

Chapter 2

Representation and Abstraction

Overview

This chapter presents the concept of how real-world objects, actions, and changes in the state of a process are represented through a programming language on a computer. Programs can be viewed as either a set of instructions for the computer to execute or as a model of some real-world process. Languages designed to support these views will exhibit different properties. The language designer must establish a set of goals for the language and then examine them for consistency, importance, and restrictions. Principles for evaluating language design are presented. Classification of languages into groups is by no means an easy task. Categories for classifying languages are discussed.

Representation may be explicit or implicit, coherent or diffused.

2.1 What Is a Program?

We can view a program two ways.

1. *A program is a description of a set of actions that we want a computer to carry out.* The actions are the primitive operations of some real or abstract machine, and they are performed using the primitive parts of a machine. Primitive actions include such things as copying data from one machine register or a memory location to another, applying an operation to a register, or activating an input or output device.

2. *A program is a model of some process in the real or mathematical world.* The programmer must set up a correspondence between symbols in the program and real-world objects, and between program functions and real-world processes. Executing a function represents a change in the state of the world or finding a solution to a set of specifications about elements of that world.

These two world-views are analogous to the way a builder and an architect view a house. The builder is concerned with the method for achieving a finished house. It should be built efficiently and the result should be structurally sound. The architect is concerned with the overall function and form of the house. It should carry out the architect's concepts and meet the client's needs.

The two world-views lead to very different conclusions about the properties that a programming language should have. A language supporting world-view (1) provides ready access to every part of the computer so that the programmer can prescribe in detail *how* the computer should go about solving a given problem. The language of a builder contains words for each material and construction method used. Similarly, a program construction language allows one to talk directly about hardware registers, memory, data movement, I/O devices, and so forth. The distinction isn't simply whether the language is "low-level" or "high-level", for assembly language and C are both designed with the builder in mind. Assembly language is, by definition, low-level, and C is not, since it includes control structures, type definitions, support for modules, and the like. However, C permits (and forces) a programmer to work with and be aware of the raw elements of the host computer.

A language supporting world-view (2) must be able to deal with abstractions and provide a means for expressing a model of the real-world objects and processes. An architect deals with abstract concepts such as space, form, light, and functionality, and with more concrete units such as walls and windows. Blueprints, drawn using a formal symbolic language, are used to represent and communicate the plan. The builder understands the language of blueprints and chooses appropriate methods to implement them.

The languages **Smalltalk** and **Prolog** were designed to permit the programmer to represent and communicate a world-model easily. They free the programmer of concerns about the machine and let him or her deal instead with abstract concepts. In **Smalltalk** the programmer defines classes of objects and the processes relevant to these classes. If an abstract process is relevant to several classes, the programmer can define how it is to be accomplished for each. In **Prolog** the programmer represents the world using formulas of mathematical logic. In other languages, the programmer may use procedures, type declarations, structured loops, and block structure. to represent and describe the application. Writing a program becomes a process of representing objects, actions, and changes in the state of the process being modeled [Exhibit 2.1].

The advantage of a "builder's" language is that it permits the construction of efficient software that makes effective use of the computer on which it runs. A disadvantage is that programs tailored to a particular machine cannot be expected to be well suited to another machine and hence they are not particularly portable.

Exhibit 2.1. Modeling a charge account and relevant processes.

Objects: A program that does the accounting for a company's charge accounts must contain representations for several kinds of real-world objects: accounts, payments, the current balance, items charged, items returned, interest.

Actions: Each action to be represented involves objects from a specified class or classes. The actions to be represented here include the following:

- Credit a payment to an account.
- Send a bill to the account owner.
- Debit a purchase to an account.
- Credit a return to an account.
- Compute and debit the monthly interest due.

Changes of state: The current balance of an account, today's date, and the monthly payment date for that account encode the *state* of the account. The balance may be positive, negative, or zero, and a positive balance may be either ok or overdue. Purchases, returns, payments, and monthly due dates and interest dates all cause a change in the state of the account.

Moreover, a programmer using such a language is forced to organize ideas at a burdensome level of detail. Just as a builder must be concerned with numerous details such as building codes, lumber dimensions, proper nailing patterns, and so forth, the program builder likewise deals with storage allocation, byte alignment, calling sequences, word sizes, and other details which, while important to the finished product, are largely unrelated to its form and function.

By way of contrast, an "architect's" language frees one from concern about the underlying machine and allows one to describe a process at a greater level of abstraction, omitting the minute details. A great deal of discretion is left to the compiler designer in choosing methods to carry out the specified actions. Two compilers for the same architect's language often produce compiled code of widely differing efficiency and storage requirements.

In fact, there is no necessary reason why there must be a compiler at all. One could use the architect's language to specify the form and function of the finished program and then turn the job over to a program builder. However, the computer *can* do a fairly good job of automatically producing a program for such languages, and the ability to have it do so gives the program architect a powerful tool not available to the construction architect—the ability to rapidly prototype designs. This is the power of the computer, and one of the aspects that makes the study of programming language so fascinating!

Exhibit 2.2. Representations of a date in English.

Dates are abstract real-world objects. We represent them in English by specifying an era, year, month, and day of the month. The era is usually omitted and the year is often omitted in representations of dates, because they can be deduced from context. The month is often encoded as an integer between 1 and 12.

Full, explicit representation:	January 2, 1986 AD
Common representations:	January 2, 1986
	Jan. 2, '86
	Jan. 2
	1-2-86
	2 Jan 86
	86-1-2

2.2 Representation

A *representation of an object* is a list of the relevant facts about that object in some language [Exhibit 2.2]. A *computer representation of an object* is a mapping of the relevant facts about that object, through a computer language, onto the parts of the machine.

Some languages support *high-level* or *abstract* representations, which specify the functional properties of an object or the symbolic names and data types of the fields of the representation [Exhibit 2.3]. A high-level representation will be mapped onto computer memory by a translator. The actual number and order of bytes of storage that will be used to represent the object may vary from translator to translator. In contrast, a computer representation is *low level* if it describes a particular implementation of the object, such as the amount of storage that will be used, and the position of each field in that storage area [Exhibit 2.4].

A *computer representation of a process* is a sequence of program definitions, specifications, or

Exhibit 2.3. High-level computer representations of a date.

An encoding of the last representation in Exhibit 2.2 is often used in programs. In a high-level language the programmer might specify that a date will be represented by three integers, as in this Pascal example:

```
TYPE date = RECORD year, month, day: integer END;
VAR BirthDate: date;
```

The programmer may now refer to this object and its components as:

```
BirthDate or
BirthDate.year or BirthDate.month or BirthDate.day
```

Exhibit 2.4. A low-level computer representation of a date.

In a low-level language such as assembler or FORTH, the programmer specifies the exact number of bytes of storage that must be allocated (or ALLOTted) to represent the date. In the FORTH declaration below, the keyword VARIABLE causes 2 bytes to be allocated, and 4 more are explicitly allocated using ALLOT. Then the programmer must manually define selection functions that access the fields of the object by adding an appropriate offset to the base address.

```
VARIABLE birth_date 4 ALLOT
: year 0 + ;      ( Year is first -- offset is zero bytes. )
: month 2 + ;     ( Month starts two bytes from the beginning. )
: day 4 + ;      ( Day is the fifth and sixth bytes. )
```

The variable named `birth_date` and its component fields can now be accessed by writing:

```
birth_date or
birth_date year or birth_date month or birth_date day
```

statements that can be performed on representations of objects from specified sets. We say that the representation of a process is *valid*, or correct, if the transformed object representation still corresponds to the transformed object in the real world.

