

Part II

Describing Computation

Chapter 5

Primitive Types

Overview

This chapter explains the concept of types within programming languages and the hardware that supports these types. Computer memory is an array of bits usually grouped into addressable 8-bit segments called *bytes*. *Words* are groups of bytes, usually 2, 4, and sometimes 8 bytes long. All data types in programming languages must be mapped onto the bytes and words of the machine. Logical computer instructions operate on bytes and words, but other instructions operate on objects that are represented by codes which are superimposed on bit strings. Common codes include ASCII, EBCDIC, binary integer, packed decimal and floating point.

A data type is an abstraction: a description of a set of properties independent of any specific object that has those properties. A previously defined type is referred to by a type name. A type description identifies the parts of a nonprimitive type. A specific type is a homogeneous set of objects, while a generic type is a set that includes objects of more than one specific type. Each type is a set of objects with an associated set of functions. A type defines the representation for program objects. Several attributes are defined by the type of an object, including encoding, size, and structure.

Every language supports a set of primitive data types. Usually these include integer, real, Boolean, and character or string. A language standard determines the minimum set of primitive types that the language compiler must implement. Hardware characteristics influence which types a language designer chooses to make primitive. If the hardware does not support a required type, that type may have to be emulated, that is, implemented in the software.

Type declarations have a long history, going back to the earliest languages which sup-

ported primitive types built into the instruction set of the host machine. By the late 1960s, types were recognized as abstractions. Type declarations defined value constructors, selectors, and implicit type predicates. In the 1970s, with the emergence of *Ada*, types were treated as objects in a limited way. There were cleaner type compatibility rules, support for type portability, and explicit constraint on the values of a type. Recent research includes the issues of type hierarchies with inherited properties and the implementation of nonhomogeneous types in a semantically sound way.

5.1 Primitive Hardware Types

A translator must map a programmer's objects and operations onto the storage and instructions provided by the computer hardware. To understand the primitive types supported by languages, one should also understand the hardware behind those types.

5.1.1 Bytes, Words, and Long Words

Computer memory is a very long array of bits, normally organized into groups.¹ Each group has an address, used to store and fetch data from it. Modern machines usually have 8-bit bytes and are *byte addressable*. Bytes are grouped into longer 2- and 4-byte units called *words* and *long words*. Some machines have a few hardware instructions that support *double word*, or 8-byte, operations.

Bytes and words form the basis for all representation and all computation. They are the primitive data type onto which all other data types must be mapped.

Computer instruction sets include some instructions that operate on raw, uninterpreted bytes or words. These are called *logical* instructions. They include right and left shifts, and bitwise complement, and, or, and exor operations.

Most computer instructions, though, are intended to operate on objects other than bit strings, such as numbers or characters. All objects must be *represented* by bit strings, but they have semantics above and beyond the bits that represent them. These objects are represented by codes that are superimposed on the bit strings. Common encodings include ASCII, EBCDIC, binary integer, packed decimal, and floating point.

5.1.2 Character Codes

Long before IBM built computers, it built *unit record equipment*, which processed data recorded on punched cards. Keypunches were used to produce this data, and line printers could copy cards to fanfold paper. Tabulating machines were used to process the data. These had “plug boards”,

¹The Burroughs memory of the B1700/B1800 series of computers was an undivided string of bits that was actually bit-addressable.

on which a skilled person could build programs by using plug-in cables to connect holes that represented card columns to holes that represented functions such as “+” and “-”.²

Punched cards were in common use before computers were invented and quite naturally became the common input medium for computers. The Hollerith character code, used for punched cards, was adapted for use in computers, and called *Binary Coded Decimal*, or BCD. Hollerith code was a decimal code. It used one column with twelve punch positions to represent each digit. (These positions were interpreted as +, -, 0...9.) Alphabetic letters were represented as pairs of punches, one in the “zone” area (+, -, 0), and one in the “digit” area (1..9). This gives 27 combinations, which is one too many for our alphabet, and the 0:1 punch combination was not used for any letter. The tradition was that this combination was omitted from the alphabet because the two closely spaced punches made it physically weak. However, this combination was used to represent “/”. Thus the alphabet had a nonalpha character in its middle.

The entire Hollerith character set had no more than 64 codes. Letters and digits accounted for 36 of these; the rest were other punctuation and control characters and were represented by double or triple punches. The BCD code used sixty four 6-bit codes to represent this character set.

The BCD character code was reflected in various ways in the computer hardware of the 1950s and early 1960s. It has always been practical to make word size a multiple of the character code size. Hardware was built with word lengths of 24, 36, 48, and 60 bits (making 4, 6, 8, and 10 characters per word). Floating-point encoding was invented for the IBM 704; its 36-bit words were long enough to provide adequate range and precision.

Software, also, showed the effects of this character code. FORTRAN was designed around this severely limited character set (uppercase only, very few available punctuation symbols). FORTRAN identifiers were limited to six characters because that is what would fit into one machine word on the IBM 704 machine. COBOL implemented numeric data input formats that were exactly like Hollerith code. If you wanted to input the number -371, you punched only three columns, and put the “-” sign *over* the rightmost, giving the *number code* “37J”. The number +372 was encoded as “37B”. This wholly archaic code is still in use in COBOL today and is increasingly difficult to explain and justify to students.

Dissatisfaction with 6-bit character codes was rampant; sixty four characters are just not enough. People, reasonably, wanted to use both upper- and lower-case letters, and language designers felt unreasonably restricted by the small set of punctuation and mathematical symbols that BCD provided. Two separate efforts in the early 1960s produced two new codes, EBCDIC (Extended BCD Interchange Code) and ASCII (American Standard Code for Information Interchange).

EBCDIC was produced and championed by IBM. It was an 8-bit code, but many of the 256 possible bit combinations were not assigned any interpretation. Upper- and lowercase characters were included, with ample punctuation and control characters. This code was an extension of BCD; the old BCD characters were mapped into EBCDIC in a systematic way. Certainly, that made compatibility with old equipment less of a problem.

²These were archaic in the early 1960s, but a few early computer science students had the privilege of learning to use them.

Unfortunately, the EBCDIC code was not a “sensible” code because the collating sequence was not normal alphabetical order.³ Numbers were greater than letters, and like BCD, alphabetic characters were intermingled with nonalphabetic characters.

ASCII code grew out of the old teletype code. It uses seven bits, allowing 128 characters. Upper- and lowercase letters, numerals, many mathematical symbols, a variety of useful control characters, and an “escape” are supported. The “escape” could be used to form 2-character codes for added items.⁴ ASCII is a “sensible” code; it follows the well-established English rules for alphabetization. It has now virtually replaced EBCDIC, even on IBM equipment.

