to the program.
3. The relevant input data are read into variables in the program, the variables are manipulated, and the result data are written back to its external format.
4. The program terminates.

The lifetime of the variables in the program are determined by the execution time of the program; however, the lifetime of the data often extends beyond that single execution. We say that the data are *persistent* and continue to exist between executions of the program.

Many applications today do not easily fit within this model. Consider an airline reservation system. To reserve a seat, you call a travel agent who interrogates the reservation system and invokes programs that check on schedules, prices, and destinations. The data and the programs coexist essentially indefinitely. In this case, having a language that represented persistent data would allow designing such transaction-based systems more efficiently. With such a language, variables woul declared whose lifetime extended beyond the execution of the program. Programming would be simp because there would be no need to specify the form of data in an external file before manipula it. The language translator would already know where the data were stored and their form. Section 11.4.1, we discuss current research on the development of persistent languages. How until such languages become widely used, in Section 5.3.3 we discuss the use of files for transferr persistent data into local program variables.

5.1.2 Data Types

A *data type* is a class of data objects together with a set of operations for creating and manipula them. Although a program deals with particular *data objects* such as an array *A*, the integer varia *X*, or the file *F*, a programming language necessarily deals more commonly with *data types* such as t class of arrays, integers, or files and the operations provided for manipulating arrays, integers, or fi

Every language has a set of *primitive* data types that are built into the language. In additio language may provide facilities to allow the programmer to define new data types. One of the ma differences between older languages such as FORTRAN and COBOL and newer languages such Java and Ada lies in the area of programmer-defined data types—a subject taken up in Section The newest trend is to allow for types to be manipulated by the programming language. This major feature added to ML and is one of the features present in the object-oriented programm models discussed in chapter 7.

The basic elements of a *specification* of a data type are as follows:

1. The *attributes* that distinguish data objects of that type,
2. The *values* that data objects of that type may have, and
3. The *operations* that define the possible manipulations of data objects of that type.

For example, in considering the specification of an array data type, the attributes might in the number of dimensions, the subscript range for each dimension, and the data type of the compon the values would be the sets of numbers that form valid values for array components; and operations would include subscripting to select individual array components and possibly operations to create arrays, change their shape, access attributes such as upper and lower bound subscripts, and perform arithmetic on pairs of arrays.

The following are basic elements of the *implementation* of a data type:

1. The *storage representation* that is used to represent the data objects of the data type in storage of the computer during program execution.

2. The manner in which the operations defined for the data type are represented in terms particular *algorithms* or *procedures* that manipulate the chosen storage representation of the d objects. The implementation of a data type defines the simulation of those parts of the vi computer in terms of the more primitive constructs provided by the underlying layer of vi computer, which may be directly the hardware computer or a hardware/software computer defined by an operating system or microcode.

The last concern connected with a data type lies in its *syntactic representation*. Both specificati and implementation are largely independent of the particular syntactic forms used in the langua

Attributes of data objects are often represented syntactically by declarations or type definitions. Values may be represented as literals or defined constants. Operations may be invoked by using special symbols, built-in procedures, or functions such as *sin* or *read*, or implicitly through combinations of other language elements. The particular syntactic representation makes little difference, but the information present in the program syntax provides information to the language translator that may be vital in determining the binding time of various attributes, and thus in allowing the translator to set up efficient storage representations or perform checking for type errors.

Specification of Elementary Data Types

An elementary data object contains a single data value. A class of such data objects over which various operations are defined is termed an *elementary data type*. Although each programming language tends to have a somewhat different set of elementary data types, the types *integer*, *real*, *character*, *Boolean*, *enumeration*, and *pointer* are often included, although the exact specification may differ significantly between two languages. For example, although most languages include Boolean data, it is treated quite differently in Java and C++.

Attributes. Basic attributes of any data object, such as data type and name, are usually invariant during its lifetime. Some of the attributes may be stored in a descriptor (also called a *dope vector*) as part of the data object during program execution; others may be used only to determine the storage representation of the data object and may not appear explicitly during execution. Note that the value of an attribute of a data object is different from the *value that the data object contains*. The value contained may change during the data object's lifetime and is always represented explicitly during program execution.

Values. The type of a data object determines the set of possible values that it may contain. For example, the integer data type determines a set of integer values that may serve as the values for data objects of this type. C defines the following four classes of integer types: *int*, *short*, *long*, and *char*. Because most hardware implements multiple precision integer arithmetic (e.g., 16-bit and 32-bit integers or 32-bit and 64-bit integers), C permits the programmer to choose among these hardware-defined implementations. *Short* uses the shortest value of the integer word length, *long* uses the longest value implemented by the hardware, and *int* uses the most efficient value that the hardware implements. This may be the same as *short*, the same as *long*, or some value intermediate to those. It is interesting to note that in C, characters are stored as 8-bit integers in the type *char*, which is a subtype of integer.

The set of values defined by an elementary data type is usually an *ordered set* with a least value and a greatest value; for any pair of distinct values, one is greater than the other. For example, for an integer data type, there is usually a greatest integer corresponding to the greatest integer that can be conveniently represented in memory, a least integer, and the integers between arranged in their usual numerical ordering.

Operations. The set of operations defined for a data type determine how data objects of that type may be manipulated. The operations may be *primitive operations*, which means they are specified as part of the language definition, or they may be *programmer-defined operations*, in the form of subprograms or method declarations as part of class definitions. This chapter emphasizes primitive operations; programmer-defined operations are considered in greater detail in subsequent chapters.

EXAMPLE 5.1. C Signatures of operations

(a) Integer addition is an operation that takes two integer data objects as arguments and produces an integer data object as its result (a data object usually containing the sum of the values of its two arguments). Thus its specification is:

+: *integer* × *integer* → *integer*

(b) The operator "=" that tests for equality of the values of two integer data objects and produces a data object containing a Boolean (true or false) result is specified:

=: *integer* × *integer* → Boolean

(c) A square-root operation, SQRT, on real number data objects is specified:

SQRT: real → real

An operation is a mathematical *function*; for a given input *argument* (or arguments), it has a well-defined and uniquely determined *result*. Each operation has a *domain* (the set of possible input arguments on which it is defined) and a *range* (the set of possible results that it may produce). The action of the operation defines the results produced for any given set of arguments.

An *algorithm* that specifies how to compute the results for any given set of arguments is a common method for specifying the action of an operation, but other specifications are possible. For example, to specify the action of a multiplication operation, you might give a multiplication table that simply lists the result of multiplying any two pairs of numbers, rather than an algorithm for multiplying any two numbers.

To specify the signature of an operation, the number, order, and data types of the arguments in the domain of an operation are given as well as the order and data type of the resulting range. It is convenient to use the usual mathematical notation for this specification:

op name: *arg type* × *arg type* × ··· *arg type* → *result type*

In C, we call this the function *prototype*.

An operation that has two arguments and produces a single result is termed a *binary* (or *dyadic*) operation. If it has one argument and one result, it is a *unary* (or *monadic*) operation. The number of arguments for an operation is often called the *arity* of the operation. Most of the primitive operations in programming languages are binary or unary operations.

A precise specification of the action of an operation ordinarily requires more information than just its signature. In particular, the storage representation of argument types usually determines how arguments of those types may be manipulated. For example, an algorithm for multiplication of two numbers, where the numbers are represented in binary notation, is different from a multiplication algorithm for decimal numbers. Thus in the specification of an operation, an informal description of the action is usually given. A precise specification of the action then is part of the implementation of the operation after the storage representations for the arguments have been determined.

It is sometimes difficult to determine a precise specification of an operation as a mathematical function. There are four main factors that combine to obscure the definition of many programming language operations:

1. *Operations that are undefined for certain inputs.* An operation that is defined over some domain may in fact be undefined for certain inputs in the domain (e.g., the square-root function on the negative integer domain). The exact domain on which an operation is undefined may be extremely difficult to specify, as, for example, the sets of numbers that cause underflow or overflow in arithmetic operations.

2. *Implicit arguments.* An operation in a program ordinarily is invoked with a set of explicit arguments. However, the operation may access other implicit arguments through the use of global variables or other nonlocal identifier references. Complete determination of all the data that may affect the result of an operation is often obscured by such implicit arguments.

3. *Side effects (implicit results).* An operation may return an explicit result, as in the sum returned as the result of an addition, but it may also modify the values stored in other data objects, both programmer- and system-defined. Such implicit results are termed *side effects*. A function may modify its input arguments as well as return a value. Side effects are a basic part of many operations, particularly those that modify data structures. Their presence makes exact specification of the range of an operation difficult.

4. *Self-modification (history sensitivity).* An operation may modify its own internal structure, either local data that are retained between executions or its own code. The results produced by the operation for a particular set of arguments then depend not only on those arguments but on the entire history of preceding calls during the computation and the arguments given at each call. The operation is said to be *history sensitive* in its actions. A common example is the *random number generator* found as an operation in many languages. Typically this operation takes a constant argument and yet returns a different result each time it is executed. The operation not only returns its result but also modifies an internal *seed number* that affects its result on the next execution. Self-modification through changes in local data retained between calls is common; self-modification through change in the code of an operation is less common but possible in languages such as LISP.

Subtypes. When describing a new data type, we often want to say that it is similar to another data type. For example, C defines the types of *int*, *long*, *short*, and *char* as variations on integers. All behave similarly, and we would like some operations, such as + and ×, to be defined in an analogous manner. If a data type is part of a larger class, we say it is a *subtype* of the larger class, and the larger class is a *supertype* of this data type. For example, in Pascal, we may create subranges of integers as in

type SmallInteger = 1..20;

giving a subtype of integer with values limited to 1, 2, 3, ..., 20.

With a subtype, we assume that the operations available to the larger class of objects are also available to the smaller class. How can we determine this? In Pascal and Ada, subranges of types are explicitly part of the language. How do we extend this to other data types that are not primitive to the language? The important concept of *inheritance*, discussed in chapter 7, is a generalization of this subtype property. We would say that the C *char* type *inherits* the operations of integer type *int*.

Implementation of Elementary Data Types

The implementation of an elementary data type consists of a *storage representation* for data objects and values of that type, and a set of *algorithms* or *procedures* that define the operations of the type in terms of manipulations of the storage representation.

Storage representation. Storage for elementary data types is strongly influenced by the underly computer that will execute the program. For example, the storage representation for integer or re values is almost always the integer or floating-point binary representation for numbers used in t underlying hardware. The reason for this choice is simple: If the hardware storage representatio are used, then the basic operations on data of that type may be implemented using the hardware-p vided operations. If the hardware storage representations are not used, then the operations must software simulated, and the same operations will execute much less efficiently.

The attributes of elementary data objects are treated similarly:

1. For efficiency, many languages are designed for data attributes to be determined by t compiler. The attributes are not stored in the run-time storage representation. This is usual method in C, where efficiency of storage use and execution speed are primary goa
2. The attributes of a data object may be stored in a *descriptor* as part of the data object at time. This is the usual method in languages such as LISP and Prolog, where flexibility ra than efficiency is the primary goal. Because most hardware does not provide stor representations for descriptors directly, descriptors and operations on data objects w descriptors must be software simulated.

The representation of a data object is ordinarily independent of its location in memory. The stor representation is usually described in terms of the size of the block of memory required (the numb of memory words, bytes, or bits needed) and the layout of the attributes and data values within block. Usually the address of the first word or byte of such a block of memory is taken to repre the location of the data object.

Implementation of operations. Each operation defined for data objects of a given type may implemented in one of three main ways:

1. *Directly as a hardware operation.* For example, if integers are stored using the hardw representation for integers, then addition and subtraction are implemented using the arithm operations built into the hardware.

2. *As a procedure or function subprogram.* For example, a square-root operation is usu not provided directly as a hardware operation. It might be implemented as a square-root subprog that calculates the square root of its argument.

3. *As an inline code sequence.* An *inline code sequence* is also a software implementation of operation. However, instead of using a subprogram, the operations in the subprogram are copied i the program at the point where the subprogram would otherwise have been invoked. For example, absolute-value function on numbers defined by

abs(x) = **if** x < 0 **then** −x **else** x

is usually implemented as an inline code sequence:

(a) Fetch value of x from memory.

(b) If x > 0, skip the next instruction.