We will consider three aspects of the quality of a representation: semantic intent, explicitness, and coherence. Abstract representations have these qualities to a high degree; low-level representations often lack them.

2.2.1 Semantic Intent

A data object (variable, record, array, etc.) in a program has some intended meaning that is known to the programmer but cannot be deduced with certainty from the data representation itself. This intended meaning is the programmer's *semantic intent*. For example, three 2-digit integers can represent a woman's measurements in inches or a date. We can only know the intended meaning of a set of data if the programmer communicates, or declares, the context in which it should be interpreted.

A program has *semantic validity* if it faithfully carries out the programmer's explicitly declared semantic intent. We will be examining mechanisms in various languages for expressing semantic intent and ensuring that it is carried out. Most programming languages use a data type to encode part of the semantic intent of an object. Before applying a function to a data object, the language translator tests whether the function is defined for that object and, therefore, is meaningful in its context. An attempt to apply a function to a data object of the wrong type is identified as a semantic error. A type checking mechanism can thus help a programmer write semantically valid (meaningful) programs.

Exhibit 2.5. The structure of a table expressed implicitly.

Pascal permits the construction and use of sorted tables, but the fact that the table is sorted cannot be explicitly declared. We can deduce that the table is sorted by noting that a sort algorithm is invoked, that a binary search algorithm is used, or that a sequential search algorithm is used that can terminate a search unsuccessfully before reaching the end of the table.

The order of entries (whether ascending or descending) can be deduced by careful analysis of three things:

- The comparison operator used in a search (< or >)
- The order of operands in relation to this operator
- The result (true or false) which causes the search to terminate.

Deductions of this sort are beyond the realistic present and future abilities of language translators.

2.2.2 Explicit versus Implicit Representation

The structure of a data object can be reflected *implicitly* in a program, by the way the statements are arranged [Exhibit 2.5], or it can be declared *explicitly* [Exhibit 2.6]. A language that can declare more kinds of things explicitly is more *expressive*.

Information expressed explicitly in a program may be used by the language translator. For example, if the COBOL programmer supplies a KEY clause, the processor will permit the programmer to use the efficient built-in binary search command, because the KEY clause specifies that the file is sorted in order by that field. The less-efficient sequential search command must be used to search any table that does not have a KEY clause.

A language that permits explicit communication of information must have a translator that can identify, store, organize, and utilize that information. For example, if a language permits programmers to define their own types, the translator needs to implement type tables (where type descriptions are stored), new allocation methods that use these programmer-defined descriptions, and more elaborate rules for type checking and type errors.

These translator mechanisms to identify, store, and interpret the programmer's declarations form the *semantic basis* of a language. Other mechanisms that are part of the semantic basis are those which implement binding (Chapters 6 and 9), type checking and automatic type conversion (Chapter 15), and module protection (Chapter 16).

2.2.3 Coherent versus Diffuse Representation

A representation is *coherent* if an external entity (object, idea, or process) is represented by a single symbol in the program (a name or a pointer) so that it may be referenced and manipulated as a unit [Exhibit 2.7]. A representation is *diffuse* if various parts of the representation are known by

Exhibit 2.6. COBOL: The structure of a table expressed explicitly.

COBOL allows explicit declaration of sorted tables. The key field(s) and the order of entries may be declared as in the following example. This table is intended to store the names of the fifty states of the United States and their two-letter abbreviations. It is to be stored so that the abbreviations are in alphabetical order.

```
01 state-table.
  02 state-entry
    OCCURS 50 TIMES
    ASCENDING KEY state-abbrev
    INDEXED BY state-index.
  03 state-abbrev          PICTURE XX.
  03 state-name           PICTURE X(20).
```

This table can be searched for a state abbreviation using the binary-search utility. A possible call is:

```
SEARCH ALL state-entry
  AT END PERFORM failed-search-process
  WHEN st-abbrev (state-index) = search-key PERFORM found-process.
```

COBOL also permits the programmer to declare Pascal-like tables for which the sorting order and key field are not explicitly declared. The `SEARCH ALL` command cannot be used to search such a table; the programmer can only use the less efficient sequential search command.

different names, and no one name or symbol applies to the whole [Exhibits 2.8 and 2.9].

A representation is coherent if all the parts of the represented object can be named by one symbol. This certainly does not imply that all the parts must be stored in consecutive (or contiguous) memory locations. Thus an object whose parts are connected by links or pointers can still be coherent [Exhibit 2.10].

The older languages (FORTRAN, APL) support coherent representation of complex data objects

Exhibit 2.7. A stack represented coherently in Pascal.

A stack can be represented by an array and an integer index for the number of items currently in the array. We can represent a stack coherently by grouping the two parts together into a record. One parameter then suffices to pass this stack to a function.

```
TYPE stack = RECORD
    store: ARRAY [1..max_stack] of stack_type;
    top: 0..max_stack
END;
```

Exhibit 2.8. A stack represented diffusely in FORTRAN and Pascal.

A stack can be represented diffusely as an array of items and a separate index (an integer in the range 0 to the size of the array).

FORTRAN: A diffuse representation is the only representation of a stack that is possible in FORTRAN because the language does not support heterogeneous records. These declarations create two objects which, taken together, comprise a stack of 100 real numbers:

```
REAL    STSTORE( 100 )
INTEGER STTOP
```

Pascal: A stack can be represented diffusely in Pascal. This code allocates the same amount of storage as the coherent version in Exhibit 2.7, but two parameters are required to pass this stack to a procedure.

```
TYPE stack_store = ARRAY [1..max_stack] of stack_type;
   stack_top = 0..max_stack;
```

Exhibit 2.9. Addition represented diffusely or coherently.

FORTH: The operation of addition is represented diffusely because addition of single-length integers, double-length integers, and mixed-length integers are named by three different symbols, (+, D+, and M+) and no way is provided to refer to the general operation of “integer +”.

C: The operation of addition is represented coherently. Addition of single-length integers, double-length integers, and mixed-length integers are all named “+”. The programmer may refer to “+” for addition, without concern for the length of the integer operands.

Exhibit 2.10. A coherently but not contiguously represented object.

LISP: A linked tree structure or a LISP list is a coherent object because a single pointer to the head of the list allows the entire list to be manipulated. A tree or a list is generally implemented by using pointers to link storage cells at many memory locations.

C: A sentence may be represented as an array of words. One common representation is a contiguous array of pointers, each of which points to a variable-length string of characters. These strings are normally allocated separately and are not contiguous.

only if the object can be represented by a homogeneous array of items of the same data type.¹ Where an object has components represented by different types, separate variable names must be used. COBOL and all the newer languages support coherent heterogeneous groupings of data. These are called “records” in COBOL and Pascal, and “structures” in C.

The FORTRAN programmer can use a method called *parallel arrays* to model an array of heterogeneous records. The programmer declares one array for each field of the record, then uses a single index variable to refer to corresponding elements of the set of arrays. This diffuse representation accomplishes the same goal as a Pascal array of records. However, an array of records represents the problem more clearly and explicitly and is easier to use. For example, Pascal permits an array of records to be passed as a single parameter to a function, whereas a set of parallel arrays in FORTRAN would have to be passed as several parameters.

Some of the newest languages support coherence further by permitting a set of data representations to be grouped together with the functions that operate on them. Such a coherent grouping is called a “module” in Modula-2, a “cluster” in CLU, a “class” in Smalltalk, and a “package” in Ada.

2.3 Language Design

In this section we consider reasons why a language designer might choose to create an “architect’s language” with a high degree of support for abstraction, or a “builder’s language” with extensive control over low-level aspects of representation.