An extended 8-bit version of ASCII code is now becoming common. It uses the additional 128 characters for the accented and unlauded European characters, some graphic characters, and several Greek letters and symbols used in mathematics. Hardware intended for the international market supports extended ASCII.

5.1.3 Numbers

We take integers and floating-point numbers for granted, but they are not the only ways, or even the only common and useful ways, to represent data.

In the late 1950s and early 1960s, machines were designed to be either *scientific computers* or *business computers*. The memory of a scientific computer was structured as a sequence of words (commonly 36 bits per word) and its instruction set performed binary arithmetic. Instructions were fixed length and occupied one word of memory.

Packed Decimal

The memory of a business computer was a series of BCD bytes with an extra bit used to mark the beginning of each variable-length “word”. Objects and instructions were variable length. Numbers were represented as a series of decimal (BCD) digits. Arithmetic was done in base ten, not base two.

The distinction between scientific and business computers profoundly affected the design of programming languages. COBOL, a “business language”, was oriented toward variable-length objects and supported base ten arithmetic. In contrast, FORTRAN was a “scientific language”. Its data objects were one word long, or arrays of one-word objects, and computation was done either in binary or in floating point. Characters were not even a supported data type.

In 1964, IBM introduced a family of computers with innovative architecture, intended to serve both the business and scientific communities.⁵ The memory of the IBM 360 was byte-addressable. The hardware had general-purpose registers to manipulate byte, half-word (2-byte), and word (4-byte) sized objects, plus four 8-byte registers for floating-point computation. The instruction

³The *collating sequence* of a code is the order determined by the “<” relationship. To print out a character code in collation order, start with the code 00000000, print it as a character, then add 1 and repeat, until you reach the largest code in the character set.

⁴It is often used for fancy I/O device control codes, such as “reverse video on”.

⁵Gorsline [1986], p. 317.

Exhibit 5.1. Packed-decimal encoding.

- A number is a string of digits. The sign may be omitted if the number is positive, or represented as the first or last field of the string.
 - Each decimal digit is represented by 4 bits, and pairs of digits are packed into each byte. The string may be padded on the left to make the length even.
 - The code “0000” represents the digit 0, and “1001” represents 9. The six remaining possible bit patterns, “1010” ... “1111”, do not represent legal digits.
-

set supported computation on binary integers, floating point, *and* integers represented in packed decimal with a trailing sign [Exhibit 5.1].

Many contemporary machines support packed decimal computation. Although the details of packed-decimal representations vary somewhat from machine to machine, the necessary few instructions are included in the Intel chips (IBM PC), the Motorola chips (Apollo workstations, Macintosh, Atari ST), and the Data General MV machines.

Packed-decimal encoding is usually used to implement *decimal fixed-point arithmetic*. A decimal fixed-point number has two integer fields, one representing the magnitude, the other the scale (the position of the decimal point). The scale factors must be taken into account for every arithmetic operation. For instance, numbers must be adjusted to have the same scale factors before they can be added or subtracted. Languages such as Ada and COBOL, which support fixed-point arithmetic, do this adjustment for the programmer.⁶

Base two arithmetic is convenient and fast for computers, but it cannot represent most base ten fractions exactly. Furthermore, almost all input and output is done using base ten character strings. These strings must be converted to/from binary during input/output. The ASCII to floating point conversion routines are complex and slow.

Arithmetic is slower with packed decimal than with binary integers because packed-decimal arithmetic is inherently more complex. Input and output conversions are much faster; a packed-decimal number consists of the last 4 bits of each ASCII or EBCDIC digit, packed two digits per byte. Arithmetic is done in *fixed point*; a specified number of digits of precision is maintained, and numbers are rounded or truncated after every computation step to the required precision. Control over rounding is easy, and no accuracy is lost in changing the base of fractions.

In a data processing environment, packed decimal is often an ideal representation for numbers. Most business applications do more input and output than computation. Some, such as banking

⁶Unfortunately, the Ada standard does not require that fixed-point declarations be implemented by decimal fixed-point arithmetic! It is permissible in Ada to approximate decimal fixed-point computation using numbers represented in binary, not base ten, encoding!

Exhibit 5.2. Representable values for signed and unsigned integers.

These are the smallest and largest integer values representable on a two's complement machine.

Type	Length	Minimum	Maximum
Signed	4 bytes	-2,147,483,648	2,147,483,647
	2 bytes	-32,768	32,767
	1 byte	-128	127
Unsigned	4 bytes	0	4,294,967,295
	2 bytes	0	65,535
	1 byte	0	255

and insurance computations, require total control of precision and rounding during computation in order to meet legal standards. For these applications, binary encoding for integers and floating-point encoding for reals is simply not appropriate.

Binary Integers

Binary numbers are “built into” modern computers. However, there are several ways that binary numbers can be represented. They can be different lengths (2- and 4-byte lengths being the most common), and be signed or unsigned. If the numbers are signed, the negative values might be represented in several ways.

Unsigned integers are more appropriate than signed numbers for an application that simply does not deal with negative numbers, for example, a variable representing a machine address or a population count. Signed and unsigned numbers of the same length can represent exactly the same number of integers; only the range of representable numbers is different [Exhibit 5.2]. On a modern two's complement machine, unsigned arithmetic is implemented by the same machine instructions as signed arithmetic.

Some languages, for example C, support both signed and unsigned integers as primitive types. Others, for example Pascal and LISP, support only signed integers. Having “unsigned integer” as a primitive type is not usually necessary. Any integer that can be represented as an “unsigned” can also be represented as a “signed” number that is one bit longer. There are only a few situations in which this single bit makes a difference:

- An application where main storage must be conserved and a 1-byte or 2-byte integer could be used, but only if no bit is wasted on the sign.
- An application where very large machine addresses or very large numbers must be represented as integers, and every bit of a long integer is necessary to represent the full range of possible values.

- The intended application area for the language involves extensive use of the natural numbers (as opposed to the integers). By using type “unsigned” we can constrain a value to be nonnegative, thereby increasing the explicitness of the representation and the robustness of the program.