(c) Set x = −x.

(d) Store new value of x in memory.

Here, each line is implemented by a single hardware operation.

In writing a program, the programmer determines the name and type of each data object that is needed. Also the lifetime of each data object, during what part of program execution it is needed, as well as the operations to be applied to it, must be specified.

A declaration is a program statement that serves to communicate to the language translator information about the name and type of data objects needed during program execution. By its placement in the program (e.g., within a particular subprogram or class definition), a declaration may also serve to indicate the desired lifetimes of the data objects. For example, the C declaration

```
float A, B;
```

at the start of a subprogram indicates that two data objects of type *float* are needed during execution of the subprogram. The declaration also specifies the binding of the data objects to the names *A* and *B* during their lifetimes.

The previous C declaration is an *explicit declaration*. Many languages also provide *implicit* or *default* declarations, which are declarations that hold when no explicit declaration is given. For example, in a FORTRAN subprogram, a simple variable *INDEX* may be used without explicit declaration; by default, it is assumed by the FORTRAN compiler to be an integer variable because its name begins with one of the letters I–N. In Perl, simply assigning a value to a variable declares it:

```
$abc = 'a string';          # $abc is now a string variable
$abc = 7;                   # $abc is now an integer variable
```

A declaration may also specify the value of the data object if it is a constant or the initial value of the data object, if not. Other bindings for the data object may also be specified in the declaration: a name for the data object or the placement of the data object as a component of a larger data object. Sometimes implementation details such as binding to a particular storage location or to a particular specialized storage representation are also specified. For example, the COBOL designation of an integer variable as COMPUTATIONAL usually indicates that a binary rather than a character-string storage representation for the value of that data object is needed (to allow more efficient arithmetic operations to be used).

Declarations of Operations

The information required during translation is primarily the signature of each operation. No explicit declaration of argument types and result types for primitive operations that are built into a language is ordinarily required. Such operations may be invoked as needed in writing a program, and the argument and result types are determined implicitly by the language translator. However, argument and result types for programmer-defined operations must usually be made known to the language translator before the subprogram may be called. For example, in C, the head of a subprogram definition, its *prototype*, provides this information. Thus,

```
float Sub(int X, float Y)
```

declares Sub to have the signature

Sub: *int* × *float* → *float*

Purposes for Declarations

Declarations serve several important purposes:

1. *Choice of storage representations.* If a declaration provides information to the langu translator about the data type and attributes of a data object, then the translator can often determ the best storage representation for that data object.

2. *Storage management.* Information provided by declarations about the lifetimes of d objects often makes it possible to use more efficient storage management procedures du program execution. In C, for example, data objects declared at the beginning of a subprogram have the same lifetime (equal to the duration of execution of the subprogram) and thus may allocated storage as a single block on entry to the subprogram, with the entire block being freed exit. Other C data objects are created dynamically by use of a special function *malloc*. Because lifetimes of these data objects are not declared, they must be allocated storage individually.

3. *Polymorphic operations.* Most languages use special symbols such as + to designate one of several different operations depending on the data types of the arguments provided. *A* + in C, for example, means "perform integer addition" if *A* and *B* are of integer type or "perform addition" if *A* and *B* are of *float* type. Such an operation symbol is said to be *overloaded* becau does not designate *one* specific operation, but rather denotes a generic "add" operation that have several different *type-specific* forms for arguments of different types. In most languages, basic operation symbols such as +, *, and / are overloaded (i.e., they denote generic operations); other operation names uniquely identify a particular operation. However, Ada allows the progr mer to define overloaded subprogram names and add additional meanings to existing opera symbols. ML expands this concept with full *polymorphism*, where a function name may take variety of implementations depending on the types of arguments. When we discuss object-orien designs, polymorphism is a major feature that allows the programmer to extend the language new data types and operations.

Declarations usually allow the language translator to determine at compile time the parti operation designated by an overloaded operation symbol. For example, in C, the compiler de mines from the declarations of Variables *A* and *B* which of the two possible operations (integer tion or float addition) is designated by $A + B$. No run-time checking is required. In Smallta contrast, because there are no declarations of types for variables, the determination of which + o ation to perform must be made each time a + operation is encountered during program execu

4. *Type checking.* The most important purpose for declarations, from the programmer's point, is that they allow for *static* rather than *dynamic* type checking.

5.1.4 Type Checking and Type Conversion

Data storage representations that are built into the computer hardware usually include no type i mation, and the primitive operations on the data do no type checking. For example, a particular in the computer memory during execution of a program may contain the bit sequence 111001 ...0011. This bit sequence might represent an integer, a real number, a sequence of characters, instruction; there is no way to tell. The hardware primitive operation for integer addition c check whether its two arguments represent integers; they are simply bit sequences. Thus, at the hard level, conventional computers are particularly unreliable in detecting data type errors.

Type checking means checking that each operation executed by a program receives the p number of arguments of the proper data type. For example, before executing the assignment state

$$X = A + B * C$$

the compiler must determine for each operation—addition, multiplication, and assignment—that each receives two arguments of the proper data type. If + is defined only for integer or real arguments and A names a character data object, then there is an argument *type error*. Type checking may be done at run time (*dynamic type checking*) or at compile time (*static type checking*). A major advantage of using a high-level language in programming is that the language implementation can provide type checking for all (or almost all) operations, and thus the programmer is protected against this particularly insidious form of programming error.

Dynamic type checking is run-time type checking usually performed immediately before the execution of a particular operation. Dynamic type checking is usually implemented by storing a type tag in each data object that indicates the data type of the object. For example, an integer data object would contain both the integer value and an integer type tag. Each operation is then implemented to begin with a type-checking sequence in which the type tag of each argument is checked. The operation is performed only if the argument types are correct; otherwise an error is signaled. Each operation must also attach the appropriate type tags to its results so that subsequent operations can check them.

Some programming languages such as Perl and Prolog are designed to require dynamic type checking. In these languages, no declarations for variables are given and no default declaration of type is assumed (in contrast to the default typing structures of FORTRAN). The data types of variables such as *A* and *B* in the expression *A* + *B* may change during the course of program execution. In such circumstances, the types of *A* and *B* must be checked dynamically each time the addition is performed at run time. In languages without declarations, the variables are sometimes said to be *typeless* because they have no fixed type.

The major advantage of dynamic types is the flexibility in program design. No declarations are required, and the type of data object associated with a variable name may change as needed during program execution. The programmer is freed from most concerns about data types. However, dynamic type checking has several major disadvantages:

1. Programs are difficult to debug (i.e., to completely remove all argument type errors). Because dynamic type checking checks data types at the time of execution of an operation, operations on program execution paths that are not executed are never checked. During program testing, not all possible execution paths can be tested, in general. Any untested execution paths may still contain argument type errors.

2. Dynamic type checking requires that type information be kept during program execution. The extra storage required can be substantial.

3. Dynamic type checking must ordinarily be implemented in software because the underlying hardware seldom provides support. This reduces the speed for executing the operation.

Most languages attempt to eliminate or minimize dynamic type checking by performing type checking at compile time. *Static type checking* is performed during translation of a program. The needed information is usually provided in part by declarations that the programmer provides and in part by other language structures. The information required includes the following:

1. *For each operation, the number, order, and data types of its arguments and results* (i.e., its signature).

2. *For each variable, the type of data object named.* The type of the data object associated with a variable name must be invariant during program execution as well. However, in checking an

expression such as $A + B$, it can be assumed that the type of data object named by A is the same on each execution of the expression even if the expression is executed repeatedly with different bindings of A to particular data objects.

3. *The type of each constant data object.* The syntactic form of a literal usually indicates its type (e.g., 2 is an integer, 2.3 is a real number). Each defined constant must be matched with its definition to determine its type.

During the initial phases of program translation, the compiler (or other translator) collects information from declarations in the program into various tables, primarily a symbol table (see chapter that contains type information about variables and operations. After all the type information is collected, each operation invoked by the program is checked to determine whether the type of each argument is valid. Note that if the operation is a polymorphic one, as discussed earlier, then any of several argument types may be valid. If the argument types are valid, then the result types are determined and the compiler saves this information for checking later operations. Note that the polymorphic operation name may also be replaced by the name of the particular *type-specific operation* that uses arguments of the designated types.

Because static type checking includes all operations that appear in any program statement, all possible execution paths are checked, and further testing for type errors is not needed. Thus type tags on data objects at run time are not required, and no dynamic type checking is needed. The result is a substantial gain in efficiency of storage use and execution speed.

Concern for static type checking tends to affect many aspects of the language: declarations, data-control structures, and provisions for separate compilation of subprograms, to name a few. In most languages, static type checking is not possible for some language constructs in certain cases. These flaws in the type-checking structure may be treated in two ways:

1. *By dynamic type checking.* Often the storage cost of this option is high because type tags for data objects must be stored at run time, although the type tags are rarely checked.
2. *By leaving the operations unchecked.* Unchecked operations can cause serious and subtle program errors, as noted earlier, but sometimes they are accepted where the cost of dynamic checking is considered too great.

Strong typing. If we can detect all type errors statically in a program, we say that the language is *strongly typed*. In general, strong typing provides a level of security to our programs. We call a function f, with signature $f: S \to R$, *type safe* if execution of f cannot generate a value outside of R. For all operations that are type safe, we know statically that the results will be of the correct type and that no dynamic checking need be done. Obviously, if every operation is type safe, the language is strongly typed.

Few languages are truly strongly typed. For example, in C if X and Y are of type *short* (i.e., short integers), then $X + Y$ and $X * Y$ may have a result outside of the range allowable for short integers and cause a type error. Although true strong typing is difficult, if we restrict conversion between one type and another, we come close to strong typing. We discuss such conversion in the subsection that follows.

Type inference. ML has an interesting approach toward data types. Type declarations are not necessary if the interpretation is unambiguous. The language implementation will infer any missing type information from other declared types. The language has a relatively standard syntax for declaring arguments to functions, as in

```
fun area(length:int, width:int):int = length * width;
```

which declares the function *area* to return the area (as an integer) if given the integer sides of a rectangle. In this case, once the type of one of *length*, *width*, or *area* is determined, then the other two are also determined. Leaving out any two of these declarations still leaves the function with only one interpretation. Knowing that * can multiply together either two reals or two integers, ML interprets the following as equivalent to the prior example:

```
fun area(length, width):int = length * width;
fun area(length:int, width) = length * width;
fun area(length, width:int) = length * width;
```

However,

```
fun area(length, width) = length * width;
```

is invalid because it is now ambiguous as to the type of arguments. They could all be int or they could all be real.

Type Conversion and Coercion

If, during type checking, a mismatch occurs between the actual type of an argument and the expected type for that operation, then either

1. The type mismatch may be flagged as an error and an appropriate error action taken, or
2. A *coercion* (or *implicit type conversion*) may be applied to change the type of the actual argument to the correct type.

A *type conversion* is an operation with the signature

$$conversion\_op : type_1 \rightarrow type_2$$

That is, the conversion takes a data object of one type and produces the corresponding data object of a different type. Most languages provide type conversions in two ways:

1. As a set of *built-in functions* that the programmer may explicitly invoke to effect the conversion. For example, Pascal provides the function *round* that converts a real-number data object to an integer data object with a value equal to the rounded value of the real. In C, we cast an expression to coerce it to the correct type; (int) *X*, for *float X* converts the value of *X* to type integer.
2. As *coercions* invoked automatically in certain cases of type mismatch. For example, in Pascal, if the arguments for an arithmetic operation such as "+" are of mixed real and integer types, the integer data object is implicitly converted to type real before the addition is performed. Unlike in C++, Java permits implicit coercions if the operation is a widening. Thus an int value can be assigned to a float variable in Java, but in C++ an explicit cast to float must be given.

The basic principle driving coercions is not to lose information. Because every short integer (in C) can be represented as a long integer, no information is lost by automatically invoking a short int → long int conversion. Such coercions are called *widenings* or *promotions*. Similarly, because integers (in most languages) can be exactly represented as a real data object, small-valued integers are usually widened to reals with no loss of information.

However, the coercion of a real to an integer may lose information. Although 1.0 is exactly equal to the integer 1, 1.5 is not so representable as an integer. It will be converted to either 1 or 2. In this case, we call the coercion a *narrowing*, and information gets lost.