2.3.1 Competing Design Goals

Programming languages have evolved greatly since the late 1950s when the first high-level languages, FORTRAN and COBOL, were implemented. Much of this evolution has been made possible by the improvements in computer hardware: today’s machines are inconceivably cheap, fast, and large (in memory capacity) compared to the machines available in 1960. Although those old machines were physically bulky and tremendously expensive, they were hardly more powerful than machines that today are considered to be toys.

Along with changes in hardware technology came improvements in language translation techniques. Both syntax and semantics of the early languages were ad hoc and clumsy to translate. Formal language theory and formal semantics affected language design in revolutionary ways and have resulted in better languages with cleaner semantics and a more easily translatable syntax.

There are many aspects of a language that the user cannot modify or extend, such as the data structuring facilities and the control structures. Unless a language system supports a preprocessor, the language syntax, also, is fixed. If control structures and data definition facilities are not built

¹The EQUIVALENCE statement can be used to circumvent this weakness by defining the name of the coherent object as an overlay on the storage occupied by the parts. This does not constitute adequate support for compound heterogeneous objects.

in, they are not available. Decisions to include or exclude such features must, therefore, be made carefully. A language designer must consider several aspects of a potential feature to decide whether it supports or conflicts with the design goals.

During these thirty years of language development, a consensus has emerged about the importance of some language features, for example, type checking and structured conditionals. Most new languages include these. On other issues, there has been and remains fundamental disagreement, for instance, over the question of whether procedural or functional languages are “better”. No single set of value judgments has yet emerged, because different languages have different goals and different intended uses. The following are some potential language design goals:

- Utility. Is a feature often useful? Can it do important things that cannot be done using other features of the language?
- Convenience. Does this feature help avoid excessive writing? Does this feature add or eliminate clutter in the code?
- Efficiency. Is it easy or difficult to translate this feature? Is it possible to translate this feature into efficient code? Will its use improve or degrade the performance of programs?
- Portability. Will this feature be implementable on any machine?
- Readability. Does this form of this feature make a program more readable? Will a programmer other than the designer understand the intent easily? Or is it cryptic?
- Modeling ability. Does this feature help make the meaning of a program clear? Will this feature help the programmer model a problem more fully, more precisely, or more easily?
- Simplicity. Is the language design as a whole simple, unified, and general, or is it full of dozens of special-purpose features?
- Semantic clarity. Does every legal program and expression have one defined, unambiguous, meaning? Is the meaning constant during the course of program execution?

These goals are all obviously desirable, but they conflict with each other. For example, a simple language cannot possibly include all useful features, and the more features included, the more complicated the language is to learn, use, and implement. *Ada* illustrates this conflict. *Ada* was designed for the Department of Defense as a language for embedded systems, to be used in all systems development projects, on diverse kinds of hardware. Thus it necessarily reflects a high value placed on items at the beginning and middle of the preceding list of design goals. The result is a very large language with a long list of useful special features.

Some language researchers have taken as goals the fundamental properties of language shown at the end of the list of design goals. Outstanding examples include *Smalltalk*, a superior language for modeling objects and processes, and *Miranda*, which is a list-oriented functional language that achieves both great simplicity and semantic clarity.

Exhibit 2.11. A basic type not supported by Pascal.

Basic type implemented by most hardware: bit strings

Common lengths: 8, 16, and 32 bits (1, 2, and 4 bytes)

Operations built into most hardware	Symbol in C
a right shift n places	$a \gg n$
a left shift n places	$a \ll n$
a and b	$a \& b$
a or b	$a b$
a exclusive or b	$a \wedge b$
complement a	$\sim a$

2.3.2 The Power of Restrictions

Every language imposes restrictions on the user, both by what it explicitly prohibits and by what it simply doesn't provide. Whenever the underlying machine provides instructions or capabilities that cannot be used in a user program, the programming language is imposing a restriction on the user. For example, Pascal does not support the type "bit string" and does not have "bit string" operators [Exhibit 2.11]. Thus Pascal restricts access to the bit-level implementations of objects.

The reader must not confuse logical operators with bitwise operators. Pascal supports the logical (Boolean) data type and *logical operators* *and*, *or*, and *not*. Note that there is a difference between these and the *bitwise operators* [Exhibit 2.12]. Bitwise operators apply the operation between every corresponding pair of bits in the operands. Logical operators apply the operation to the operands as a whole, with 00000000 normally being interpreted as False and anything else as True.

In general, restrictions might prevent writing the following two sorts of sentences:

1. Useless or meaningless sentences such as " $3 := 72.9 + 'a'$ ".
2. Sentences useful for modeling some problem, that could be written efficiently in assembly

Exhibit 2.12. Bitwise and logical operations.

The difference between bitwise and logical operations can be seen by comparing the input and output from these operations in C:

Operation	Operands as bit strings	Result	Explanation
bitwise and	10111101 & 01000010	00000000	no bit pairs match
logical and	10111101 && 01000010	00000001	both operands represent True
complement	\sim 01000010	10111101	each bit is flipped
logical not	! 01000010	00000000	operand is True

Exhibit 2.13. Useful pointer operations supported in C.

In the C expressions below, `p` is a pointer used as an index for the array named “ages”. Initially, `p` will store the machine address of the beginning of `ages`. To make `p` index the next element, `p` will be incremented by the number of bytes in one element of `ages`. The array element is accessed by dereferencing the pointer.

This code contains an error in logic, which is pointed out in the comments. It demonstrates one semantic problem that Pascal’s restrictions were designed to prevent. The loop will be executed too many times, and this run-time error will not be detected. Compare this to the similar Pascal loop in Exhibit 2.14.

```

int ages[10];          /* Array "ages" has subscripts 0..9. */
...
p = ages;             /* Make p point at ages[0]. */
end = &ages[10];     /* Compute address of eleventh array element. */
while (p <= end){    /* Loop through eleventh array address. */
    printf("%d \n", *p); /* Print array element in decimal integer format. */
    p++;              /* Increment p to point at next element of ages. */
}

```

code but are prohibited.

A good example of a useful facility that some languages prohibit is explicit address manipulation. This is supported in C [Exhibit 2.13]. The notation for pointer manipulation is convenient and is generally used in preference to subscripting when the programmer wishes to process an array sequentially.

In contrast, manipulation of addresses is restricted in Pascal to prevent the occurrence of meaningless and potentially dangerous dangling pointers (see Chapter 6). Address manipulation is prohibited and address arithmetic is undefined in Pascal. Nothing comparable to the C code in Exhibit 2.13 can be written in Pascal. A pointer can’t be set to point at an array element, and it cannot be incremented to index through an array.

This is a significant loss in Pascal because using a subscript involves a lot of computation: the subscript must be checked against the minimum and maximum legal values, multiplied by the size of an array element, and added to the base address of the array. Checking whether a pointer has crossed an array boundary and using it to access an element could be done significantly faster.

Let us define *flexibility* to mean the absence of a restriction, and call a restriction *good* if it prevents the writing of nonsense, and *bad* if it prevents writing useful things. Some restrictions might have both good and bad aspects. A *powerful* language must have the flexibility to express a wide variety of actions—preferably a variety that approaches the power of the underlying machine.

But power is not a synonym for flexibility. The most flexible of all languages is assembly language, but assemblers lack the power to express a problem solution succinctly and clearly. A

Exhibit 2.14. A meaningless operation prohibited in Pascal but not in C.