“Unsigned” will probably be included as a primitive type in any language whose intended applications fit one of these descriptions. C was intended for systems programming, in which access to all of a machine’s capabilities is important, and so supports “unsigned” as a primitive type.⁷

Signed Binary Integers The arithmetic instructions of a computer define the encoding used for numbers. The `ADD 1` instruction determines the order of the bit patterns that represent the integers. Most computers “count” in binary, and thus support binary integer encoding. Most compilers use this encoding to represent integers. Although this is not the only way to represent the integers, binary is a straightforward representation that is easy for humans to learn and understand, and it is reasonably cheap and fast to implement in hardware.⁸

Large machines support both word and long word integers; very small ones may only support byte or word sized integers. On such machines, a compiler writer must use the short word instructions to emulate arithmetic on longer numbers that are required by the language standard. For example, the instruction set on the Commodore 64 supported only byte arithmetic, but Pascal translators for the Commodore implemented 2-byte integers. Adding a pair of 2-byte integers required several instructions; each half was added separately and then the carry was propagated.

Negative Numbers One early binary integer representation was *sign and magnitude*. The left-most bit was interpreted as the sign, and the rest of the bits as the magnitude of the number. The representations for +5 and −5 differed only in one bit. This representation is simple and appealing to humans, but not terrific for a computer. An implementation of arithmetic on sign-and-magnitude numbers required a complex circuit to propagate carries during addition, and another one to do borrowing during subtraction.

CPU circuitry has always been costly, and eventually designers realized that it could be made less complex and cheaper by using complement notation for negative numbers. Instead of implementing “+” and “−”, a complement machine could use “+” and “negate”. Subtraction is equivalent to negation followed by addition. Negation is trivially easy in one’s complement representation—just flip the bits. Thus “00000001” represented the integer 1 and “11111110” represented the integer negative one. A carry off the left end of the word was added back in on the right. The biggest drawback of this system is that zero has two representations, “00000000” (or +0) and “11111111” (or −0).

A further insight occurred in the early 1960s: complement arithmetic could be further simplified by using two’s complement instead of one’s complement. To find the two’s complement of a

⁷We must also note that the primitive type “byte” or “bitstring” is lacking in C, and “unsigned” is used instead. While this is semantically unattractive, it works.

⁸Other kinds of codes have better error correction properties or make carrying easier.

Exhibit 5.3. IEEE floating-point formats.

Format Name	Length	Bit fields in representation		
		Sign	Exponent	Mantissa
Short real	4 bytes	31	30–23	22–0, with implicit leading 1
Long real	8 bytes	63	62–52	51–0, with implicit leading 1
Temporary real	10 bytes	79	78–64	63–0, explicit leading 1

number, complement the bits and add one. The two’s complement of “00000001” (representing 1) is “11111111” (representing -1). Two’s complement representation has two good properties that are missing in one’s complement: there is a unique representation for zero, “00000000”, and carries off the left end of a sum can simply be ignored. Two’s complement encoding for integers has now become almost universal.

Floating Point

Many hardware representations of floating-point numbers have been used in computers. Before the advent of ASCII code, when characters were 6 bits long, machine words were often 36 or 48 bits long. (It has always been convenient to design a machine’s word length to be a multiple of the byte length.) Thirty-six bits is enough to store a floating-point number with a good range of exponents and about eight decimal digits of precision. Forty-eight or more bits allows excellent precision. However, word size now is almost always 32 bits, which is a little too small.

In order to gain the maximum accuracy and reasonable uniformity among machines, the IEEE has developed a standard for floating-point representation and computation. In this discussion, we focus primarily on this standard. The IEEE standard covers all aspects of floating point—the use of bits, error control, and processor register requirements. It sets a high standard for quality. Several modern chips, including the IBM 8087 coprocessor, have been modeled after it.

To understand floats, you need to know both the format and the semantics of the representation. A floating-point number, N , has two parts, an exponent, e , and a mantissa, m . Both parts are signed numbers, in some base. If the base of the exponent is b , then $N = m * b^e$.

The IEEE standard supports floats of three lengths: 4, 8, and 10 bytes. Let us number the bits of a float starting with bit 0 on the right end. The standard prescribes the float formats shown in Exhibit 5.3. The third format defines the form of the CPU register to be used during computation. Exhibit 5.4 shows how a few numbers are represented according to this standard.

The sign bit, always at the left end, is the sign of the entire number. A “1” is always used for negative, “0” for a positive number.

The exponent is a signed number, often represented in bias notation. A constant, called the bias, is added to the actual exponent so that all exponent values are represented by unsigned positive numbers. In the case of bias 128, this is like two’s complement with the sign bit reversed.

The advantage of a bias representation is that, if an ordinary logical comparison is made,

Exhibit 5.4. Floating point on the SparcStation.

The SparcStation is a new RISC workstation built by Sun Microsystems. It has a floating-point coprocessor modeled after the IEEE standard. Using the Sun C compiler to explore its floating-point representation, we find that the C “float” type is implemented by a field with the IEEE 4-byte encoding. A few numbers are shown here with their representations printed in hex notation and in binary with the implied 1-bit shown to the left of the mantissa.

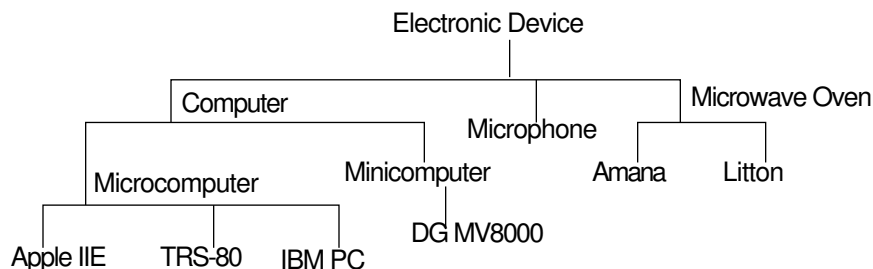
Decimal	Hex	Binary Representation		
		Sign	Exponent	Mantissa
0.00	00000000	0	00000000	0.0000000 00000000 00000000
0.25	3E800000	0	01111101	1.0000000 00000000 00000000
0.50	3F000000	0	01111110	1.0000000 00000000 00000000
1.00	3F800000	0	01111111	1.0000000 00000000 00000000
-1.00	BF800000	1	01111111	1.0000000 00000000 00000000
10.00	41200000	0	10000010	1.0100000 00000000 00000000
5.00	40A00000	0	10000001	1.0100000 00000000 00000000
2.50	40200000	0	10000000	1.0100000 00000000 00000000
1.25	3FA00000	0	01111111	1.0100000 00000000 00000000

positive numbers are greater than negative numbers. Absolutely no special provision needs to be made for the sign of the number. With 8 bits in the exponent, “00000000” represents the smallest possible negative exponent, “11111111” is the largest positive exponent. “10000000” generally represents an exponent of either zero or one. When interpreted as a binary integer, “10000000” is 128. If this represents an exponent of zero, we say that the notation is “bias 128”, because $128 - 0 = 128$. When “10000000” represents an exponent of one, we say the notation is “bias 127”, because $128 - 1 = 127$.