With dynamic type checking, coercions are made at the point that the type mismatch is detected during execution. For such languages, narrowing conversions could be allowed if the data object had an appropriate value (e.g., 1.0 could be converted to type integer, but 1.5 could not). For static type checking, extra code is inserted in the compiled program to invoke the conversion operation at the appropriate point during execution. Because efficient execution is usually a desired attribute, narrowing coercions are usually prohibited so that run-time code would not have to be executed to determine whether the coercion would be legal.

A type-conversion operation may require extensive change in the run-time storage representation of the data object. For example, in COBOL and PL/I, numbers often are stored in character-string form. To perform addition of such numbers on most machines, the character-string storage representation must be converted to a hardware-supported binary-number representation, with the result being converted back to character-string form before it is stored. The type-conversion operations here may take hundreds of times longer than the actual addition.

Implementors of language translators, however, sometimes confuse the semantics of a data object and its storage representation. Decimal data in COBOL and PL/I are a case in point. PL/I translators normally store FIXED DECIMAL data in packed decimal format. This is a hardware representation, but one that executes rather slowly. In the PL/C compiler [CONWAY and WILCOX 1973], FIXED DECIMAL data are stored as 16-digit double-precision floating-point data. By storing it as 16 digits, there is no loss of precision (important for storing decimal data). For example, computing 123.45 + 543.21 requires a rather slow packed decimal add (or even slower software simulation if packed decimal data are not hardware implemented), whereas PL/C performs this as a single, faster floating-point add of 12345 + 54321. The compiler keeps track of the decimal point (e.g., divide by 10^2 to get true value) as a compile-time attribute of the resulting value.

Two opposed philosophies exist regarding the extent to which the language should provide coercions between data types. In Pascal and Ada, almost no coercions are provided; any type mismatch, with few exceptions, is considered an error. In C, coercions are the rule; a type mismatch causes the compiler to search for an appropriate conversion operation to insert into the compiled code to provide the appropriate change of type. Only if no conversion is possible is the mismatch flagged as an error.

Type mismatch is a common minor programming error and type conversion is a common need, particularly in languages that have a large number of data types. There are also subtle questions as to the meaning of the notion type mismatch. (See Section 6.5.) Coercions often free the programmer from concern with what otherwise would be tedious detail—the invocation of numerous type-conversion operations explicitly in a program. However, coercions may also mask serious programming errors that might otherwise be brought to the programmer's attention during compilation.

PL/I, in particular, is infamous for the propensity of its compilers to take a minor programming error, such as a misspelled variable name, and, through a sometimes subtle coercion, hide the error so that it becomes a program bug that is difficult to detect. Because PL/I's narrowing coercions, results are surprising. For example, 9 + 10/3 is invalid! To see this, assume that 10/3 is converted into 3.33333333... up to the implementation-defined maximum number of digits. But 9 + 3.333... has one additional digit in the result, giving rise to an OVERFLOW exception. If this exception is disabled (i.e., ignored), then the result is truncated automatically to 2.333..., not quite what was expected.

5.1.5 Assignment and Initialization

Most of the operations for the common elementary data types—in particular, numbers, enumerations, Booleans, and characters—take one or two argument data objects of the type, perform a relatively simple arithmetic, relational, or other operation, and produce a result data object, which may be of the same or different type. The operation of assignment, however, is somewhat more subtle and deserves special attention.

Assignment is the basic operation for changing the binding of a value to a data object. This change, however, is a side effect of the operation. In some languages, such as C and LISP, assignment also returns a value, which is a data object containing a copy of the value assigned. These factors become clear when we try to write a specification for assignment. In Pascal, the specification for assignment of integers would be

$$assignment(:=) : integer_1 \times integer_2 \rightarrow void$$

with the action: Set the value contained in data object *integer*$_1$ to be a copy of the value contained in data object *integer*$_2$ and return no explicit result. (The change to *integer*$_1$ is an implicit result or side effect.) In C, the specification is

$$assignment(=) : integer_1 \times integer_2 \rightarrow integer_3$$

with the action: Set the value contained in data object *integer*$_1$ to be a copy of the value contained in data object *integer*$_2$ and also create and return a new data object *integer*$_3$, containing a copy of the new value of *integer*$_2$.

Consider the assignment $X := X$. What is interesting about this statement is the different interpretation given to both references of the variable X. The rightmost X refers to the value contained in the named data object. Such references are often called the *right-hand side (of the assignment operator) value* or *r-value* of a data object. Similarly, the leftmost X refers to the location of the data object that will contain the new value. Such references are called the *left-hand side (of the assignment operator) value* or *l-value*. We can then define an assignment operation as

1. Compute the l-value of the first operand expression.
2. Compute the r-value of the second operand expression.
3. Assign the computed r-value to the computed l-value data object.
4. Return the computed r-value as the result of the operation.

If we have an assignment operator (as in C), we then say that the operator returns the r-value of the newly assigned data object. In addition, C contains a set of unary operators for manipulating l-values and r-values of expressions that gives C programs the ability to perform many useful, and strange, manipulations of such assignments. This dual use of the assignment operator, as a mechanism to change a data object's value (via its l-value) and as a function that also returns a value (its r-value), is heavily exploited by C and is discussed further in chapter 8.

Using l-values and r-values gives a more concise way to describe expression semantics. Cons the C assignment $A = B$ for integers A and B. In C, as in many other languages, this means "Assig copy of the value of variable B to variable A" (i.e., assign to the l-value of A the r-value of B). N consider the assignment $A = B$, where A and B are pointer variables. If B is a pointer, then B's r-va is the l-value of some other data object. This assignment then means, "Make the r-value of A refe the same data object as the r-value of B" (i.e., assign to the l-value of A the r-value of B, which is l-value f some other data object). Thus, the assignment $A = B$ means "Assign a copy of the poi stored in variable B to variable A," as shown in Figure 5.2.

Equality and equivalence. The assignment statement is so pervasive in languages that few ques its semantics. However, there is a major issue that needs to be resolved. Consider the assignme A in some new language *Zork*:

$$A \rightarrow 2 + 3.$$

Does this mean

1. Evaluate the expression 2 + 3 and assign its equivalent value of 5 to A? or
2. Assign the operation "2 + 3" to A?

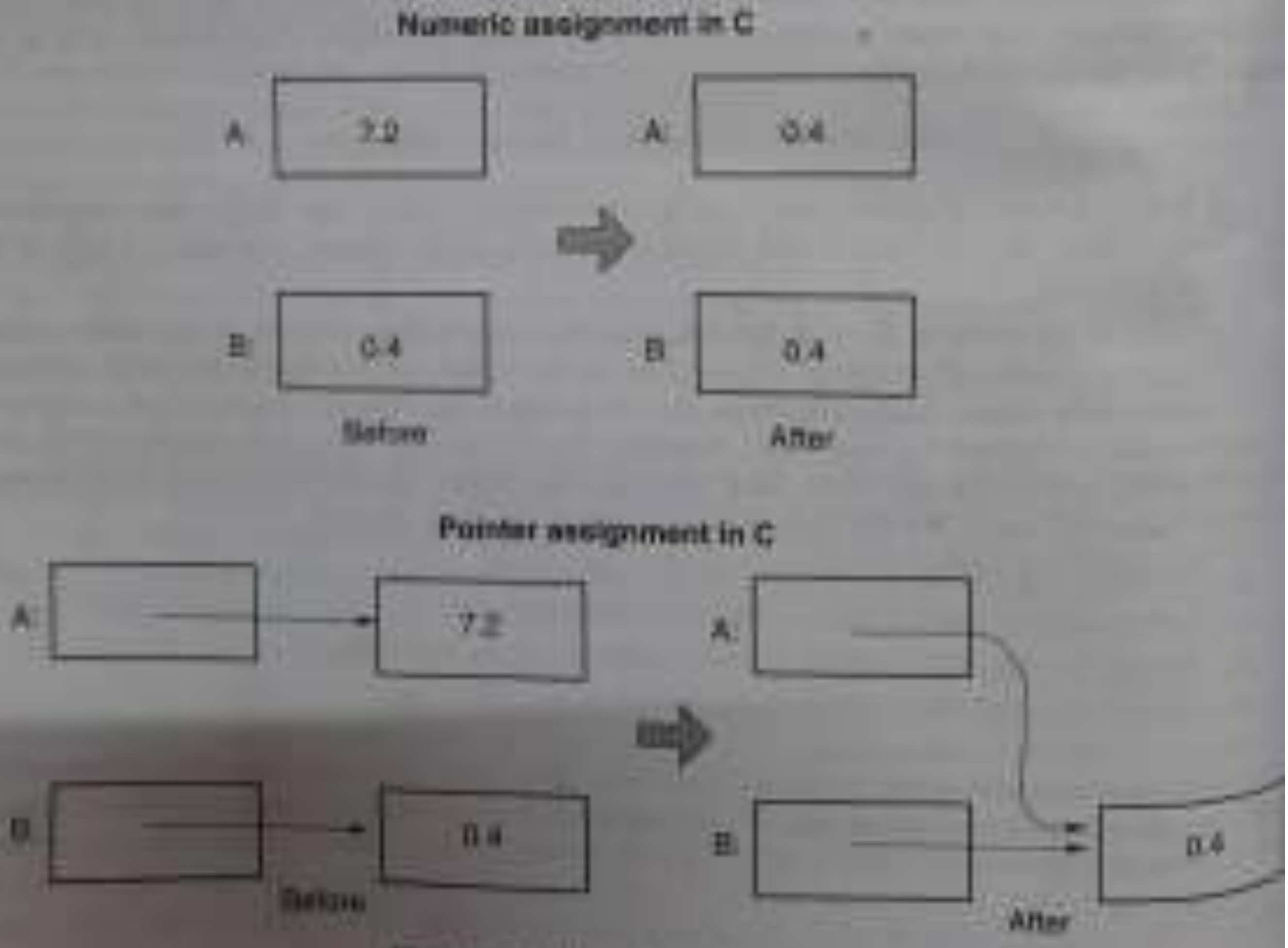


Figure 5.2. Two views of assignment.

In languages with static types for data, the type for *A* decides which semantics to follow. If *A* is of type integer, then only the assignment of 5 to *A* makes sense; if *A* is of type operation, then only the second is meaningful. However, for a dynamically typed language, where *A* is given a type by virtue of the assignment of a value to it, both semantics may be applicable and the prior assignment may be ambiguous.

Exactly this situation arises within Prolog. The operator is means assign the equivalent value, whereas the operator = means assign the pattern. Equality is then determined by the current value of the assigned variable. Thus, in Prolog, the first clause in the following succeeds because *X* is first assigned the value 5 and its type is set to integer, whereas the second clause fails because *X* is assigned the operation "2 + 3," which is not the same as the integer 5:

```
1. X is 2 + 3, X = 5.
2. X = 2 + 3, X = 5.
```

Note that in this case, the symbol = appears to stand for both an assignment operator and a Boolean comparison operator depending on context. Actually, this is an example of Prolog's unification principle (i.e., in the second clause, the assignment of both 2 + 3 and 5 to *X* are not mutually compatible). This is further explained in Section 8.4.3.

Initialization. An *uninitialized variable*, or more generally, an *uninitialized data object*, is a data object that has been created but not yet assigned a value (i.e., an l-value with no corresponding r-value). Creation of a data object ordinarily involves only allocation of a block of storage. Without any further action, the block of storage retains whatever bit pattern it happened to contain when the allocation was made. An explicit assignment is ordinarily required to bind a data object to a valid value. In some languages (e.g., Pascal), initialization must be done explicitly with assignment statements. In other languages (e.g., APL), initial values for each data object must be specified when the object is created; the assignment of initial values is handled implicitly without use of the assignment operation by the programmer.