Subscripts are checked at run time in Pascal. Every subscript that is used must be within the declared bounds.

```
VAR ages: array[0..9] of integer;
    p: integer;
...
p := 0;
while p <= 10 do begin /* Loop through last array subscript. */
    writeln( ages[p] ); /* Print the array element. */
    p:=p+1 /* Make p point at next element of array. */
end;
```

The last time around this loop the subscript, *p*, has a value that is out of range. This will be detected, and a run-time error comment will be generated. The analogous C code in Exhibit 2.13 will run and print garbage on the last iteration. The logical error will not be detected, and no error comment will be produced.

second kind of power is provided by sophisticated mechanisms in the semantic basis of a language that let the programmer express a lot by saying a little. The type definition and type checking facility in any modern language is a good example of a powerful mechanism.

A third kind of power can come from “good” restrictions that narrow the variety of things that can be written. If a restriction can eliminate troublesome or meaningless sentences automatically, then programmers will not have to check, explicitly, whether such meaningless sections occur in their programs. Pascal programs rarely run wild and destroy memory. But C and FORTH programs, with unrestricted pointers and no subscript bounds checking, often do so. A language should have enough good restrictions so that the programmer and translator can easily distinguish between a meaningful statement and nonsense.

For example, an attempt to access an element of an array with a subscript greater than the largest array subscript is obviously meaningless in any language. The underlying machine hardware permits one to **FETCH** and **STORE** information beyond the end of an array, but this can have no possible useful meaning and is likely to foul up the further operation of the program. The semantics of standard Pascal prescribe that the actual value of each subscript expression should be checked at run time. An error comment is generated if the value is not within the declared array bounds. Thus, all subscripting in Pascal is “safe” and cannot lead to destruction of other information [Exhibit 2.14].

No such array bounds check is done in C. Compare Exhibits 2.13 and 2.14. These two code fragments do analogous things, but the logical error inherent in both will be trapped by Pascal and ignored by C. In C, a **FETCH** operation with too large a subscript can supply nonsensical information, and a **STORE** can destroy vital, unrelated information belonging to variables allocated before or after the array. This situation was exploited to create the computer network “worm” that

invaded hundreds of computer systems in November 1988. It disabled these systems by flooding their processing queues with duplicates of itself, preventing the processing of normal programs. This escapade resulted in the arrest and conviction of the programmer.

Often, as seen in Exhibit 2.13, a single feature is both useful and dangerous. In that case, a language designer has to make a value judgement about the relative importance of the feature and the danger in that feature. If the designer considers the danger to outweigh the importance, the restriction will be included, as Wirth included the pointer restrictions in Pascal. If the need outweighs the danger, the restriction will not be included. In designing C, Kernighan and Ritchie clearly felt that address manipulation was vital, and decided that the dangers of dangling pointers would have to be avoided by careful programming, not by imposing general restrictions on pointers.

2.3.3 Principles for Evaluating a Design

In the remaining chapters of this book we will sometimes make value judgments about the particular features that a language includes or excludes. These judgments will be based on a small set of principles.

Principle of Frequency

The more frequently a language feature will be used, the more convenient its use should be, and the more lucid its syntax should be. An infrequently used feature can be omitted from the core of the language and/or be given a long name and less convenient syntax.

C provides us with examples of good and poor application of this principle. The core of the C language does not include a lot of features that are found in the cores of many other languages. For example, input/output routines and mathematical functions for scientific computation are not part of the standard language. These are relegated to libraries, which can be searched if these features are needed. There are two C libraries which are now well standardized, the “math library” and the “C library” (which includes the I/O functions).

The omission of mathematical functions from C makes good sense because the intended use of C was for systems programming, not scientific computation. Putting these functions in the math library makes them available but less convenient. To use the math library, the loader must have the library on its search path and the user must include a header file in the program which contains type declarations for the math functions.

On the other hand, most application programs use the input-output functions, so they should be maximally convenient. In C they aren't; in order to use them a programmer must include the appropriate header file containing I/O function and macro declarations, and other essential things. Thus nearly every C application program starts with the instruction “`#include <stdio.h>`”. This could be considered to be a poor design element, as it would cost relatively little to build these definitions into the translator.

Principle of Locality

A good language design enables and encourages, perhaps even enforces, locality of effects. The further the effects of an action reach in time (elapsed during execution) or in space (measured in pages of code), the more complex and harder it is to debug a program. The further an action has influence, the harder it is to remember relevant details, and the more subtle errors seem to creep into the code.

To achieve locality, the use of global variables should be minimized or eliminated and all transfers of control should be short-range. A concise restatement of this principle, in practical terms is:

Keep the effects of everything confined to as local an area of the code as possible.

Here are some corollaries of the general principle, applied to lexical organization of a program that will be debugged on-line, using an ordinary nonhierarchical text editor:

- A control structure that won't fit on one screen is too long; shorten it by defining one or more scopes as subroutines.
- All variables should be defined within one screen of their use. This applies whether the user's screen is large or small—the important thing is to be able to see an entire unit at one time.
- If your subroutine won't fit on two screens, it is too long. Break it up.

Global Variables. Global variables provide a far more important example of the cost of nonlocality. A global variable can be changed or read anywhere within a program. Specifically, it can be changed accidentally (because of a typographical error or a programmer's absentmindedness) in a part of the program that is far removed from the section in which it is (purposely) used.

This kind of error is hard to find. The apparent fault is in the section that is supposed to use the variable, but if that section is examined in isolation, it will work properly. To find the cause of the error, a programmer must trace the operation of the entire program. This is a tedious job. The use of unnecessary global variables is, therefore, dangerous.

If the program were rewritten to declare this variable locally within the scope in which it is used, the distant reference would promptly be identified as an error or as a reference to a semantically distinct variable that happens to have the same name.

Among existing languages are those that provide only global variables, provide globals but encourage use of locals and parameters, and provide only parameters.

Unrestricted use of global variables. A BASIC programmer cannot restrict a variable to a local scope. This is part of the reason that BASIC is not used for large systems programs.

Use of global variables permitted but use of locals encouraged. Pascal and C are block structured languages that make it easy to declare variables in the procedure in which they are used.² Their default method of parameter passing is call-by-value. Changing a local variable or value parameter has only local effects. Programmers are encouraged to use local declarations, but they can use global variables in place of both local variables and parameters.

Use of global variables prohibited. In the modern *functional languages* there are no global variables. Actually, there are no variables at all, and parameter binding takes the place of assignment to variables. Assignment was excluded from this class of languages because it can have nonlocal effects. The result is languages with elegant, clean semantics.

Principle of Lexical Coherence

Sections of code that logically belong together should be physically adjacent in the program. Sections of code that are not related should not be interleaved. It should be easy to tell where one logical part of the program ends and another starts. A language design is good to the extent that it permits, requires, or encourages lexical coherence.

This principle concerns only the surface syntax of the language and is, therefore, not as important as the other principles, which concern semantic power. Nonetheless, good human engineering is important in a language, and lexical coherence is important to make a language usable and readable.

Poor lexical coherence can be seen in many languages. In Pascal the declarations of local variables for the main program must be near the top of the program module, and the code for `main` must be at the bottom [Exhibit 2.15]. All the function and procedure definitions intervene. In a program of ordinary size, several pages of code come between the use of a variable in `main` and its definition.

Recently, hierarchical editors have been developed for Pascal. They allow the programmer to "hide" a function definition "under" the function header. A program is thus divided into levels, with the main program at the top level and its subroutines one level lower. If the subroutines have subroutines, they are at level three, and so on. When the main program is on the screen, only the top level code appears, and each function definition is replaced by a simple function header. This brings the main program's body back into the lexical vicinity of its declarations. When the programmer wishes to look at the function definition, simple editor commands will allow him to descend to that level and return.

A similar lack of coherence can be seen in early versions of LISP.³ LISP permits a programmer to write a function call as a literal function, called a *lambda expression*, followed by its actual arguments, as shown at the top of Exhibit 2.16. The dummy parameter names are separated from the matching parameter values by an arbitrarily long function body.