In the IEEE standard, the exponent is represented in bias 127 notation, and the exponent “10000000” represents +1. This can be seen easily in Exhibit 5.4. The representation for 2.50 has an exponent of “10000000”. The binary point in “1.0100000” must be moved one place to the right to arrive at “10.1”, the binary representation of 2.50. Thus “10000000” represents +1.

Floating-point hardware performs float operations in a very long register, much longer than the 24 bits that can be stored in a float. To maintain as many bits of precision as possible, the mantissa is normalized after every operation. This means that the leading “0” bits are shifted to the left until the leftmost bit is a “1”. Then when you store the number, all bits after the twenty-fourth are truncated (discarded). A normalized mantissa always starts with a “1” bit, therefore this bit has no information value and can be regenerated by the hardware when needed. So only bits 2–24 of the mantissa are stored, in bits 22–0 of the float number.

The mantissa is a binary fraction with an implied binary point. In the IEEE standard, the point is between the implied “1” bit and the rest of the mantissa. Some representations place the

Exhibit 5.5. A hierarchy of abstractions.


binary point to the left of the implied “1” bit. These interpretations give the same precision but different ranges of representable numbers.

5.2 Types in Programming Languages

5.2.1 Type Is an Abstraction

An *abstraction* is the description of a property independent from any particular object which has that property. Natural languages contain words that form hierarchies of increasing degrees of abstraction, such as “TRS-80”, “microcomputer”, “computer”, and “electronic device” [Exhibit 5.5]. “TRS-80” is itself an abstraction, like a type, describing a set of real objects, all alike. Most programming language development since the early 1980s has been aimed at increasing the ability to express and use abstractions within a program. This work has included the development of abstract data types, generic functions, and object-oriented programming. We consider these topics briefly here and more extensively later.

A *data type* is an abstraction: it is the common property of a set of similar data objects. This property is used to define a representation for these objects in a program. Objects are said to “have” or to “be of” that type to which they belong. Types can be primitive, defined by the system implementor, or they can be programmer defined. We refer to a previously defined type by using a *type name*. A *type declaration* defines a type name and associates a *type description* with it, which identifies the parts of a nonprimitive type [Exhibit 5.6]. The terms *type* and *data type* are often used loosely; they can refer to the type name, the type description, or the set of objects belonging to the type.

If all objects in a type have the same size, structure, and semantic intent, we call the type *concrete* or *specific*. A specific type is a homogeneous set of objects. All the primitive types in Pascal are specific types, as are Pascal arrays, sets, and ordinary records made out of these basic types. A *variant record* in Pascal is not a specific type, since it contains elements with different structures and meanings.

Exhibit 5.6. Type, type name, and type description.

- Real-world objects: a set of rectangular boxes.
- Type: We will represent a box by three real numbers: its length, width, and depth.
- Type name declared in Pascal: `TYPE box_type =`
- Possible type descriptions in Pascal:

```
ARRAY [1..3] OF real
RECORD length, width, depth: real END
```

A *generic domain* is a set that includes objects of more than one concrete type [Exhibit 5.7]. A specific type that is included in a generic domain is called an *instance* or *species* of the generic domain, as diagrammed in Exhibit 5.8. Chapters 15 and 17 explore the subjects of type abstraction and generic domains.

5.2.2 A Type Provides a Physical Description

The properties of a type are used to map its elements onto the computer's memory. Let us focus on the different attributes that are part of the type of an object. These include encoding, size, and structure.

Exhibit 5.7. Specific types and generic domains.**Specific types:**

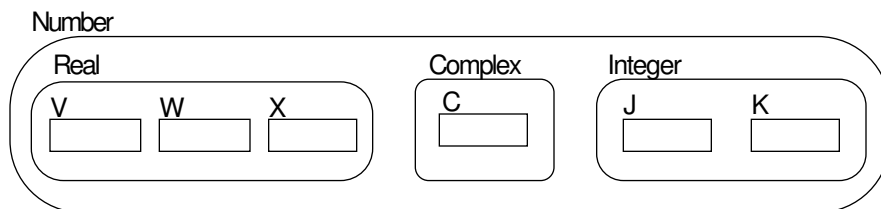
- Integer arrays of length 5
- Character arrays of length 10
- Real numbers
- Integer numbers

Generic domains:

- Intarray: The set of integer arrays, of all lengths.
 - Number: All representations on which you can do arithmetic, including floating point, integer, packed decimal, etc.
-

Exhibit 5.8. Specific types are instances of generic domains.

- The generic domain Number has several specific subtypes, including Real, Integer, and Complex.
- Objects (variables) have been declared that belong to these types. Objects named V, W, and X belong to type Real; objects J and K belong to type Integer, and C belongs to type Complex.
- All six objects also belong to the generic domain Number.



Encoding. The instruction set of each machine includes instructions that do useful things on certain encodings (bit-level formats) of data. For example, the Data General MV8000 has instructions that perform addition if applied to numbers encoded with 4-bits per decimal digit. Because of this “built-in” encoding, numbers can be conveniently represented in packed-decimal encoding in Data General COBOL. Where an encoding must be implemented that is not directly supported by the hardware, the implementation tends to be inefficient.

Size. The size of an object can be described in terms of hardware quantities such as words or bytes, or in terms of something meaningful to the programmer, such as the range of values or the number of significant digits an object may take on.

Structure. An object is either simple or it is compound. A *simple object* has one part with no subparts. No operators exist within a language that permit the programmer to decompose simple objects. In a language that has “integer” as a simple type, integer is generally undecomposable.

In standard Pascal, integers are simple objects, as are reals, Booleans, and characters. In various Pascal extensions, though, an integer can be decomposed into a series of bytes. In these dialects “integer” is not a simple type. Primitive types may or may not be simple. In both cases, “integer” is a primitive type; that is, it is a predefined part of the language.

A compound object is constructed of an ordered series of fields of specific types. A list of these fields describes the structure of the object. If the fields of the compound object all have the same type, it is a homogeneous compound. These are commonly called “array”, “vector”, “matrix”,

or “string”. The dimensions of an array and its base type (the type of its elements) define its structure.

If the fields of a compound object have different types, it is a heterogeneous compound. These are commonly called “records” or “structures”. An ordered list of the types of each field of a record defines its structure.