Uninitialized variables are a serious source of programming error for professional programmers as well as beginners. The random bit pattern contained in the value storage area of an uninitialized data object ordinarily cannot be distinguished from a valid value because the valid value also appears as a bit pattern. Thus, a program often may compute with the value of an uninitialized variable and appear to operate correctly, when in fact it contains a serious error. Because of the effect of uninitialized variables on program reliability, immediate initialization of variable values on creation is often considered good programming practice, and newer languages such as Ada provide facilities to do this more easily. For example, in Ada, each variable declaration may also include an initial value for the variable using the same syntax used for ordinary assignment. For instance,

```
A: array(1..3) of float := (17.2, 20.4, 23.6);
```

creates an array *A* and assigns each element an initial value explicitly in the declaration. Because ordinarily the array *A* is created dynamically during program execution, the implementation of initial-value assignment requires the generation of code by the compiler that, when executed, explicitly assigns the specified initial values of the data object.

5.2 SCALAR DATA TYPES

We first consider a class of elementary data objects known as scalar data objects. These are objects that have a single attribute for its data object. For example, an integer object has an integral value

Programming lang Syntax :

(e.g., 1, 3, 47), and no other information can be obtained from that object. Next (Section 1.3) consider composite data where an object may have multiple attribute values. For example, a character string contains a sequence of characters as its data value, but also may have other attributes, such as a string size, as an additional attribute value.

In general the scalar objects follow the hardware architecture of a computer (e.g., integers, floating-point numbers, characters). Composite data are usually complex structures created by the compiler that are not primitive hardware-implemented objects (e.g., strings).

5.2.1 Numeric Data Types

Some form of numeric data is found in almost every programming language. Integer and real number types are the most common because they are often directly supported in the computer hardware. The properties of numeric data representations and arithmetic on computers differ substantially from the numbers and arithmetic operations discussed in ordinary mathematics. The fascinating subject of computer arithmetic has been treated in other books, however, and is not central to an understanding of the larger structure of programming languages, so only a brief treatment is given here.

Integers

Specification. The set of integer values defined for the type forms an ordered subset, within some finite bounds, of the infinite set of integers studied in mathematics. The maximum integer value is sometimes represented as a defined constant (e.g., in Pascal, it is the constant *maxint*). The range of values is then ordinarily defined to be from −*maxint* to *maxint*. As given earlier, C has four different integer specifications: int, short, long, and char.

Operations on integer data objects typically include the main groups:

Arithmetic operations. Binary arithmetic operations have the signature

BinOp: *integer* × *integer* → *integer*

where *BinOp* may be addition (+), subtraction (−), multiplication (*), division (/ or div), remainder (mod), or a similar operation. Unary arithmetic operations have the signature

UnaryOp: *integer* → *integer*

where, for example, *UnaryOp* may be negation (−) or identity (+). Commonly other arithmetic operations are included as well, often as library function subprograms (e.g., absolute value).

Relational operations. The relational operations each have the signature

RelOp: *integer* × *integer* → *Boolean*

where *RelOp* may be *equal*, *not equal*, *less than*, *greater than*, *less than or equal*, and *greater than or equal*. The relational operation compares the values of its two argument data values and returns a Boolean (true or false value) data object as its result.

Assignment. Assignment between integer data objects may be specified as one of:

assignment: *integer* × *integer* → *void* *assignment: integer* × *integer* → *integer*

as discussed in the preceding section.

Bit operations. In a language with few primitive data types, integers fulfill many roles. In C, integers also play the role of Boolean values. Therefore, additional bit operations are also defined using the operations

BinOp: integer × *integer* → *integer*

C includes operations to *and* the bits together (&), *or* the bits together (|), and shift the bits (<<), among others.

Implementation. The language-defined integer data type is most often implemented using a hardware-defined integer storage representation and set of hardware arithmetic and relational primitive operations on integers. Figure 5.3 shows three possible storage representations for integers. The first has no run-time descriptor; only the value is stored. This representation is possible where the language provides declarations and static type checking for integer data objects. The second form stores the descriptor in a separate memory location, with a pointer to the full-word integer value. This representation is often used in LISP. Its disadvantage is that it potentially may double the storage required for a single integer data object; its advantage is that the value is stored using the built-in hardware representation so that hardware arithmetic operations may be used. The third form stores the descriptor and value in a single memory location by shortening the size of the integer sufficiently to provide space for the descriptor. Here storage is conserved, but the hardware arithmetic operations

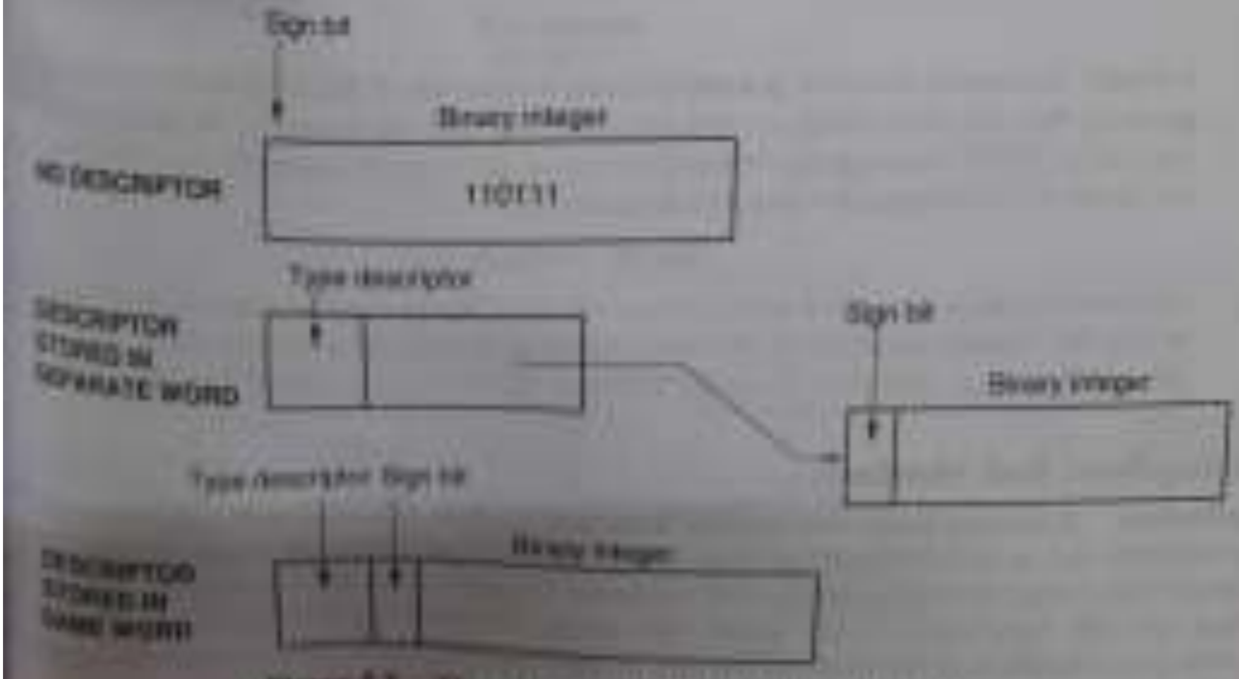


Figure 5.3. Three storage representations for integers.

cannot be used without first clearing the descriptor from the integer data object, performing t arithmetic and then reinserting the descriptor. Because a sequence of hardware instructions mu executed to perform a single arithmetic operation, arithmetic is inefficient. This form is only prac for hardware-implemented type descriptors, which are not common on the processors in use tod

Subranges

Specification. A subrange of an integer data type is a subtype of the integer data type and con of a sequence of integer values within some restricted range (e.g., the integers in the range 1 to 1 in the range −5 to 50). A declaration of the form A: 1..10 (Pascal) or A: *integer range* 1..10 (Ad often used. A subrange type allows the same set of operations to be used as for the ordinary in type; thus, a subrange may be termed a subtype of the *base type* integer.

Implementation. Subrange types have two important effects on implementations:

1. *Smaller storage requirements.* Because a smaller range of values is possible, a subra value can usually be stored in fewer bits than a general integer value. For example, an inte in the subrange 1..10 requires only four bits of storage for its representation, whereas a f integer value might require 16, 32, or more on typical machines. However, because arithm operations on shortened integers may need software simulation for their execution (a thus be much slower), subrange values are often represented as the smallest number of b for which the hardware implements arithmetic operations. This can generally be 8 or 16 b In C, for example, characters are stored as 8-bit integers that may be directly manipula by most microprocessor hardware.

2. *Better type checking.* Declaration of a variable as being of a subrange type allows m precise type checking to be performed on the values assigned to that variable. For exam if variable *Month* is: *Month*: 1..12, then the assignment

   Month := 0

   is invalid and can be detected at compile time. In contrast, if *Month* is declared to be of i ger type, then the assignment is valid and the error must be found by the programmer d ing testing. Many subrange type checks cannot be performed at compile time, however, if check involves a computed value. For example, in

   Month := Month + 1

   run-time checking is needed to determine whether the new value assigned to *Month* is within the bounds declared. In this case, the bounds must be available at run time to a the checking (e.g., *Month* = 12 would be legal; *Month* = 13 would be invalid).

Floating-Point Real Numbers

Specification. A floating-point real-number data type is often specified with only the single d type attribute *real*, as in FORTRAN, or *float*, as in C. As with type *integer*, the values form an order sequence from some hardware-determined minimum negative value to a maximum value, but values are not distributed evenly across the range. Alternatively, the precision required f floating-point numbers, in terms of the number of digits used in the decimal representation, may specified by the programmer, as in Ada.

The same arithmetic, relational, and assignment operations described for integers are usually provided for reals, although the Boolean operations are sometimes restricted. Due to roundoff ... equality between two real numbers is rarely achieved. Programs that check for equality to exit ... may never terminate. For this reason, equality between two real numbers may be prohibited ... the language designer to prevent this form of error. In addition, most languages provide other ... as built-in functions such as the sine and maximum value functions

$$sin: real \rightarrow real$$

and

$$max: real \times real \rightarrow real$$

...plementation. Storage representations for floating-point real types are ordinarily based on an underlying hardware representation in which a storage location is divided into a mantissa (i.e., the significant digits of the number) and an exponent, as shown in Figure 5.4. This model emulates scientific notation, where any number N can be expressed as $N = m \times 2^k$ for m between 0 and 1 and

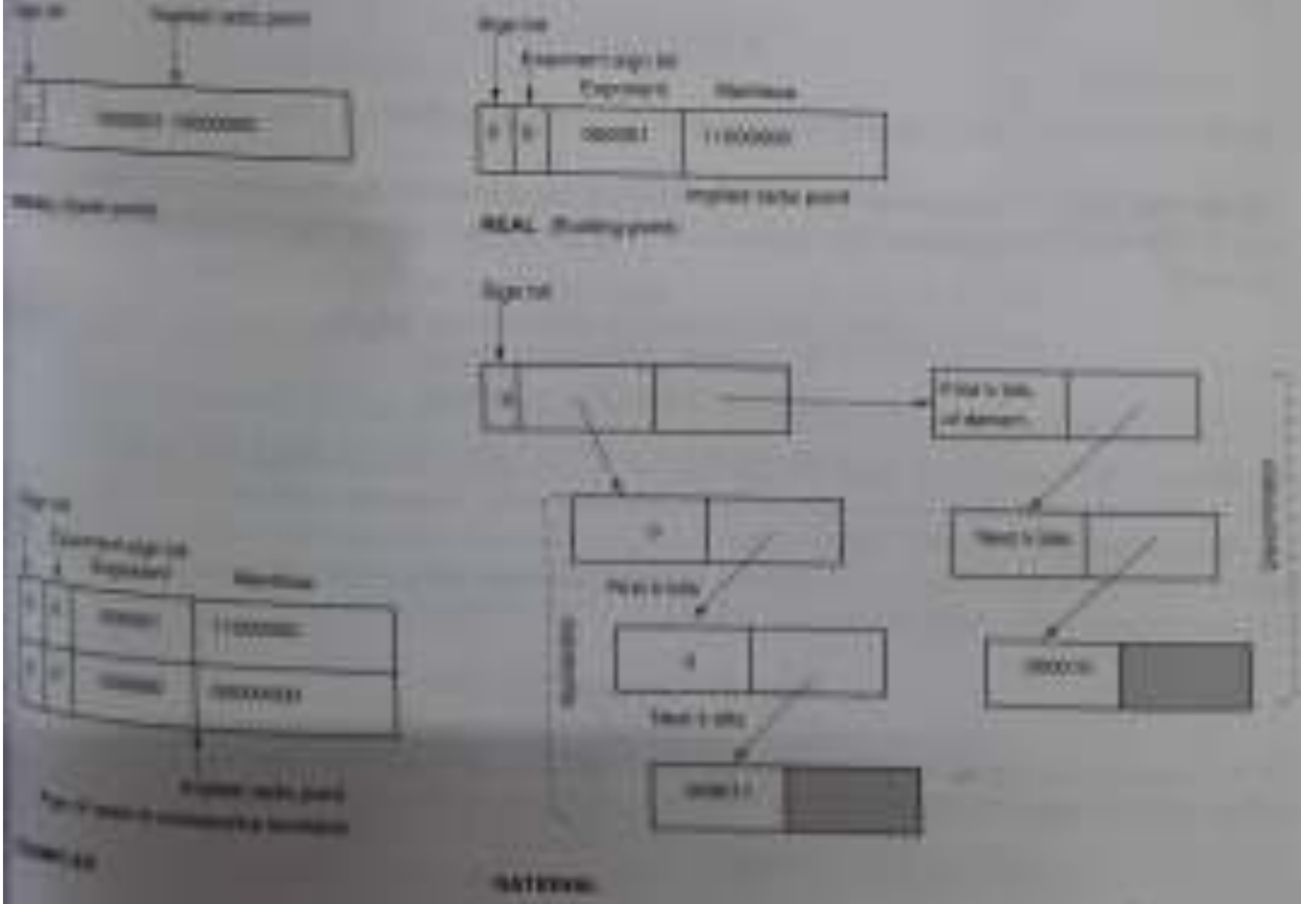


Figure 5.4. Representations of 1.5 without descriptors.