²Local declarations are explained fully in Chapter 6; parameters are discussed in Chapter 9, Section 9.2.

³McCarthy et al. [1962].

Exhibit 2.15. Poor lexical coherence for declarations and code in Pascal.

The parts of a Pascal program are arranged in the order required to permit one-pass compilation:

- Constant declarations.
- Type declarations.
- Variable declarations.
- Procedure and Function declarations.
- Code.

Good programming style demands that most of the work of the program be done in subroutines, and the part of the program devoted to subroutine definitions is often many pages long. The variable declarations and code for the main program are, therefore, widely separated, producing poor lexical coherence.

Exhibit 2.16. Syntax for lambda expressions in LISP.

The order of elements in the primitive syntax is:

```
((lambda
  ( <list of dummy parameter names> )
  ( <body of the function> ))
<list of actual parameter values>))
```

The order of elements in the extended syntax is:

```
(let
  ( <list of dummy name - actual value pairs> )
  ( <body of the function> ))
```

Exhibit 2.17. A LISP function call with poor coherence.

The following literal function is written in the primitive LISP syntax. It takes two parameters, x , and y . It returns their product plus their difference. It is being called with the arguments 3.5 and $a + 2$. Note that the parameter declarations and matching arguments are widely separated.

```
((lambda ( x y )
  (+ (* x y)
     (- x y) ))
 3.5 (+ a 2) )
```

This lack of lexical coherence makes it awkward and error prone for a human to match up the names with the values, as shown in Exhibit 2.17. The eye swims when interpreting this function call, even though it is simple and the code section is short.

Newer versions of LISP, for example Common LISP,⁴ offer an improved syntax with the same semantics but better lexical coherence. Using the `let` syntax, dummy parameter names and actual values are written in pairs at the top, followed by the code. This syntax is shown at the bottom of Exhibit 2.16, and an example of its use is shown in Exhibit 2.18.

A third, and extreme, example of poor lexical coherence is provided by the syntax for function definitions in SNOBOL. A SNOBOL IV function is defined by a function header of the following form:

```
( '<name>' ( <parameter list> ) ( <local variable name list> ) , '<entry label>' )
```

The code that defines the action of the subroutine can be anywhere within the program module, and it starts at the line labeled *<entry label>*. It does not even need to be all in the same place,

⁴Kessler[1988], p. 59.

Exhibit 2.18. A LISP function call with good coherence.

The following function call is written in LISP using the extended “let” syntax. It is semantically equivalent to the call in Exhibit 2.17.

```
(let ((x 3.5) (y (+ a 2) ))
  ((+ (* x y)
      (- x y) ) )
```

Compare the ease of matching up parameter names and corresponding arguments here, with the difficulty in Exhibit 2.17. The lexically coherent syntax is clearly better.

Exhibit 2.19. Poor lexical coherence in SNOBOL.

SNOBOL has such poor lexical coherence that semantically unrelated lines can be interleaved, and no clear indication exists of the beginning or end of any program segment. This program converts English to Pig Latin. It is annotated below.

```

1.      DEFINE('PIG(X) Y, Z', 'PIG1')  :(MAIN)
2.  PROC  OUTPUT = PIG(IN)
3.  MAIN  IN = INPUT                      :F(END) S(PROC)
4.  PIG1  PIG = NULL
5.      X SPAN(' ') =                      :F(RETURN)
6.  LOOP  X BREAK(' ') . Y SPAN(' ') =    :F(RETURN)
7.      Y LEN(1) . Z =
8.      PIG = PIG Y Z 'AY'                :(LOOP)
9.  END   OUTPUT = '.'
```

Program Notes. The main program begins on line 1, with the declaration of a header for a subroutine named PIG. Line 1 directs that execution is to continue on the line named MAIN. The subroutine declaration says that the subroutine PIG has one parameter, X, and two local variables, Y and Z. The subroutine code starts on the line with the label “PIG1”.

Lines 2, 3, and 9 belong to the main program. They read a series of messages, translate each to Pig Latin, write them out, and quit when a zero-length string is entered.

Lines 4 through 8 belong to the subroutine PIG. Line 4 initializes the answer to the null string. Line 5 strips leading blanks off the parameter, X. Line 6 isolates the next word in X (if any), and line 7 isolates its first letter. Finally, line 8 glues this word onto the output string with its letters in a different order and loops back to line 6.

since each of its lines may be attached to the next by a GOTO.

Thus a main program and several subroutines could be interleaved. (We do admit that a sane programmer would never do such a thing.) Exhibit 2.19 shows a SNOBOL program, with subroutine, that translates an English sentence into Pig Latin. The line numbers are not part of the program but are used to key it to the program notes that follow.

Principle of Distinct Representation

Each separate semantic object should be represented by a separate syntactic item. Where a single syntactic item in the program is used for multiple semantic purposes, conflicts are bound to occur, and one or both sets of semantics will be compromised. The line numbers in a BASIC program provide a good example.

BASIC was the very first interactive programming language. It combined an on-line editor, a file system, and an interpreter to make a language in which simple problems could be programmed

Exhibit 2.20. BASIC: GOTOs and statement ordering both use line numbers.

Line numbers in BASIC are used as targets of GOTO and also to define the proper sequence of the statements; the editor accepts lines in any order and arranges them by line number. Thus the user could type the following lines in any order and they would appear as follows:

```

2  SUM = SUM + A
4  PRINT SUM
6  IF A < 10 GO TO 2
8  STOP

```

Noticing that some statements have been left out, the programmer sees that three new lines must be inserted. The shortsighted programmer has only left room to insert one line between each pair, which is inadequate here, so he or she renumbers the old line 2 as 3 to make space for the insertion. The result is:

```

1  LET SUM = 0
2  LET A = 1
3  SUM = SUM + A
4  PRINT SUM
5  LET A = A + 1
6  IF A < 10 GO TO 2
8  STOP

```

Notice that the loop formed by line 6 now returns to the wrong line, making an infinite loop. Languages with separate line numbers and statement labels do not have this problem.

quickly. The inclusion of an editor posed a new problem: how could the programmer modify the program and insert and delete lines? The answer chosen was to have the programmer number every line, and have the editor arrange the lines in order by increasing line number.

BASIC was developed in the context of FORTRAN, which uses numeric line numbers as statement labels. It was, therefore, natural for BASIC to merge the two ideas and use one mechanism, the monotonically increasing line number, to serve purposes (1) and (2) below. When the language was extended to include subroutines, symbolic names for them were not defined either. Rather, the same line numbers were given a third use. Line numbers in BASIC are, therefore, multipurpose:

1. They define the correct order of lines in a program.
2. They are the targets of GOTOs and IFs.
3. They define the entry points of subroutines (the targets of GOSUB).

A conflict happens because inserting code into the program requires that line numbers change, and GOTO requires that they stay constant. Because of this, adding lines to a program can be a

complicated process. Normally, BASIC programmers leave regular gaps in the line numbers to allow for inserting a few lines. However, if the gap in numbering between two successive lines is smaller than the number of lines to be inserted, something will have to be renumbered. But since the targets of GOTOs are not marked in any special way, renumbering implies searching the entire program for GOTOs and GOSUBs that refer to any of the lines whose numbers have been changed. When found, these numbers must be updated [Exhibit 2.20]. Some BASIC systems provide a renumbering utility, others don't. In contrast, lines can be added almost anywhere in a C program with minimal local adjustments.

Principle of Too Much Flexibility

A language feature is bad to the extent that it provides flexibility that is not useful to the programmer, but that is likely to cause syntactic or semantic errors.