The distinctions among structure, encoding, and size are seen most clearly in COBOL, where these three properties are specified separately by the programmer.

Structure in COBOL. The internal structure of each data object is defined by listing its fields and subfields, in order. The subfields of a field are listed immediately following the field and given higher level numbers to indicate that they are subordinate to it.

Encoding in COBOL. Character data has only one encoding: the character code built into the machine hardware. Depending on the compiler, several encodings may be provided for numbers, with `DISPLAY` being the default. The COBOL programmer may specify the encoding in a `USAGE` clause. In Data General COBOL, the programmer can choose from the following set:

<code>DISPLAY</code>	ASCII or EBCDIC characters
<code>COMPUTATIONAL</code>	binary fixed point
<code>COMP-2</code>	packed binary-coded-decimal fixed point
<code>COMP-3</code>	floating point

Double-precision encoding is also provided in some COBOL implementations. Each encoding has inherent advantages, which must be understood by the programmer. Input and output require operands of `DISPLAY` usage. Arithmetic can be done on all usages except `DISPLAY`. The most efficient numeric I/O conversion is between `DISPLAY` and `COMP-2`. The most efficient arithmetic is done in `COMPUTATIONAL`.

Conversion from one encoding to another is performed automatically when required in COBOL. If a numeric variable does not have the default usage, `DISPLAY`, conversion is performed during the input and output processes, as in most languages. If a numeric variable represented in `DISPLAY` usage is used in an arithmetic statement, it will be converted to packed decimal. (This conversion is fast and efficient). The arithmetic will be done in packed-decimal encoding, and the result will be converted back to display usage if it is stored in a `DISPLAY` variable.

Size in COBOL. Size is defined by supplying a `PICTURE` clause for every field that has no subfields [Exhibit 5.9]. The `PICTURE` illustrates the largest number of significant characters or decimal digits that will ever be needed to represent the field. Note that the programmer describes the size of the object being represented, not the size, in bytes, of the representation. Different amounts of storage could be allocated for equal size specifications with different encoding specifications.

At the other extreme from COBOL, the language BASIC permits the programmer to specify only whether the object will encode numeric or alphanumeric objects, and to declare the structure of arrays (number of dimensions and size of each). The encoding is chosen by the translator and hidden from the user. Thus BASIC is simpler to use. It frees the programmer from concern about the appropriateness of the encoding. At the same time, it provides no easy or efficient control over

Exhibit 5.9. Size and encoding specifications in COBOL.

Three simple variables are defined, named PRICE, DISCOUNT, and ITEM.

```
01 PRICE      PICTURE 999V99.  
01 DISCOUNT PICTURE V999  USAGE COMPUTATIONAL.  
01 ITEM       PICTURE XXXX.
```

PRICE has a numeric-character encoding, indicated by the 9s in the PICTURE clause and the absence of a USAGE clause. The size of this variable is defined by the number of 9s given, and decimal position is marked by the “V”. In this case, the number has two decimal places and is less than or equal to 999.99.

DISCOUNT has binary fixed-point encoding (because of the USAGE clause). Its size is three decimal digits, with a leading decimal point.

ITEM has alphanumeric encoding, indicated by the Xs in its PICTURE. Its size is four characters. Any alphanumeric value of four or fewer characters can be stored in this variable.

precision and rounding. BASIC is thus a better tool for the beginner, but a clumsy tool for the professional.

5.2.3 What Primitive Types Should a Language Support?

The usual set of primitive data types in a language includes integer, real, Boolean, and character or string. However, Ada has many more and BASIC has fewer.

A language standard determines the minimum set of primitive types that must be implemented by a compiler. Choosing this set is the job of the language designer. A language implementor may choose to support additional types, however. For example, Turbo Pascal supports a type `string` that is not required by the standard. The `string` type is a language extension.

The decision to make a type “primitive” in a computer language is motivated by hardware characteristics and the intended uses of the language. Compromises must often be made. A language designer must decide to include or exclude a type from the primitive category by considering the cost of implementing and using it [Exhibit 5.10] as opposed to the cost of not implementing it [Exhibit 5.11].

Types that are not primitive sometimes cannot be implemented efficiently, or even implemented at all, by the user. For example, the ANSI C standard does not support packed-decimal numbers. A user *could* write his or her own packed decimal routines in C. To achieve adequate precision the user would probably map them onto integers, not floats. Masking, base 10 addition and multiplication, carrying, and the like could be implemented. However, the lack of efficiency in the finished product would be distressing, especially when you consider that many machines provide efficient hardware instructions to do this operation.

If users are expected to need a certain type frequently, the language is improved by making that type primitive. Packed decimal is not a primitive type in C because the intended usage of C

Exhibit 5.10. Costs of implementing a primitive type.

- Every added feature complicates both the language syntax and semantics. Both require added documentation. If every useful feature were supported, the language would become immense and unwieldy.
 - Standardization could become more difficult, as there is one more item about which committee members could disagree. This could be an especially severe problem if a type is complex, its primitive operations are extensive, or it is unclear what the ideal representation should be.
 - The compiler and/or library and/or run-time system become more complex, harder to debug, and consume more memory. Literals, input and output routines, and basic functions must be defined for every new primitive type.
 - If typical hardware does not provide instructions to handle the type, it may be costly and inefficient to implement it. Perhaps programmers should not be encouraged to use inefficient types.
-

Exhibit 5.11. Costs of omitting a primitive type.

- Inefficiency: failing to include an operation that is supported by the hardware leads to a huge increase in execution time.
 - Language structure may be inadequate to support the type as a user extension, as Pascal cannot support variable-length strings or bit fields with bitwise operators.
 - Some built-in functions such as `READ`, `WRITE`, assignment, and comparison are generic in nature. They work on all primitive types but not necessarily on all user-defined types. If these functions cannot be extended, a user type can never be as convenient or easy to use as a primitive type.
 - Primitive types have primitive syntax for writing literals. Literal syntax is often not extensible to user-defined types.
-

was for systems programming, not business applications. In this case, the cost of *not* implementing the type is low, and the cost of *implementing* it is increased clutter in the language.

As another example, consider the `string` type in Pascal. It was almost certainly a mistake to omit a string manipulation package from the standard language. Alphabetic data is very common, and many programs use string data. The Pascal standard recognizes that strings exist but does not provide a reasonable set of string manipulation primitives. The standard defines a “string” to be any object that is declared as a “packed array[1..n] of char”, where `n` is an integer > 1 . String output is provided by `Write` and `WriteLn`. String comparison and assignment are supported, but only for strings of equal length. Length adjustment, concatenation, and substrings are not supported, and `Read` cannot handle strings at all. A programmer using Standard Pascal must read alphabetic fields one character at a time and store each character into a character array.