for some integer k. IEEE Standard 754 [IEEE 1985] has become the accepted definition for float-point format in many implementations. A double-precision form of floating-point number is also of available, in which an additional memory word is used to store an extended mantissa. Both single a double precision (if available) are generally supported by hardware arithmetic operations for additio subtraction, multiplication, and division. Exponentiation is usually software simulated. Where bo single- and double-precision real numbers are supported, the precision declaration for the numb of digits in the value of a particular real data object is used to determine whether single- or doub precision storage representation must be used. Alternatively the programmer may simply declare a real variable to be *double* or *long real* to specify use of double precision as the storage representatio

Fixed-Point Real Numbers

Specification. Although most hardware includes both integer and floating-point data objects, the are many applications where specific rational numbers are needed. For example, data objects repre senting money contain dollars and cents, which are rational numbers written to two decimal place These cannot be written as integers and, if written as floating-point values, may have roundoff erro A form of fixed-point data can be used to represent such values.

A fixed-point number is represented as a digit sequence of fixed length, with the decimal poi positioned at a given point between two digits. In COBOL, the declaration is given as a PICTURE clause—for example,

X PICTURE 999V99

which declares *X* as a fixed-point variable with three digits before the decimal and two digits afte

Implementation. A fixed-point type may be directly supported by the hardware or, as stated earlie may be simulated by software. For example, in PL/I, fixed data are of type FIXED DECIMAL. W can write

```
DECLARE X FIXED DECIMAL (10,3),
        Y FIXED DECIMAL (10,2),
        Z FIXED DECIMAL (10,2);
```

signifying *X* is 10 digits with three decimal places. *Y* and *Z* are also 10 digits, but have two decim places. We store such data as integers, with the decimal point being an attribute of the data object. I *X* has the value 103.421, the r-value of *X* will be 103421, and the object *X* will have an attribute *scale factor* (*SF*) of three, implying that th decimal point is three places to the left. That is,

$$\text{value}(X) = \text{rvalue}(X) \times 10^{-SF}$$

SF will always be 3 regardless of the r-value of *X*. Similarly, if *Y* has the value 102.34, then it will be stored as the integer 10234 with *SF* = 2.

Consider the execution of the statement

$$Z = X + Y$$

EXAMPLE 5.4. IEEE Floating-Point Format

IEEE standard 754 specifies both a 32- and 64-bit standard for floating-point numbers. The 32-bit standard is as follows:

S	exponent	mantissa
1	8	23

Numbers consist of three fields:

S – a one-bit sign field. 0 is positive.

E – an exponent in excess-127 notation. Values (8 bits) range from 0 to 255, corresponding to exponents of 2 that range from −127 to 128.

M – a mantissa of 23 bits. Since the first bit of the mantissa in a normalized number is always 1, it can be omitted and inserted automatically by the hardware, yielding an extra 24th bit of precision.

S determines the sign of the number. Given E, and M, the value of the representation is:

Parameters	Value
E=255 and M ≠ 0	An invalid number
E=255 and M=0	∞
$0 < E < 255$	$2^{E-127}(1.M)$
E=0 and M ≠ 0	$2^{-126}.M$
E=0 and M=0	0

Some examples:

$+1 = 2^0 \times 1 = 2^{127-127} \times (1).0(binary) =$ 0 01111111 000000...

$+1.5 = 2^0 \times 1.5 = 2^{127-127} \times (1).1(binary) =$ 0 01111111 100000...

$-5 = -2^2 \times 1.25 = 2^{129-127} \times (1).01(binary) =$ 1 10000001 010000...

This gives a range from 10^{-38} to 10^{38}. In 64-bit format, the exponent is extended to 11 bits giving a range from −1022 to +1023, yielding numbers in the range 10^{-308} to 10^{308}.

If we perform this task with paper and pencil, your first task is to line up the decimal points. Because X has three positions and Y has two, you need to shift Y left one position, and you know that the sum will have three decimal digits — that is, $SF = 3$:

X =	103.421	
Y =	102.34x	← Shift left 1 position
Sum =	205.761	Sum has SF = 3

The shift is equivalent to multiplying the integral r-value of Y by 10. The actual code to compute $X + Y$ is therefore $X + 10 \times Y$ with $SF = 3$. Because Z has only two decimal places ($SF = 2$) and the sum has three, we need to remove one place (divide by 10). Therefore, the code produced is

$$Z = (X + 10 \times Y) / 10$$

As long as the scaling factor is known at compile time (and it always is), the translator knows how to scale the results. Similarly, for multiplication $X \times Y$, we get the product by multiplying the two arguments together and adding the scale factors:

$$\begin{aligned} \text{Product} &= \text{rvalue}(X) \times \text{rvalue}(Y) \\ \text{SF} &= \text{SF}(X) + \text{SF}(Y) \end{aligned}$$

Subtraction and division are handled in an analogous manner.

Other Numeric Data Types

Complex numbers. A complex number consists of a pair of numbers representing the number's real and imaginary parts. A complex-number data type may be easily provided by representing each data object as a block of two storage locations containing a pair of real values. Operations on complex numbers may be software simulated because they are unlikely to be hardware implemented. (For example, addition would simply be the two additions on the corresponding real and imaginary parts of each argument, whereas multiplication is a more complex interaction involving all four real and imaginary components of the arguments.)

Rational numbers. A rational number is the quotient of two integers. The usual reason for including a rational number data type in a language is to avoid the problems of roundoff and truncation encountered in floating- and fixed-point representations of *reals*. As a result, it is desirable to represent rationals as pairs of integers of unbounded length. Such long integers are often represented using a linked representation. Figure 5.4 illustrates some of these number representations.

5.2.2 Enumerations

We often want a variable to take on only one of a small number of symbolic values. For example, a variable *StudentClass* might have only four possible values representing freshman, sophomore, junior, and senior. Similarly, a variable *EmployeeSex* might have only two values representing male and female. In older languages such as FORTRAN or COBOL, such a variable is ordinarily given the data type integer, and the values are represented as distinct, arbitrarily chosen integers (e.g., *Freshman* = 1, *Sophomore* = 2, and so on, or *Male* = 0, *Female* = 1). The program then manipulates these values as integers. The use of subranges for these special types often saves in storage requirements. However, in such cases, the programmer is responsible for ensuring that no operations are applied to the integer variables that make no sense in terms of the intended meaning. Assigning 2 to the variable *EmployeeSex* or multiplying *StudentClass* by *Female* (e.g., the integer 1) would make no sense here, but would be allowed by the translator.

Languages such as C, Pascal, and Ada include an *enumeration* data type that allows the programmer to define and manipulate such variables more directly.

Specification. An enumeration is an ordered list of distinct values. The programmer defines both the literal names to be used for the values and their ordering using a declaration such as the following in C:

```
enum   StudentClass {Fresh, Soph, Junior, Senior};
enum   EmployeeSex {Male, Female};
```

Because ordinarily many variables of the same enumeration type are used in a program, it is common to define the enumeration in a separate type definition and give it a type name that can then be used to specify the type of several variables. In Pascal, the prior C definition can be specified as

```
type Class = (Fresh, Soph, Junior, Senior);
```

followed by declarations for variables such as

```
StudentClass: Class;
TransferStudentClass: Class;
```

Note that the type definition introduces the type name Class, which may be used wherever a primitive type name such as *integer* might be used. It also introduces the literals of *Fresh*, *Soph*, *Junior*, and *Senior*, which may be used wherever a language-defined literal such as "27" might be used. Thus, we can write

```
if StudentClass = Junior then ...
```

instead of the less understandable

```
if StudentClass = 3 then ...
```

which would be required if integer variables were used. In addition, static type checking by the compiler could find programming errors such as

```
if StudentClass = Male then ...
```

The basic operations on enumeration types are the relational operations (equal, less-than, greater-than, etc.), assignment, and the operations *successor* and *predecessor*, which give the next and previous value, respectively, in the sequence of literals defining the enumeration (and are undefined for the last and first values, respectively). Note that the full set of relational operations is defined for enumeration types because the set of values is given an ordering in the type definition.

Implementation. The storage representation for a data object of an enumeration type is straightforward. Each value in the enumeration sequence is represented at run time by one of the integers 0, 1, 2, ... Because only a small set of values is involved and the values are never negative, the usual integer representation is often shortened to omit the sign bit and use only enough bits for the range of values required, as with a subrange value. For example, the type Class defined earlier has only four possible values, represented at run time as 0 = *Fresh*, 1 = *Soph*, 2 = *Junior*, and 3 = *Senior*. Because only two bits are required to represent these four possible values in memory, a variable of type Class needs to be allocated only two bits of storage. The successor and predecessor operations involve simply adding or subtracting one from the integer representing the value and checking to see that the result is within the proper range.

In C the programmer may override this default and set any values desired for enumeration values—for example,

```
enum class {Fresh=14, Soph=36, Junior=4, Senior=42}
```

Given this storage representation for enumeration types, implementation of the basic operations on enumeration types is also straightforward because the hardware-provided operations on integers may be used. For example, relational operations such as =, >, and < may be implemented using the corresponding hardware primitives that compare integers.

5.2.3 Booleans

Most languages provide a data type for representing *true* and *false*, usually called a *Boolean* or *l*
data type.

Specification. The Boolean data type consists of data objects having one of two values—*true* or
In Pascal and Ada, the Boolean data type is considered simply a language-defined enumeration,

type Boolean = (false, true);

which both defines the names *true* and *false* for the values of the type and defines the ord
false < *true*.

The most common operations on Boolean types include assignment as well as the following lo
operations:

and:	*Boolean* × *Boolean* → *Boolean*	(conjunction)
or:	*Boolean* × *Boolean* → *Boolean*	(inclusive disjunction)
not:	*Boolean* → *Boolean*	(negation or complement)

Other Boolean operations such as equivalence, exclusive or, implication, nand (not-and), and
(not-or) are sometimes included. *Short-circuit* forms of the operations *and* and *or* are discuss
Section 8.2.

Implementation. The storage representation for a Boolean data object is a single bit of storage p
vided no descriptor designating the data type is needed. Because single bits may not be separa
addressable in memory, often this storage representation is extended to be a single addressable u
such as a byte or word. Then the values *true* and *false* might be represented in two ways within t
storage unit:

- A particular bit is used for the value (often the sign bit of the number representation) w
0 = *false*, 1 = *true*, and the rest of the byte or word ignored; or
- A zero value in the entire storage unit represents *false*, and any other nonzero va
represents *true*.

Because a large amount of storage may be used by either of these representations, provision is o
made in a language for collections of bits. The *bit-string* data type of PL/I and the *packed arra*
Boolean and set data types of Pascal are examples.