For example, any line in a BASIC program can be the target of a GOTO or a GOSUB statement. An explicit label declaration is not needed—the programmer simply refers to the line numbers used to enter and edit the program. A careless or typographical error in a GOTO line number will not be identified as a syntactic error.

Every programmer knows which lines are supposed to be the targets of GOTOs, and she or he could easily identify or label them. But BASIC supplies no way to restrict GOTOs to the lines that the programmer knows should be their targets. Thus the translator cannot help the programmer ensure valid use of labels.

We would say that the ability to GOTO or GOSUB to any line in the program without writing an explicit label declaration is excessively flexible: it saves the programmer the minor trouble of declaring labels, but it leads to errors. If there were some way to restrict the set of target lines, BASIC would be a better and more powerful language. Power comes from a translator's ability to identify and eliminate meaningless commands, as well as from a language's ability to express aspects of a model.

Another example of useless flexibility can be seen in the way APL handles GOTO and statement labels. APL provides only three control structures: the function call, sequential execution, and a GOTO statement. A GOTO can only transfer control locally, within the current function definition. All other control structures, including ordinary conditionals and loops, must be defined in terms of the conditional GOTO.

As in BASIC, numeric line numbers are used both to determine the order of lines in a program and as targets of the GOTO. But the problems in BASIC with insertions and renumbering are avoided because, unlike BASIC, symbolic labels are supported. A programmer may write a symbolic label on a line and refer to it in a GOTO, and this will have the correct semantics even if lines are inserted and renumbering happens. During compilation of a function definition (the process that happens when you leave the editor), the lines are renumbered. Each label is bound to a constant integer value: the number of the line on which it is defined. References to the label in the code are replaced by that constant, which from then on has exactly the same semantics as an integer. (Curiously, constants are not otherwise supported by the language.)

Exhibit 2.21. Strange but legal GOTOs in APL.

The `GOTO` is written with a right-pointing arrow, and its target may be any expression. The statements below are all legal in APL.

$\rightarrow (x + 2) - 6$	Legal so long as the result is an integer.
$\rightarrow 3\ 4\ 7$	An array of line numbers is given; control will be transferred to the first.
$\rightarrow \iota 0$	ιN returns a vector of N numbers, ranging from 1 to N . Thus, $\iota 0$ returns a vector of length 0, which is the null object. A branch to the null object is equivalent to a no-op.

Semantic problems arise because the labels are translated into integer constants and may be operated on using integer operations such as multiplication and division! Further, the APL `GOTO` is completely unrestricted; it can name either a symbolic label or an integer line number, whether or not that line number is defined in that subroutine. Use of an undefined line number is equivalent to a function return. These semantics have been defined so that some interpretation is given no matter what the result of the expression is [Exhibit 2.21].

Because the target of a `GOTO` may be computed and may depend on variables, any line of the subroutine might potentially be its target. It is impossible at compile time to eliminate any line from the list of potential targets. Thus, at compile time, the behavior of a piece of code may be totally unpredictable.

APL aficionados love the flexibility of this `GOTO`. All sorts of permutations and selection may be done on an array of labels to implement every conceivable variety of conditional branch. Dozens of useful idioms, or phrases, such as the one in Exhibit 2.22, have been developed using this `GOTO` and published for other APL programmers to use.

It is actually fun to work on and develop a new control structure idiom. Many language designers, though, question the utility and wisdom of permitting and relying on such idiomatic control structures. They must be deciphered to be understood, and the result of a mistake in definition or use is a totally wrong and unpredictable branch. Even a simple conditional branch

Exhibit 2.22. A computed GOTO idiom in APL.

$$\rightarrow (\text{NEG}, \text{EQ}, \text{POS}) [2 + \times N]$$

This is a three-way branch very similar to the previous example and analogous to the FORTRAN arithmetic `IF`. The signum function, \times , returns -1 if N is negative, +1 if N is positive, and 0 otherwise. Two is added to the result of signum, and the answer is used to subscript a vector of labels. One of the three branches is always taken.

to the top of a loop can be written with four different idioms, all in common use. This makes it difficult to learn to read someone else's code. Proofs of correctness are practically impossible.

We have shown that APL's totally unrestricted GOTO has the meaningless and useless flexibility to branch to any line of the program, and that the lack of any other control structure necessitates the use of cryptic idioms and produces programs with unpredictable behavior. These are severe semantic defects! By the principle of *Too Much Flexibility*, this unrestricted GOTO is bad, and APL would be a more powerful language with some form of restriction on the GOTO.

The Principle of Semantic Power

A programming language is powerful (for some application area) to the extent that it permits the programmer to write a program easily that expresses the *model*, the *whole model*, and *nothing but the model*. Thus a powerful language must support explicit communication of the model, possibly by defining a general object and then specifying restrictions on it. A restriction imposed by the language can support power at the price of flexibility that might be necessary for some applications. On the other hand, a restriction imposed by the user expresses only the semantics that the user wants to achieve and does not limit him or her in ways that obstruct programming.

The programmer should be able to specify a program that computes the "correct" results and then be able to verify that it does so. All programs should terminate properly, not "crash". Faulty results from correct data should be provably impossible.

Part of a model is a description of the data that is expected. A powerful language should let the programmer write data specifications in enough detail so that "garbage in" is detected and does not cause "garbage out".

The Principle of Portability

A *portable program* is one that can be compiled by many different compilers and run on different hardware, and that will work correctly on all of them. If a program is portable, it will be more useful to more people for more years. We live in times of constant change: we cannot expect to have the same hardware or operating system available in different places or different years.

But portability limits flexibility. A portable program, by definition, cannot exploit the special features of some hardware. It cannot rely on any particular bit-level representation of any object or function; therefore, it cannot manipulate such things. One might want to do so to achieve efficiency or to write low-level system programs.

Languages such as Standard Pascal that restrict access to pointers and to the bit-representations of objects force the programmer to write portable code but may prohibit him or her from writing efficient code for some applications.

Sometimes features are included in a language for historical reasons, even though the language supports a different and better way to write the same thing. As languages develop, new features are added that improve on old features. However, the old ones are seldom eliminated because *upward compatibility* is important. We want to be able to recompile old programs on new versions of the

Exhibit 2.23. An archaic language feature in FORTRAN.

The `arithmetic IF` statement was the only conditional statement in the original version of FORTRAN. It is a three-way conditional GOTO based directly on the conditional jump instruction of the IBM 704. An example of this statement is:

```
IF (J-1) 21, 76, 76
```

The expression in parentheses is evaluated first. If the result is negative, control goes to the first label on the list (21). For a zero value, control goes to the second label (76), and for a positive value, to the third (76). More often than not, in practical use, two of the labels are the same.

The `arithmetic IF` has been superseded by the modern `block IF` statement. Assume that `<block 1>` contains the statements that followed label 21 above, and `<block2>` contains the statements following statement 76. Then the following `block IF` statement is equivalent to the `arithmetic IF` above:

```
IF J-1 .LT. 0 THEN
    <block 1>
ELSE
    <block 2>
ENDIF
```

translator. Old languages such as COBOL and FORTRAN have been through the process of change and restandardization several times. Some features in these languages are completely archaic, and programmers should be taught not to use them [Exhibit 2.23]. Many of these features have elements that are inherently error prone, such as reliance on GOTOs. Moreover, they will eventually be dropped from the language standard. At that point, any programs that use the archaic features will require extensive modernization before they can be modified in any way.

Our answer to redundant and archaic language features is simple: don't use them. Find out what constitutes modern style in a language and use it consistently. Clean programming habits and consistent programming style produce error-free programs faster.