Virtually all implementations of Pascal extend the language to include a full string type with reasonable operations. Unfortunately, these extensions have minor differences and are incompatible with each other. Thus there are two kinds of costs associated with omitting strings from standard Pascal:

1. User implementations of string functions are required. These execute less efficiently than system implementations could.
2. Because programmers use strings all the time, many compilers are extended to support a string type and some string functions. Using these extensions makes programs less portable because the details of the extensions vary from compiler to compiler.

Including strings in the language makes a language more complex. Both the syntax and semantic definitions become longer and require more extensive documentation. The minimal compiler implementation is bigger. In the case of Pascal and strings, none of these reasons justify the omission.

When language designers do decide to include a primitive type, they must extend the language syntax for declarations, but they have some choices about how to include the operations on that type. The meanings of operators such as “`<`” are usually extended to operate on elements of the new type. New operators may also be added. Any specific function for the new type may be omitted, added to the language core, or included in a library. The latter approach becomes more and more attractive as the number of different primitive types and functions increases. A modular design makes the language core simpler and smaller, and the library features do not add complexity or consume space unless they are needed.

For example, exponentiation is a primitive operation that is important for much scientific computation. Pascal, C, and FORTRAN all support floating-point encoding but have very unequal support for exponentiation. In Pascal, exponentiation in base 10 is not supported by the standard at all; it must be programmed using the natural logarithm and exponentiation functions (“`ln`” and “`exp`”). In C, an exponentiation function, “`pow`”, is included in the mathematics library along with the trigonometric functions. In contrast, FORTRAN’s intended application was scientific com-

putation, and the FORTRAN language includes an exponentiation *operator*, “**”, as part of the language core.

5.2.4 Emulation

The types required by a language definition may or may not be supported by the hardware of machines for which that language is implemented. For example, Pascal requires the type “real”, but floating-point hardware is not included on many personal computers. In such situations, data structures and operations for that type must be implemented in software. Another example: fixed-point arithmetic is part of Ada. This is no problem on hardware that supports packed-decimal encoding, but on a strictly binary machine, an Ada translator must use a software emulation or approximation of fixed-point arithmetic.

The representation for an emulated primitive type is a compromise. On the one hand, it should be as efficient as possible for the architecture of the machine. On the other hand, it should conform as closely as possible to the typical hardware implementation so that programs are portable. The hardware version and the emulation should give the same answers!

When floating point is emulated, the exponent is sometimes represented as a 1-byte integer, and the mantissa is represented by 4 or more bytes with an implied binary point at the left end. This produces an easily manipulated object with good precision. A minimum of shifting and masking is needed when this representation is used. However, it sometimes does not produce the same answers as a 4-byte hardware implementation.

Other software emulations try to conform more closely to the hardware. Accurate emulation of floating-point hardware is more difficult and slower, but has the advantage that a program will give the same answers with or without a coprocessor. A good software emulation should try to imitate the IEEE hardware standard as closely as possible without sacrificing acceptable efficiency [Exhibit 5.12].

5.3 A Brief History of Type Declarations

The ways for combining individual data items into structured aggregates form an important part of the semantic basis of any language.

5.3.1 Origins of Type Ideas

Types Were Based on the Hardware. The primitive types supported by the earliest languages were the ones built into the instruction set of the host machine. Some aggregates of these types were also supported; the kinds of aggregates differed from language to language, depending on both the underlying hardware and the intended application area. In these old languages, there was an intimate connection between the hardware and the language.

For example, FORTRAN, designed for numeric computation, was first implemented on the IBM 704. This was the first machine to support floating-point arithmetic. So FORTRAN supported one-

Exhibit 5.12. An emulation of floating point.

This is a brief description of the software emulation of floating point used by the Mark Williams C compiler for the Atari ST (Motorola 68000 chip). Note that it is very similar to, but not quite like, the IEEE standard shown in Exhibits 5.3 and 5.4.

Bit 31: Sign
 Bits 30–23: Characteristic, base 2, bias 128
 Bits 22–0: Normalized base 2 mantissa, implied high-order “1”, binary point immediately to the left of the implied “1”.

Decimal	Hex	Binary Representation		
		Sign	Exponent	Mantissa
0.00	00000000	0	00000000	.00000000 00000000 00000000
0.25	3F800000	0	01111111	.10000000 00000000 00000000
0.50	40000000	0	10000000	.10000000 00000000 00000000
1.00	40800000	0	10000001	.10000000 00000000 00000000
-1.00	C0800000	1	10000001	.10000000 00000000 00000000
10.00	42200000	0	10000100	.10100000 00000000 00000000
5.00	41A00000	0	10000011	.10100000 00000000 00000000
2.50	41200000	0	10000010	.10100000 00000000 00000000
1.25	40A00000	0	10000001	.10100000 00000000 00000000

word representations of integers and floating-point numbers. The 704 hardware had index registers that were used for accessing elements of an array—so FORTRAN supported arrays.

COBOL was used to process business transactions and was implemented on byte-oriented “business” machines. It supported aggregate variables in the form of records and tables, represented as variable-length strings of characters. One could read or write entire COBOL records. This corresponded directly to the hardware operation of reading or writing one tape record. One could extract a field of a record. This corresponded to a hardware-level “load register from memory” instruction. The capabilities of the language were the capabilities of the underlying hardware.

“Type” was not a separate idea in COBOL. A structured variable was not an example of a structured type—it was an independent object, not related to other objects. The structured variable as a whole was named, as were all of its fields, subfields, and sub-subfields. To refer to a subfield, the programmer did not need to start with the name of the whole object and give the complete pathname to that subfield; it could be referred to directly if its name was unambiguous.

FORTRAN supported arrays, and COBOL supported both arrays (called “tables”) and records. It would be wrong, though, to say that they supported array or record *types*, because the structure of these aggregates was not abstracted from the individual examples of that structure. One could *use* a record in COBOL, and even pass it to a subroutine, but one could not talk about the type of

Exhibit 5.13. Declaration and use of a record in COBOL.