Java has an explicit Boolean type, but C does not. In C, integer is used, *true* is any nonzero va
and *false* is 0. This can potentially cause some problems. For example,

```
int flag;
flag = 7
```

sets *flag* to 7 or 111 in binary, which would represent a true value. However, if you negate *flag* b
changing all the bits in its representation, then the value of *flag* in binary would be ... 111000, or s
true. Similarly, combining flags using the bitwise *or* operation | instead of the logical *or* operation |
using the bitwise *and* operation & instead of the logical *and* && leads to similar problems. In C, it
always safer to represent *true* by the integer value 1 and not try to pack multiple Boolean values in
the same multibit integer.

5.2.4 Characters

Most data are input and output in character form. Conversion during input and output to other data types is usually provided, but processing of some data directly in character form is also important. Sequences of characters (character strings) are often processed as a unit. Provision for character-string data may be provided either directly through a character-string data type (as in ML and Prolog) or through a character data type, with a character string considered as a linear array of characters (as in C, Pascal and Ada). Character strings are considered in Section 5.3.1.

Specification. A character data type provides data objects that have a single character as their value. The set of possible character values is usually taken to be a language-defined enumeration corresponding to the standard character sets supported by the underlying hardware and operating system, such as the ASCII character set. The ordering of the characters in this character set is called the collating sequence for the character set. The collating sequence is important because it determines the alphabetical ordering given to character strings by the relational operations. Because the ordering includes all characters in the set, character strings that include spaces, digits and special characters may be alphabetized as well. Operations on character data include only the relational operations, assignment and sometimes operations to test whether a character value is one of the special classes letter, digit or special character.

Implementation. Character data values are almost always directly supported by the underlying hardware and operating system because of their use in input-output. Occasionally, however, the language definition prescribes use of a particular character set (such as the ASCII set) that is not supported by the underlying hardware. Although the same characters may be represented in both character sets, their storage representations, and thus their collating sequences may differ, or other some special characters in one set may not exist in the other. Because characters arrive from the input-output system in the hardware-supported representation, the language implementation may have to provide appropriate conversions to the alternative character-set representation or provide special implementation of the relational operations that take account of differences in the collating sequence. If the language-defined character representation is the same as that supported by the hardware, then the relational operations also are usually represented directly in the hardware or may be simulated by short in-line code sequences.

5.3 COMPOSITE DATA TYPES

The data types in this section are usually considered elementary data objects. However, their implementation usually involves a complex data structure organization by the compiler. Multiple attributes are often given for each such data type.

5.3.1 Character Strings

A character string is a data object composed of a sequence of characters. A character-string data type is important in most languages owing in part to the use of character representations of data for input and output.

Specification and syntax. At least three different treatments of character-string data types may be identified:

5.2.4 Characters

Most data are input and output in character form. Conversion during input and output to other data types is usually provided, but processing of some data directly in character form is also important. Sequences of characters (character strings) are often processed as a unit. Provision for character string data may be provided either directly through a character-string data type (as in ML and Prolog) or through a character data type, with a character string considered as a linear array of characters (as in C, Pascal, and Ada). Character strings are considered in Section 5.3.1.

Specification. A character data type provides data objects that have a single character as their value. The set of possible character values is usually taken to be a language-defined enumeration corresponding to the standard character sets supported by the underlying hardware and operating system, such as the ASCII character set. The ordering of the characters in this character set is called the collating sequence for the character set. The collating sequence is important because it determines the alphabetical ordering given to character strings by the relational operations. Because the ordering includes all characters in the set, character strings that include spaces, digits and special characters may be alphabetized as well. Operations on character data include only the relational operations, assignment, and sometimes operations to test whether a character value is one of the special classes letter, digit or special character.

Implementation. Character data values are almost always directly supported by the underlying hardware and operating system because of their use in input-output. Occasionally, however, the language definition prescribes use of a particular character set (such as the ASCII set) that is not supported by the underlying hardware. Although the same characters may be represented in both character sets, their storage representation, and thus their collating sequences, may differ; in other cases, some special characters in one set may not exist in the other. Because characters arrive from the input-output system in the hardware-supported representation, the language implementation may have to provide appropriate conversions to the alternative character set representation or provide special implementation of the relational operations that take account of differences in the collating sequences. If the language-defined character representation is the same as that supported by the hardware, then the relational operations also are usually represented directly in the hardware or may be simulated by short inline code sequences.

5.3 COMPOSITE DATA TYPES

The data types in this section are usually considered elementary data objects. However, their implementation usually involves a complex data structure organization by the compiler. Multiple attributes are often given for each such data type.

5.3.1 Character Strings

A character string is a data object composed of a sequence of characters. A character-string data type is important in most languages, owing in part to the use of character representations of data for input and output.

Specification and syntax. At least three different treatments of character-string data types may be identified:

1. Fixed declared length. A character-string data object may have a fixed length that is declared in the program. The value assigned to the data object is always a character string of this length. Assignment of a new string value to the data object results in a length adjustment of the new string through truncation of excess characters or addition of blank characters to produce a string of the correct length.

2. Variable length to a declared bound. A character-string data object may have a maximum length that is declared in the program, but the actual value stored in the data object may be a string of shorter length—possibly even the empty string of no characters. During execution, the length of the string value of the data object may vary, but it is truncated if it exceeds the bound.

3. Unbounded length. A character-string data object may have a string value of any length, and the length may vary dynamically during execution with no bound (beyond available memory).

The C case is actually a bit more complicated. In C, strings are arrays of characters, but C has no string declaration. However, there is a convention that a null character ("\0") follows the last character of a string. A string constant written as "This string has null terminator" will have the null character appended to the string constant by the C translator; that is the string, when stored in an array will have a null character appended. However, for strings made from programmer-defined arrays, it is the programmer's responsibility to make sure that strings include the final null character.

The first two methods of handling character-string data allow storage allocation for each string data object to be determined at translation time; if strings have unbounded length, then dynamic storage allocation at run time is required. The different methods also make different sorts of operations on strings appropriate.

A wide variety of operations on character-string data are provided. Some of the more important are briefly described here.

1. Concatenation. Concatenation is the operation of joining two character strings to make one long string. For example, if || is the symbol for the concatenation operation, then "BLOCK" || "HEAD" gives "BLOCKHEAD".

2. Relational operations on strings. The usual relational operations (equal, less than, greater than, etc.) may be extended to strings. The basic character set always has an ordering, as described in Section 5.2.4. Extending this ordering to character strings gives the usual lexicographic (alphabetic) ordering, in which String A is considered to be less than String B (i.e., comes before it in the ordering) if either the first character of A is less than the first character of B, or if the first characters are equal and the second character of A is less than the second character of B, and so on, with the shorter of Strings A and B extended with blank characters (spaces) to the length of the longer.

3. Substring selection using positioning subscripts. Manipulation of character strings often involves working with a contiguous substring of the overall string. For example, a string often contains a leading sequence of blank characters or be composed of words separated by blanks and punctuation marks. To facilitate manipulation of substrings, many languages provide an operation for selecting a substring by giving the positions of its first and last characters (or first character position and length of the substring

1. *Fixed declared length.* A character-string data object may have a fixed length that is declared in the program. The value assigned to the data object is always a character string of this length. Assignment of a new string value to the data object results in a length adjustment of the new string through truncation of excess characters or addition of blank characters to produce a string of the correct length.
2. *Variable length to a declared bound.* A character-string data object may have a maximum length that is declared in the prior program, but the actual value stored in the data object may be a string of shorter length—possibly even the empty string of no characters. During execution, the length of the string value of the data object may vary, but it is truncated if it exceeds the bound.
3. *Unbounded length.* A character-string data object may have a string value of any length, and the length may vary dynamically during execution with no bound (beyond available memory).

The C case is actually a bit more complicated. In C, strings are arrays of characters, but C has no string declaration. However, there is a convention that a null character ("\0") follows the last character of a string. A string constant written as "This string has null terminator" will have the null character appended to the string constant by the C translator; that is, the string, when stored in an array, will have a null character appended. However, for strings made from programmer-defined arrays, it is the programmer's responsibility to make sure that strings include the final null character.

The first two methods of handling character-string data allow storage allocation for each string data object to be determined at translation time; if strings have unbounded length, then dynamic storage allocation at run time is required. The different methods also make different sorts of operations on strings appropriate.

A wide variety of operations on character-string data are provided. Some of the more important are briefly described here:

1. *Concatenation.* Concatenation is the operation of joining two character strings to make one long string. For example, if || is the symbol for the concatenation operation, then "BLOCK" || "HEAD" gives "BLOCKHEAD."
2. *Relational operations on strings.* The usual relational operations (equal, less-than, greater-than, etc.) may be extended to strings. The basic character set always has an ordering, as described in Section 5.2.4. Extending this ordering to character strings gives the usual lexicographic (alphabetic) ordering, in which string *A* is considered to be less than string *B* (i.e., comes before it in the ordering) if either the first character of *A* is less than the first character of *B*, if the first characters are equal and the second character of *A* is less than the second character of *B*, and so on, with the shorter of strings *A* and *B* sometimes extended with blank characters (spaces) to the length of the longer.
3. *Substring selection using positioning subscripts.* Manipulation of character-string data often involves working with a contiguous substring of the overall string. For example, a string may often contain a leading sequence of blank characters or be composed of words separated by blanks and punctuation marks. To facilitate manipulation of substrings of a string, many languages provide an operation for selecting a substring by giving the positions of its first and last characters (or first character position and length of the substring), as in

the FORTRAN *NEXT* = *STR*(6 : 10), which assigns the five characters in Positions 6 through 10 of string *STR* to string variable *NEXT*. The meaning of the substring selection operation is particularly troublesome to define if it is allowed to appear on both sides of an assignment (i.e., it is both an r-value and l-value function so that a new value may be assigned to the selected substring). Consider the FORTRAN expression

STR(1:5)=STR(I:I+4)

which might be used to move a five-character substring beginning in Position *I* to the first five-character positions of the string. If the substrings referenced on the left and right of the assignment happen to overlap, the meaning of this statement must be carefully defined.

4. *Input-output formatting.* Operations for manipulating character strings are often provided primarily to aid in formatting data for output or for breaking up formatted input data into smaller data items. The formatted input-output features of FORTRAN and C are examples of extensive sets of operations provided for this purpose.

5. *Substring selection using pattern matching.* Often the position of a desired substring within a larger string is not known, but its relation to other substrings is known. For example, we might want to select the first nonblank character of a string, a sequence of digits followed by a decimal point, or the word following the word *THE*. A *pattern-matching* operation takes as one argument a pattern data structure, where the pattern specifies the form of the substring desired (e.g., its length, or that it is composed of a sequence of decimal digits) and possibly other substrings that should adjoin it (e.g., a following decimal point or a preceding sequence of blank characters). The second argument to the pattern-matching operation is a character string that is to be scanned to find a substring that matches that specified by the pattern. We have already encountered this in Perl in its handling of regular expressions (Section 3.3.3).

6. *Dynamic strings.* String values may be either static or dynamic. Perl is an example where both are possible. The string '$ABC' is static, and the statement

print '$ABC';

will print the value $ABC. However, the string "$ABC" is dynamic and the statement

print "$ABC";

will cause the string to be evaluated, and the value of the Perl variable $ABC is printed instead.

Implementation. Each of the three methods for handling character strings utilizes a different storage representation, as shown in Figure 5.5. Storage representations for characters are discussed in Section 5.2.6. For a string of fixed, declared length, the representation is essentially that used for a packed vector of characters, as described in Section 6.1.5. For the variable-length string to a declared bound, the storage representation utilizes a descriptor containing both the declared maximum length and the current length of the string stored in the data object. For strings of unbounded length, either a linked storage representation of fixed-length data objects may be used or a contiguous array of characters may be used to contain the string. This latter method is the technique used in C, which often requires dynamic run-time storage management to allocate such storage.

Hardware support for the simple fixed-length representation is usually available, but other representations for strings must usually be software simulated. Operations on strings such as concatenation, substring selection, and pattern matching are ordinarily entirely software simulated.