Another kind of redundancy is seen in Pascal, which provides two ways to delimit comments: `(* This is a comment. *)` and `{ This is a comment. }` The former way was provided, as part of the standard, for systems that did not support the full ASCII character set. It will work in all Pascal implementations and is thus more portable. The latter way, however, is considered more modern and preferred by many authors. Some programmers use both: one form for comments, the other to "comment out" blocks of code.

The language allows both kinds of comment delimiters to be used in a program. However, mixing the delimiters is a likely source of errors because they are not interchangeable. A comment must begin and end with the same kind of delimiter. Thus whatever conventions a programmer chooses should be used consistently. The programmer must choose either the more portable way or the more modern way, a true dilemma.

2.4 Classifying Languages

It is tempting to classify languages according to the most prominent feature of the language and to believe that these features make each language group fundamentally different from other groups. Such categorizations are always misleading because:

- Languages in different categories are fundamentally more alike than they are different. Believing that surface differences are important gets in the way of communication among groups of language users.
- We tend to associate things that occur together in some early example of a language category. We tend to believe that these things must always come together. This impedes progress in language design.
- Category names are used loosely. Nobody is completely sure what these names mean, and which languages are or are not in any category.
- Languages frequently belong to more than one category. Sorting them into disjoint classes disguises real similarities among languages with different surface syntax.

2.4.1 Language Families

Students do need to understand commonly used terminology, and it is sometimes useful to discuss a group of languages having some common property. With this in mind, let us look at some of the “language families” that people talk about and try to give brief descriptions of the properties that characterize each family. As you read this section, remember that these are not absolute, mutually exclusive categories: categorizations are approximate and families overlap heavily. Examples are listed for each group, and some languages are named several times.

Interactive Languages. An interactive language is enmeshed in a system that permits easy alternation between program entry, translation, and execution of code. We say that it operates using a REW cycle: the system Reads an expression, Evaluates it, and Writes the result on the terminal, then waits for another input.

Programs in interactive languages are generally structured as a series of fairly short function and object definitions. Translation happens when the end of a definition is typed in. Programs are usually translated into some intermediate form, not into native machine code. This intermediate form is then interpreted. Many interactive languages, such as FORTH and Miranda, use the term “compile” to denote the translation of source code into the internal form.

Examples: APL, BASIC, FORTH, LISP, T, Scheme, dBASE, Miranda.

Structured Languages. Control structures are provided that allow one to write programs without using GOTO. Procedures with call-by-value parameters⁵ are supported. Note that we call Pascal

⁵See Chapter 9.

a structured language even though it contains a `GOTO`, because it is not necessary to use that `GOTO` to write programs.

Examples: Pascal, C, FORTH, LISP, T, Scheme.

Strongly Typed Languages. Objects are named, and each name has a type. Every object belongs to exactly one type (types are disjoint). The types of actual function arguments are compared to the declared types of the dummy parameters during compilation. A mismatch in types or in number of parameters will produce an error comment. Many strongly typed languages, including Pascal, Ada, and ANSI C, include an “escape hatch”—that is, some mechanism by which the normal type-checking process can be evaded.

Examples: FORTRAN 77, Pascal, Ada, ANSI C (but not the original C), ML, Miranda.

Object-oriented Languages. These are extensions or generalizations of the typed languages. Objects are typed and carry their type identity with them at all times. Any given function may have several definitions, which we will call *methods*.⁶ Each method operates on a different type of parameter and is associated with the type of its first parameter. The translator must *dispatch* each function call by deciding which defining method to invoke for it. The method associated with the type of the first parameter will be used, if it exists.

Object-oriented languages have non-disjoint types and function inheritance. The concept of *function inheritance* was introduced by Simula and popularized by Smalltalk, the first language to be called “object-oriented”. A type may be a subset of another type. The function dispatcher will use this subset relationship in the dispatching process. It will select a function belonging to the supertype when none is defined for the subtype.

Actually, many of these characteristics also apply to APL, an old language. It has objects that carry type tags and functions with multiple definitions and automatic dispatching. It is not a full object-oriented language because it lacks definable class hierarchies.

Examples: Simula, Smalltalk, T, C++. APL is object-oriented in a restricted sense.

Procedural Languages. A program is an ordered sequence of statements and procedure calls that will be evaluated sequentially. Statements interact and communicate with each other through variables. Storing a value in a variable destroys the value that was previously stored there. (This is called destructive assignment.) Exhibit 2.24 is a diagram of the history of this language family. Modern procedural languages also contain extensive functional elements.⁷

Examples: Pascal, C, Ada, FORTRAN, BASIC, COBOL.

Functional Languages, Old Style. A program is a nested set of expressions and function calls. Call-by-value parameter binding, not assignment, is the primary mechanism used to give names to variables. Functions interact and communicate with each other through the parameter stack.

⁶This is the term used in Smalltalk.

⁷See Chapter 8.

Exhibit 2.24. The development of procedural languages.

Concepts and areas of concern are listed on the left. Single arrows show how these influenced language design and how some languages influenced others. Dotted double arrows indicate that a designer was strongly influenced by the bad features of an earlier language.