We declare a three-level record to store data about a father. If other variables were needed to store information about other family members, lines two through six would have to be repeated. COBOL provides no way to create a set of uniformly structured variables. Field names could be the same or different for a second family member. The prevailing style is to make them different by using a prefix, as in F-FIRST below.

```

1  FATHER.
2  NAME.
   3  LAST          PIC X(20).
   3  F-FIRST      PIC X(20).
   3  F-MID-INIT   PIC X.
2  F-AGE          PIC 99.
```

Assume that FATHER is the only variable with a field named F-FIRST, and that MOTHER also has a field named LAST. Then we could store information in FATHER thus:

```
MOVE 'Charles' TO F-FIRST. MOVE 'Brown' TO LAST IN FATHER.
```

Note that the second line gives just enough information to unambiguously identify the field desired; it does not specify a full pathname.

that record. Each record object had a structure, but that structure had no name and no existence apart from the object [Exhibit 5.13].

LISP Introduced Type Predicates. LISP was the earliest high-level language to support dynamic storage allocation, and it pioneered garbage collection as a storage management technique. In the original implementation of LISP, its primitive types, atom and list, were drawn directly from the machine hardware of the IBM 704. An “atom” was a number or an identifier. A “list” was a pointer to either an atom or a cell. A “cell” was a pair of lists, implemented by a single machine word. The 36-bit machine instruction word had four fields: operation code, address, index, and decrement. The address and decrement fields could both contain a machine address, and the hardware instruction set included instructions to fetch and store these fields.

Here again we see a close relationship between the language and the underlying hardware. This two-address machine word was used to build the two-pointer LISP cell. The three fundamental LISP functions, CAR, CDR, and CONS, were based directly on the hardware structure. CAR extracted the address field of the cell, and CDR extracted the decrement field. (Note that the “A” in CAR and the “D” in CDR came from “address” and “decrement”.) CONS constructed a cell dynamically and returned a pointer to it. This cell was initialized to point at the two arguments of CONS.

Note that all LISP allocations were a fixed size—one word. Only one word was ever allocated at a time. However, the two pointers in a cell could be used to link cells together into tree structures

of indefinite size and shape.

The concept of “type” was more fully developed in LISP than in FORTRAN or COBOL. Types were recognized as qualities that could exist separately from objects, and LISP supported *type predicates*, functions that could test the type of an argument at run time. Predicates were provided for the types “atom” and “list”. These were essential for processing tree structures whose size and shape could vary dynamically.

SNOBOL: Definable “Patterns”. SNOBOL was another language of the early 1960s. It was designed for text processing and was the first high-level language that had dynamically allocated *strings* as a primitive type. This was an important step forward, since strings (unlike arrays, records, and list cells) are inherently variable-sized objects.

New storage management techniques had to be developed to handle variable-sized objects. Variables were implemented as pointers to numbers or strings, which were stored in dynamically allocated space. Dynamic binding, not assignment, was used to associate a value with a variable. Storage objects were created to hold the results of computations and bound to an identifier. They died when that identifier was reused for the result of another computation. A storage compaction technique was needed to reclaim dead storage objects periodically. The simplest such technique is called *storage compaction*. It involves identifying all live storage objects and moving them to one end of memory. The rest of the memory then becomes available for reuse.

A second new data type was introduced by SNOBOL: the *pattern*. Patterns were the first primitive data type that did not correspond at all to the computer hardware. A pattern is a string of characters interspersed with “wild cards” and function calls. The language included a highly powerful pattern matching operation that would compare a string to a pattern and identify a substring that matched the pattern. During the matching process, the wild cards would be matched against first one substring and then another, until the entire pattern matched or the string was exhausted.⁹

COBOL permitted the programmer to define objects with complex structured data types but not to refer to those types. LISP provided type predicates but restricted the user to a few primitive types. PL/1 went further than either: its “LIKE” attribute permitted the programmer to refer to a complex user-defined type.

PL/1 was developed in the mid-1960s for the IBM 360 series of machines. It was intended to be the “universal” language that would satisfy the needs of both business and scientific communities. For this reason, features of other popular languages were merged into one large, conglomerate design. Arithmetic expressions resembled FORTRAN, and pointers permitted the programmer to construct dynamically changing tree structures. Declarations for records and arrays were very much like COBOL declarations.

Types could not be declared separately but were created as a side effect of declaring a structured variable. Once a type was created, though, more objects of the same type could be declared by saying they were LIKE the first object [Exhibit 5.14].

⁹Compare this to the pattern matching built into Prolog, Chapter 10, Section 10.4.

Exhibit 5.14. Using the LIKE attribute in PL/1.

The design of PL/1 was strongly influenced by COBOL. This influence is most obvious in the declaration and handling of structures. Here we declare a record like the one in Exhibit 5.13. We go beyond the capabilities of COBOL, though, by declaring a second variable, `MOTHER`, of the same structured type.

```
DCL 1  FATHER,
      2  NAME,
          3  LAST      CHAR (20),
          3  FIRST     CHAR (20),
          3  MID-INIT  CHAR (1),
      2  F-AGE        PIC '99';
DCL 1  MOTHER LIKE FATHER;
```

To create unambiguous references, field names of both `MOTHER` and `FATHER` must be qualified by using the variable name:

```
MOTHER.LAST = FATHER.LAST;
```

5.3.2 Type Becomes a Definable Abstraction

By the late 1960s, types were recognized as abstractions—things that could exist apart from any instances or objects. The fundamental idea, developed by C. Strachey and T. Standish, is that a *type* is a set of constructors (to create instances), selectors (to extract parts of a structured type), and a predicate (to test type identity). Languages began to provide ways to define, name, and use types to create homogeneous sets of objects. `ALGOL-68` and `Simula` were developed during these years.

`Simula` pioneered the idea that a type definition could be grouped together with the functions that operate on that type, and objects belonging to the type, to form a “class”. Thus `Simula` was the first language to support type modules and was a forerunner of the modern object-oriented languages.¹⁰

`ALGOL-68` contained type declarations and very carefully designed type compatibility rules. The type declarations defined constructors (specifications by which structured variables could be allocated), selectors (subscripts for arrays and part names for records), and implicit type predicates. Type identity was the basis for extensive and carefully designed type checking and compatibility rules. Some kinds of type conversions were recognized to be (usually) semantically valid, and so were supported. Other type relationships were seen as invalid. The definition of the language was immensely complex, partly because of the type extension and compatibility rules, and partly because the design goal was super-generality and power.

¹⁰See Chapter 17 for a discussion of object-oriented languages.

Reactions to Complexity: C and Pascal

Two languages, C and Pascal, were developed at this time as reactions against the overwhelming size and complexity of PL/1 and ALGOL-68. These were designed to achieve the maximum amount of power with the minimum amount of complexity.