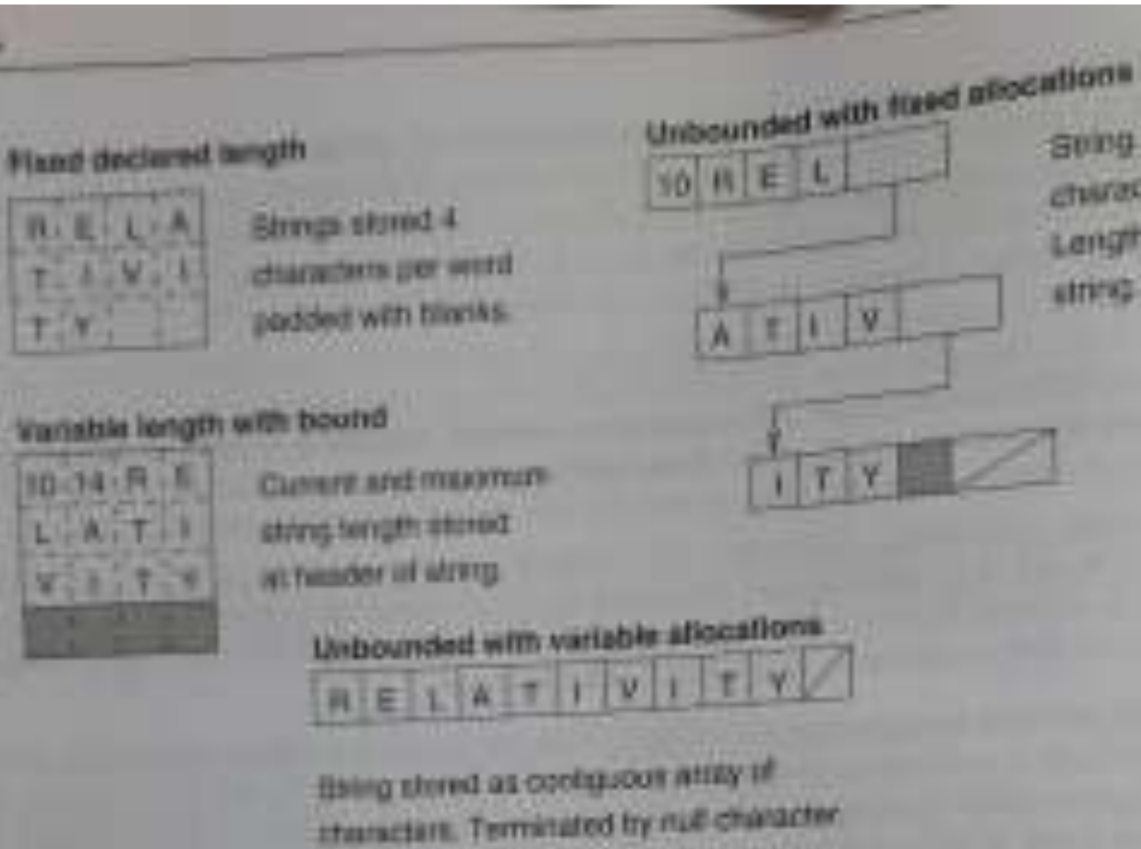


Figure 5.5. Storage representation for strings.

5.3.2 Pointers and Programmer-Constructed Data Objects

Commonly, rather than including a variety of types of variable-size, linked data objects into a programming language, facilities are provided to allow the construction of any structure using pointers to link together the component data objects as desired. Several language features are needed to make this possible:

1. An elementary data type *pointer* (also called a *reference* or *access* type). A pointer data object contains the location of another data object (i.e., its l-value) or may contain the null pointer, nil or null. Pointers are ordinary data objects that may be simple variables or components of arrays and records.

2. A *creation operation* for data objects of fixed size, such as arrays, records, and elementary types. The creation operation both allocates a block of storage for the new data object and returns its l-value, which may then be stored as the r-value of a pointer data object. The creation operation differs in two ways from the ordinary creation of data objects caused by declarations: (a) The data objects created need have no names because they are accessed through pointers, and (b) data objects may be created in this way at any point during execution of a program, not just on entry to a subprogram.

3. A *dereferencing operation* for pointer values that allows a pointer to be followed to the data object to which it points.

Specification. A pointer data type defines a class of data objects whose values are the locations of other data objects. A single data object of pointer type might be treated in two ways:

1. *Pointers may reference data objects only of a single type.* The pointers (locations) that are allowed as a value of the pointer data object may be restricted to point only to data objects of the same type. This is the approach used in C, Pascal, and Ada, where type declarations and static type checking are used. To declare a pointer variable in C (e.g., that may point to any data object of type *List*), use

```
List *P;
```

the * designates the type of *P* as type pointer. The type *List* designates that the value of *P* may be the l-value of an object of type *List*. A separate type definition must be given to define the structure of data objects of type *List*:

```
struct List { int ListValue; List *NextItem; };
```

2. *Pointers may reference data objects of any type.* An alternative is to allow a pointer data object to point to data objects of varying types at different times during program execution. This is the approach used in languages like Smalltalk, where data objects carry type descriptors during execution and dynamic type checking is performed.

In some languages (e.g., C and C++), pointers are data objects that may be manipulated by the program. In others (e.g., Java), pointers are part of the hidden data structures managed by the language implementation.

The creation operation allocates storage for (and thus creates) a fixed-size data object and also creates a pointer to the new data object that may be stored in a pointer data object. In Ada, this operation is named *new*. In C, the system function malloc (memory allocator) provides this function. (C++ and Java simplify C by renaming the function *new* as an allocator.) Consider a subprogram containing a declaration of pointer variable *P* (as defined earlier). On entry to the subprogram, only storage for the data object *P* is allocated (storage for a single pointer value). Later during execution of the subprogram, data of type *List* may be created by executing the statement

```
P = malloc(sizeof(List))
```

Because *P* has been declared to point to objects only of the type *List*, this statement has the meaning "Create a two-word block of storage to be used as an object of type *List*, and store its l-value in P."

The selection operation allows a pointer value to be followed to reach the data object designated. Because pointers are ordinary data objects, the pointer data object may also be selected using only the ordinary mechanisms for selection. For example, in C the selection operation that follows a pointer to its designated object is written *. To select a component of the vector pointed to by *P*, you write **P.first*. The * operator simply takes the r-value of the pointer and makes it an l-value. Thus, **P.first* accesses the value in *P*, assumes it is now an l-value, and uses this to access the *first* component of the record pointed to by *P*.

Implementation. A pointer data object is represented as a storage location containing the address of another storage location. The address is the base address of the block of storage representing the data object pointed to by the pointer. Two major storage representations are used for pointer values:

1. *Absolute addresses.* A pointer value may be represented as the actual memory address of the storage block for the data object.
2. *Relative addresses.* A pointer value may be represented as an *offset* from the *base address* of some larger heap storage block within which the data object is allocated.

If absolute addresses are used for pointer values, then data objects created by the crea operation, new, may be allocated storage anywhere in memory. Usually this allocation takes place wi a general *heap* storage area. Selection using absolute addresses is efficient because the pointer va provides direct access to the data object using the hardware memory-accessing operations. The disadva tage of absolute addresses is that storage management is more difficult because no data object may b moved within memory if there exists a pointer to it stored elsewhere unless the pointer value is chang to reflect the new position of the data object. Recovery of storage for data objects that have becom garbage is also difficult because each such data object is recovered individually and its storage blo must be integrated back into the overall pool of available storage. These issues are treated in chapter

The use of relative addresses as pointers requires the initial allocation of a block of storage wi which subsequent allocation of data objects by new takes place. There may be one area for each type data object to be allocated, or a single area for all data objects. Each area is managed as a *heap* stora area. Assuming one area for each type of data object, then new may allocate storage in fixed-size blo within the area, which makes storage management particularly simple. Selection using this form pointer value is more costly than for an absolute address because the offset must be added to the ba address of the area to obtain an absolute address before the data object may be accessed. However, th advantage of relative pointers lies in the opportunity to move the area block as a whole at any time wi out invalidating any of the pointers. For example, the area might be written out to a file and later re back into a different place in primary memory. Because the offset of each pointer value is unchange access to a data object in the new area may use the same offset with the new base address of the ar An additional advantage is that the entire area may be treated as a data object that is created on en to a subprogram, used by new within that subprogram (and any subprograms it calls), and then dele on exit. No storage recovery of individual data objects within the area is necessary; they may be allow to become garbage because the entire area is recovered as a whole when the subprogram is exited.

Static type checking is possible for references using pointer values if each pointer data obje restricted to point to other data objects of a single type, as described earlier. Without this restricti it cannot be determined during translation what type of data object a pointer will designate at r time, so dynamic type checking is necessary. In some languages, selections using pointer values a simply left unchecked. Run-time checking for a *nil* pointer value is also required before selection.

The major implementation problem associated with pointers and programmer-constructed da objects that use pointer linkages is the storage allocation associated with the creation opera Because this operation may be used to create data objects of different sizes at arbitrary times du program execution, it requires an underlying storage management system capable of manag general heap storage area. For C, the other parts of the language require only a stack-based stor management system, so the addition of pointers and the malloc operation to the language requ substantial extension of the overall run-time storage management structure. Pointer data ob introduce the potential for generating garbage if all pointers to a created data object are lo Dangling references are also possible if data objects can be destroyed and the storage recovered f reuse. These issues are treated again in chapter 6.

5.3.3 Files and Input-Output

A *file* is a data structure with two special properties:

1. It ordinarily is represented on a secondary storage device such as a disk or tape and th may be much larger than most data structures of other types.
2. Its lifetime may encompass a greater span of time than that of the program creating it.

Sequential files are the most common type of file, but many languages also provide *direct-access files* and *indexed sequential files*. Two general uses for files are seen: for input and output of data to the external operating environment (see chapter 11) and as temporary *scratch storage* for data when not enough high-speed memory is available. The components of a file are often termed *records*, but this use of the term is avoided here because of the conflict with the record data structure discussed in Section 6.1.6.

Sequential Files

A *sequential file* is a data structure composed of a linear sequence of components of the same type. It is of varying length with no fixed maximum bound (beyond available storage). In Pascal, a file is declared by giving its name and the type of component it contains. For example,

Master: **file of** EmployeeRec;

defines a file named *Master* whose components are of type *EmployeeRec*. Variable-size data structures ordinarily cannot be components of files (thus, no files of files or files of stacks). In addition, data structures that are in linked representation or that include pointer values are often not allowed as components of files because the pointer values usually become meaningless after the lifetime of the program creating the file. When the data are later read from the file, the storage locations referenced by the pointer values may be in use for another purpose.

For input-output, data are usually represented in character form. Each component of such a file is then a single character, and the file is known in Pascal as a *textfile*. Most languages provide a special set of input-output operations for textfiles, in addition to the ordinary file operations. A textfile also commonly divides character sequences into groups called *lines*. We consider ordinary sequential files first and then note some special characteristics of textfiles.

Typically the file may be accessed in either *read mode* or *write mode*. In either mode, there is a *file-position pointer* that designates a position before the first file component, between two components, or after the last component. In write mode, the file-position pointer is always positioned after the last component, and the only operation possible is to assign (*write*) a new component to that position, thus extending the file by one component. In read mode, the file-position pointer may be positioned anywhere in the file, and the only access provided is access (*read*) to the component at (immediately following) the position designated. No assignment of new components or component values is provided.

Specification. The major operations on sequential files are as follows:

1. *Open*. Ordinarily, before a file may be used, it must be opened. The open operation is given the name of a file and the access mode (read or write). If the mode is *read*, then the file is presumed to already exist. The *open* operation ordinarily requests information from the operating system about the location and properties of the file, allocates the required internal storage for buffers and other information (see the following implementation discussion), and sets the file-position pointer to the first component of the file. If the mode is *write*, then a request is made to the operating system to create a new empty file or, if a file already exists with the given name, delete all the existing components of the file so that it is empty. The file-position pointer is set to the start of the empty file.

   Ordinarily, an explicit open statement is provided. In Pascal, the procedure *reset* opens a file in read mode, and procedure *rewrite* opens a file in write mode. Sometimes a language provides for an *implicit* open operation on a file at the time of the first attempt to read or write the file.

2. *Read.* A *read* operation transfers the contents of the current file component (desig by the file-position pointer) to a designated variable in the program. The transfer is defined as having the same semantics as an assignment from the file component to the pro variable.

3. *Write.* A *write* operation creates a new component at the current position in the (always at the end) and transfers the contents of a designated program variable to the component. Again this transfer is usually defined as a form of assignment.

4. *End-of-file test.* A *read* operation fails if the file position pointer designates the end of file. Because the file is of variable length, an explicit test for the end-of-file posi needed so that the program may take special action.

5. *Close.* When processing of a file is complete, it must be closed. Ordinarily, this oper involves notification to the operating system that the file can be detached from the prog (and potentially made available to other programs) and possibly also deallocation of stor storage used for the file (such as buffers and buffer variables). Often files are closed ally when the program terminates without explicit action by the programmer. Howev change the mode of access to a file from write to read, or vice versa, the file must explicitly closed and then reopened in the new mode.

Implementation. In most computer systems, the underlying operating system has the prim responsibility for the implementation of files because files are created and manipulated by vari programming language processors and utilities. File operations are primarily implemented by on primitives provided by the operating system.

From the language viewpoint, the primary implementation problem comes from the need provide storage for system data and buffers required by the operating system primitives. Typi when a program opens a file during its execution, storage for a *file information table* and a *buffer* be provided. The operating system open primitive stores information about the location and charac istics of the file in the file information table. Assume the file is opened in write mode. When a w operation transfers a component to be appended to the end of the file, the data are sent to an op ating system write primitive. The *write* primitive stores the data in the next available position in buffer, in memory, and updates pointers to this buffer in the file information table. No actual tr fer of data to the file takes place until enough write operations have been performed to allow a co plete block of components to have accumulated in the buffer. At this time the block of compon is transferred from the buffer to the external storage device (e.g., disk or tape). The next seque *write* operations executed by the program again fills the buffer until a complete block may be tr ferred to external storage. When a file is read, the reverse process occurs: Data are transferred the file into the buffer in blocks of components. Each *read* operation executed by the program tr fers a single component from the buffer to the program variable. The buffer is refilled as needed. Th organization is shown in Figure 5.6.

Textfiles

A *textfile* (the term comes from Pascal) is a file of characters. Textfiles are the primary form of f for input-output to the user in most languages because textfiles may be printed and may be crea directly from keyboard input. Files with components of other types ordinarily are only written

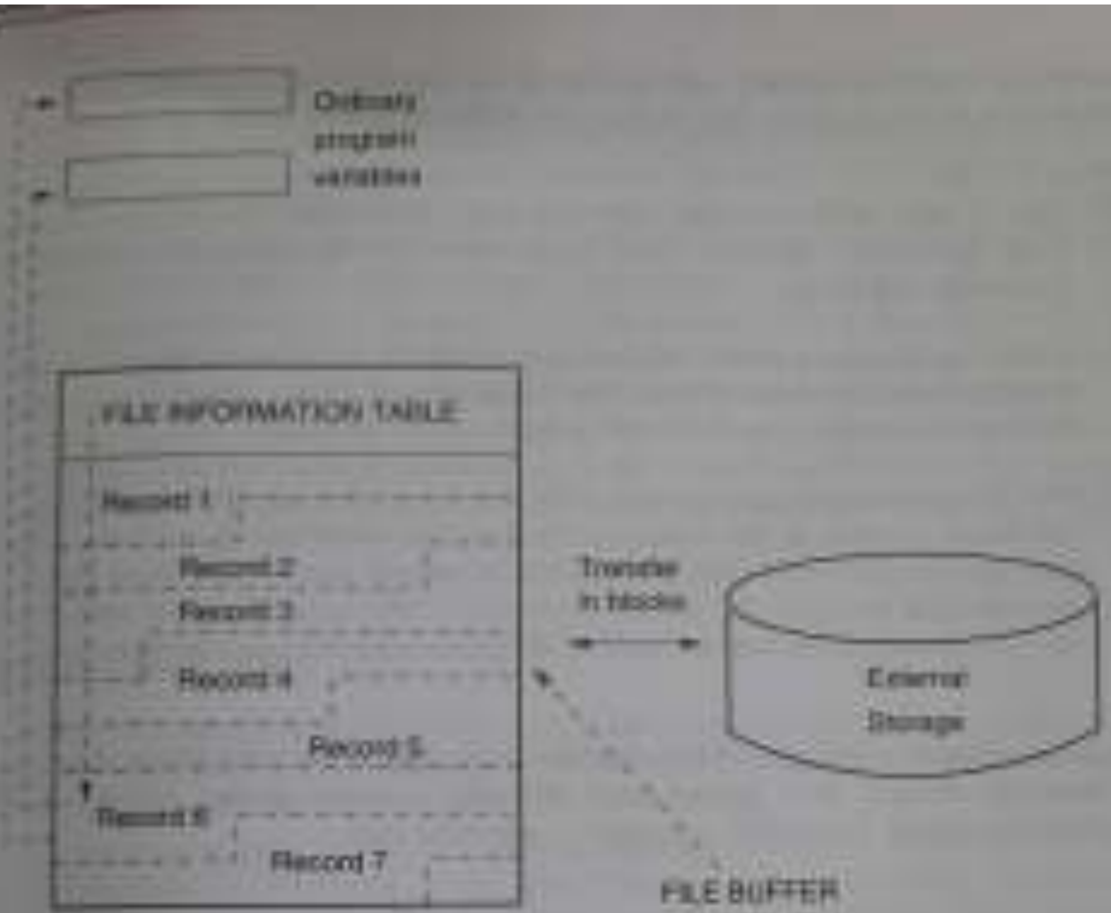


Figure 5.6. File representation using a buffer.

by programs. Textfiles are a form of ordinary sequential file and may be manipulated in the ways. However, special operations are often provided for textfiles that allow numeric data (and other types of data) to be automatically converted to internal storage representations input without storage in character form. Similar operations on output allow conversion of and other data from internal representations to character form. Along with these conversions, is ordinarily made for formatting the data on output into lines of the appropriate length include headings, spaces, and converted data items as desired for printed output. This output is an important part of the implementation of output operations for textfiles. Similar operations may be used for input, or the input operations may be *free format*, allowing to appear anywhere on a line separated by spaces. The free-format *read* operation scans the as necessary to find the numbers needed to satisfy the read request.

ctive Input–Output

a textfile that represents an interactive terminal at which a programmer is sitting. During of the program, a write operation on this file is interpreted as a command to display the on the terminal screen. A *read* operation is a command that requests input of data from the keyboard, usually beginning with display of a prompt character on the screen. In this setting, several aspects of the ordinary view of sequential files described earlier are modified:

1. The file must be in both read and write mode at the same time because ordinarily read and write operations alternate. First some data are displayed, then some input data is requested, and so on.

2. Buffering of data on input and output is restricted. Seldom can more than one line of data be collected in an input buffer before it is processed. Data collected in an output buffer must be displayed before a read request is made to the terminal.

3. The file position pointer and end-of-file test have relatively little significance. An interactive file has no position, in the sense described earlier, and it has no end because the programmer may continue to enter data indefinitely. A special control character may be used by the programmer to signal the end of a portion of the input from the terminal, but the usual notions of end-of-file test and end-of-file processing are often inappropriate.

Because of these substantial differences between interactive files and ordinary sequential files, many language designs have experienced difficulty accommodating interactive files within the input-output structure designed for ordinary sequential files.

Direct Access Files

In a sequential file, the components must be accessed in sequence in the order in which they appear in the file. Although limited operations to advance or backspace the file-position pointer are usually available, access to any component at random is usually not possible. A *direct-access file* is organized so that any single component may be accessed at random, just as in an array or a record. The subscript used to select a component is called its *key* and may be an integer or other identifier. If an integer, the key looks much like an ordinary subscript used to designate a component of the file. However, because a direct-access file is stored on a secondary storage device rather than in primary memory, the implementation of the file and the selection operation is quite different from that for an array.

A direct-access file is organized as an unordered set of components, with a key value associated with each component. Initially the file is empty. A write operation is given a component to copy into the file and the key value to be associated with that component. The write operation creates a new component on the external storage device and copies the designated value into it. The key value is ordinarily associated with the location of the component (on the external storage device) by storing the pair (key, location) in an index. The index is a vector of such pairs. Each write operation that is given a component with a new key value adds another pair to the index. However, if a write operation is given the key of an existing component, that component is overwritten with the new value. Thus writing to a direct-access file is similar to assignment to a component of an array or a record, where the key value is the subscript. A read operation is given the key of the desired component in the file. The index is searched to find the pair with that key and then the component is read from the designated location in secondary storage.

the keyboard, usually beginning with display of a prompt character on the screen. In this setting, several aspects of the ordinary view of sequential files described earlier are modified:

1. The file must be in both read and write mode at the same time because ordinarily read and write operations alternate. First some data are displayed, then some input data are requested, and so on.

2. Buffering of data on input and output is restricted. Seldom can more than one line of data be collected in an input buffer before it is processed. Data collected in an output buffer must be displayed before a read request is made to the terminal.

3. The file-position pointer and end-of-file test have relatively little significance. An interactive file has no position, in the sense described earlier, and it has no end because the programmer may continue to enter data indefinitely. A special control character may be used by the programmer to signal the end of a portion of the input from the terminal, but the usual notions of end-of-file test and end-of-file processing are often inappropriate.

Because of these substantial differences between interactive files and ordinary sequential files, many language designs have experienced difficulty accommodating interactive files within an input–output structure designed for ordinary sequential files.

Direct-Access Files

In a sequential file, the components must be accessed in sequence in the order in which they appear on the file. Although limited operations to advance or backspace the file-position pointer are usually available, access to any component at random is usually not possible. A *direct-access file* is organized so that any single component may be accessed at random, just as in an array or a record. The subscript used to select a component is called its *key* and may be an integer or other identifier. If an integer, the key looks much like an ordinary subscript used to designate a component of the file. However, because a direct-access file is stored on a secondary storage device rather than in central memory, the implementation of the file and the selection operation is quite different from that for an array.

A direct-access file is organized as an unordered set of components, with a key value associated with each component. Initially the file is empty. A *write* operation is given a component to copy into the file and the key value to be associated with that component. The *write* operation creates a new component on the external storage device and copies the designated value into it. The key value is ordinarily associated with the location of the component (on the external storage device) by storing the pair (key, location) in an index. An *index* is a vector of such pairs. Each write operation that writes a component with a new key value adds another pair to the index. However, if a *write* operation is given the key of an existing component, that component is overwritten with the new value. Thus, writing on a direct-access file is similar to assignment to a component of a vector, where the key value is the subscript. A *read* operation is given the key of the desired component in the file. The index is searched to find the pair with that key, and then the component is read from the designated location in secondary storage.

Indexed Sequential Files

An indexed sequential file is similar to a direct-access file, with the additional facility to access components in sequence, beginning from a component selected at random. For example, if component with key value 27 is selected (read), then the subsequent *read* operation may choose the next component in sequence, rather than giving a key value. This file organization provides a compromise between the pure sequential and pure direct-access organizations.

An indexed sequential file requires an index of key values, just as for a direct-access file, but the entries in the index must be *ordered* by key values. When a *read* or *write* operation selects a component with a particular key value, that pair in the index becomes the *current component* of the file (e.g., the file-position pointer is positioned at that component). To advance to the next file component in sequence, the next entry in the index is accessed, and that entry becomes the current component. Thus sequential access to components is possible without a major change from the direct-access organization.

5.4 FORTRAN OVERVIEW

History. FORTRAN is one of the first and still widely used language for scientific and engineering computation. It has undergone much evolution in its 40-year life, has been deemed obsolete and irrelevant numerous times, and yet is still with us and still evolving.

FORTRAN was the first high-level programming language to become widely used. It was first developed in 1957 by IBM for execution on the IBM 704 computer. At that time, the utility of any high-level language was open to question by programmers schooled in assembly language programming. Their most serious complaint concerned the efficiency of execution of code compiled from high-level language programs. As a result, the design of the earliest versions of FORTRAN was oriented heavily toward providing execution efficiency. The success of this early FORTRAN and its dependence on features oriented toward efficient execution on the IBM 704 computer have been a problem with the language ever since. The first standard definition of the language was