C moved backwards with respect to type abstractions. The designers valued simplicity and flexibility of the language more than its ability to support semantic validity. They adopted type declarations as a way to define classes of structured objects but omitted almost all use of types to control semantics. C supported record types (`structs` and `unions`) with part selectors and arrays with subscripting. Record types were full abstractions; they could be named, and the names used to create instances, declare parameters, and select subfields. Arrays, however, were not fully abstracted, independent types; array types could not be named and did not have an identity that was distinct from the type of the array elements.

The purpose of type declarations in C was to define the constructors and selectors for a new type. The declaration supplied information to the compiler that enabled it to allocate and access compound objects efficiently. The field names in a record were translated into offsets from the beginning of the object. The size of the base type of an array became a multiplier, to be applied to subscripts, producing an offset. At run time, when the program selected a field of the compound, the offset was added to the address of the beginning of the compound, giving an effective address.

At this period of history, types were not generally used as vehicles for expressing semantic intent. Except for computing address offsets, there were very few contexts in early C in which the type of an object made a difference in the code the translator generated.¹¹ Type checking was minimal or nonexistent. Thus C type declarations did not define type predicates. Type identity was not, in general, important. The programmer could not test it directly, as was possible in LISP, nor was it checked by the compiler before performing function calls, as it is in Pascal.¹²

Niklaus Wirth, who participated in the ALGOL-68 committee for some time, designed Pascal to prove that a language *could* be simple, powerful, and semantically sound at the same time.¹³ Pascal retained both the type declarations and type checking rules of ALGOL-68 and achieved simplicity by omitting ALGOL-68's extensive type conversion rules. The resulting language is more restrictive than C, but it is far easier to understand and less error prone.

Ada: The Last ALGOL-Like Language?

In the late 1960s, the U.S. Department of Defense (DoD) realized that the lack of a common computer language among its installations was becoming a major problem. By 1968, small-scale research efforts were being funded to develop a core language that could be extended in various directions to meet the needs of different DoD groups. Design goals included generality of the core language, extensibility, and reasonable efficiency.

¹¹The exception was automatic conversions between numeric types in mixed expressions.

¹²Type checking and the semantic uses of types are discussed at length in Chapter 15.

¹³It is said that he never dreamed that Pascal would achieve such widespread use as a teaching language.

In the early 1970s, DoD decided to strictly limit the number of languages in use and to begin design of one common language. A set of requirements for this new language were developed by analyzing the needs of various DoD groups using computers. Finalized in 1976, these requirements specified that the new language must support modern software engineering methods, provide superior error checking, and support real-time applications. After careful consideration, it was decided that no existing language met these criteria.

Proposals were sought in 1977 for an ALGOL-like language design that would support reliable, maintainable, and efficient programs. Four proposals were selected, from seventeen submitted, for further development. One of these prototype languages was selected in 1979 and named *Ada*. Major changes were made, and a proposed language standard was published in 1980.

Ada took several major steps forward in the area of data types. These included

- Cleaner type compatibility rules.
- Explicit constraints on the values of a type.
- Support for type portability.
- Types treated as objects, in a limited way.

Ada was based on *Pascal* and has similar type compatibility rules. These rules are an important aid to achieving reliable programs. However, *Pascal* is an old language, and its compatibility rules have “holes”; some things are compatible that, intuitively, should not be. *Ada* partially rectified these problems.

The idea of explicit *constraints* on the values belonging to a type was present in *Pascal* in the subrange types. In *Ada*, this idea is generalized; more kinds of constraints may be explicitly stated. These constraints are automatically checked at run time when a value is stored in a constrained variable.¹⁴

In the older languages, the range of values belonging to a type often depended on the hardware on which a program ran. A program, debugged on one computer, often ran incorrectly on another. By providing a means to specify constraints, *Ada* lets the programmer explicitly state data characteristics so that appropriate-sized storage objects may be created regardless of the default data type sizes on any given machine. A programmer can increase the portability of code substantially by using constrained types.

A *Pascal* type can be used to *declare* a parameter, but it cannot *be* a parameter. *Ada* carries the abstraction of types one step further. *Ada* supports modules called *generic packages*. These are collections of declarations for types, data, and functions which depend on type parameters and/or integer parameters. Each type declaration in a generic package defines a *generic type*, or a family of types, and must be *instantiated*, or expanded with specific parameters, to produce a specific type declaration during the first phase of compilation.¹⁵ Thus although the use of type as parameters is restricted to precompile time, *Ada* types are *objects* in a restricted sense.

¹⁴To achieve efficiency, *Ada* permits this checking to be “turned off” after a program is considered fully debugged.

¹⁵See Chapter 17.

Recent Developments

Since the early 1980s, data type research has been directed toward implementing abstract data types, type hierarchies with inherited properties, and implementing nonhomogeneous types in a semantically sound way. These issues are covered in Chapter 17.

Exercises

1. Define: bit, byte, word, long word, double word.
2. What is unique about logical instructions?
3. How are objects represented in the computer? Explain.
4. What is the purpose of computer codes? Explain.
5. What were the dissatisfactions with early computer codes?
6. What was the difference between the memory of a business and a scientific computer?
7. What is packed decimal? How is it used?
8. What is an unsigned number? How is it represented in memory?
9. What is sign and magnitude representation? One's complement? Two's complement?
10. How are negative numbers represented in modern computers?
11. What is a floating-point number? What are the problems associated with the representation of floating-point numbers?
12. Name the computer that you use. For each number below, give the representation (in binary) that is used on your computer.
 - a. The largest 32-bit integer
 - b. The largest positive floating-point number
 - c. Negative one-half, in floating-point
 - d. The smallest positive float (closest to zero)
 - e. The negative floating-point number with the greatest magnitude
13. Even though FORTH does not contain semantic mechanisms that implement subscripting, array bounds, or variables with multiple "slots", it can be said that FORTH "has" arrays. Explain.
14. What is a data type? Type declaration? Type description?

15. Compare the way that the type of an object is represented in Pascal and in APL. Point out similarities and differences.
16. What is a specific data type? Generic data type?
17. Explain the three attributes of a data type: encoding, size, and structure.
18. What determines the set of primitive data types associated with a language?
19. What is the usual set of primitive types associated with a language?
20. What is type emulation? Why is it needed?
21. How were types supported by the earliest languages? Give a specific example.
22. How is type represented in COBOL? LISP? SNOBOL?
23. Compare the pattern matching in SNOBOL to the database search in Prolog.
24. What is type checking? Type compatibility?
25. What are value constructors? Selectors?
26. What new ideas did Simula pioneer?
27. Why was Ada developed?
28. What major steps in the area of data typing were used in Ada?