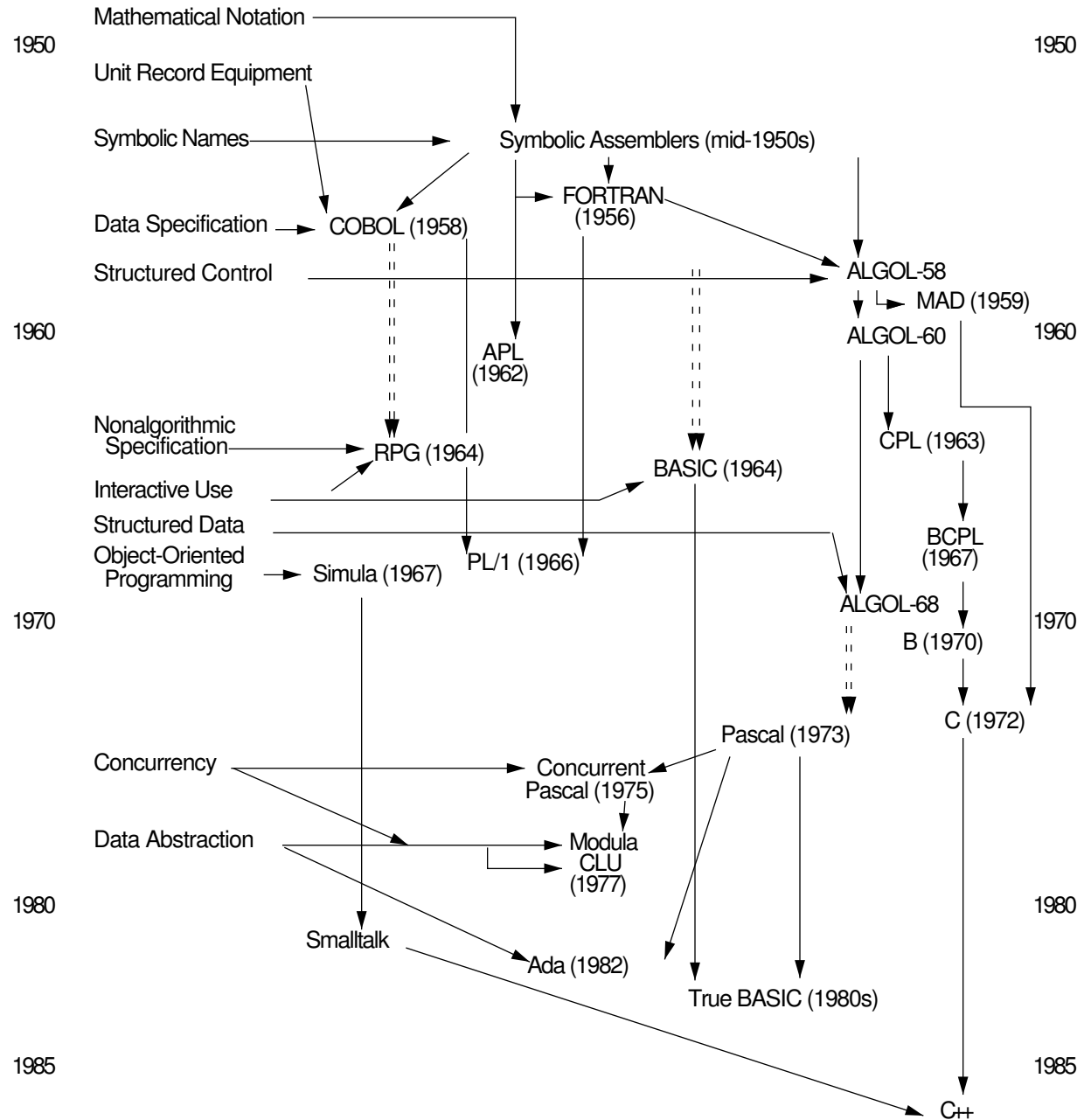
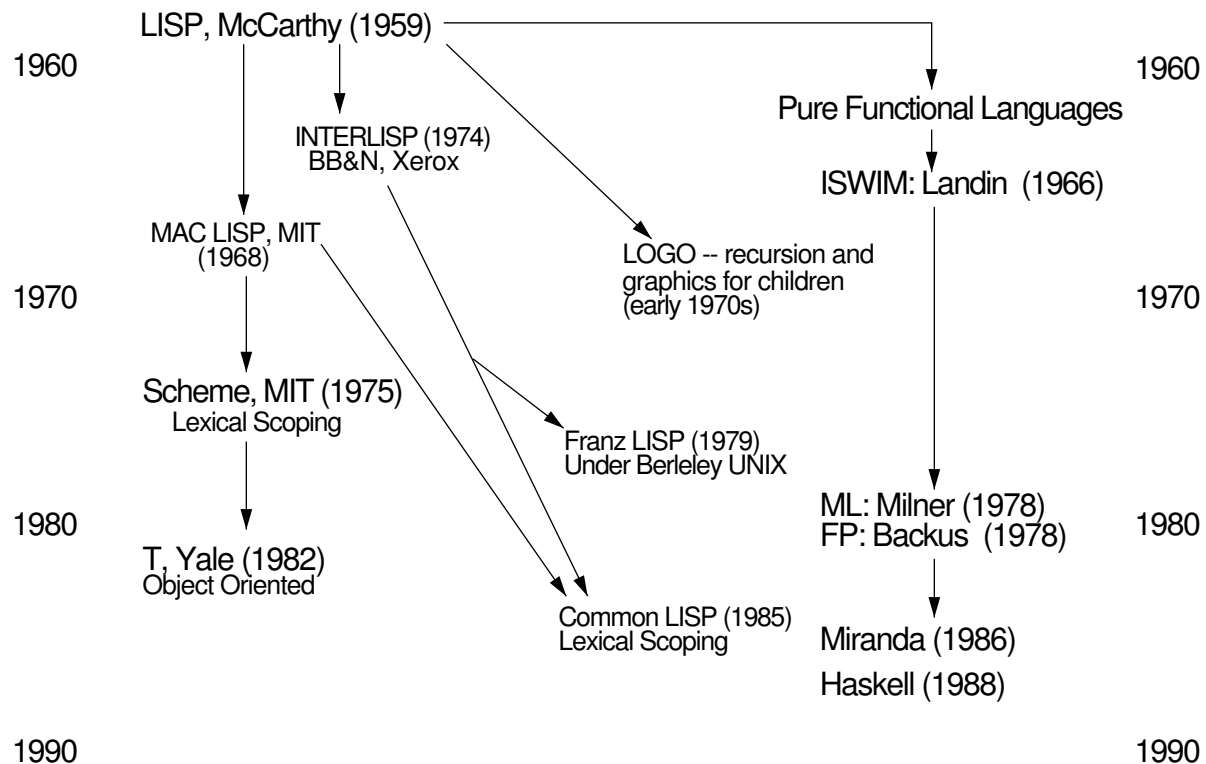


Exhibit 2.25. The development of functional languages.



Certain characteristics are commonly associated with functional languages. Most are interactive and oriented toward the list data structure. Functions are objects that can be freely manipulated, passed as parameters, composed, and so on. Permitting functions to be manipulated like other objects gives a language tremendous power. Exhibit 2.25 is a diagram of the development of this language family.

LISP and its modern lexically scoped descendants support destructive assignment and sequences of expressions, which are evaluated in order. When these features are used, these languages become “procedural”, like Pascal. These languages are, therefore, “functional” in the same sense that Pascal is “structured”. It is never necessary to use the semantically messy `GOTO` in Pascal. Any semantics that can be expressed with it can be expressed without it. Similarly, it is not necessary to use the semantically messy destructive assignment in LISP, but it is used occasionally, to achieve efficiency, when changing one part of a large data structure.

Examples: LISP, Common LISP, T, Scheme.

Functional Languages, New Style. Considerable work is now being done on developing functional languages in which sequences of statements, variables, and destructive assignment do not exist at all. Values are passed from one part of a program to another by function calls and parameter binding.

There is one fundamental difference between the old and new style functional languages. The LISP-like languages use call-by-value parameters, and these new languages use call-by-need (lazy evaluation).⁸ A parameter is not evaluated until it is needed, and its value is then kept for future use. Call-by-need is an important semantic development, permitting the use of “infinite lists”, which are objects that are part data and part program, where the program part is evaluated, as needed, to produce the next item on the list.

The terminology used to talk about these new functional programming languages is sometimes different from traditional programming terminology. A program is an unordered series of static definitions of objects, types, and functions. In *Miranda* it isn't even called a “program”, it is called a “script”. “Executing a program” is replaced by “evaluating an expression” or “reducing an expression to its normal form.” In either case, though, computation happens.

Since pure functional programming is somewhat new, it has not reached its full development yet. For example, efficient array handling has yet to be included. As the field progresses, we should find languages that are less oriented to list processing and more appropriate for modeling nonlist applications.

Examples: ML, *Miranda*, Haskell.

Parallel Languages. These contain multitasking primitives that permit a program to fork into two or more asynchronous, communicating tasks that execute some series of computations in parallel. This class of languages is becoming increasingly important as highly parallel hardware develops.

Parallel languages are being developed as extensions of other kinds of languages. One of the intended uses for them is to program highly parallel machines such as the HyperCube. There is a great deal of interest in using such machines for massive numeric applications like weather prediction and image processing. It is not surprising, therefore, that the language developed for the HyperCube resembled a merger of the established number-oriented languages, FORTRAN and APL.

There is also strong interest in parallel languages in the artificial intelligence community, where many researchers are working on neural networks. Using parallelism is natural in such disciplines. In many situations, a programmer wishes to evaluate several possible courses of action and choose the first one to reach a goal. Some of the computations may be very long and others short, and one can't predict which are which. One cannot, therefore, specify an optimal order in which to evaluate the possibilities. The best way to express this is as a parallel computation: “Evaluate all these computations in parallel, and report to me when the first one terminates”. List-oriented parallel languages will surely develop for these applications.

⁸Parameter passing is explained fully in Chapter 9.

Finally, the clean semantics of the assignment-free functional languages are significantly easier to generalize to parallel execution, and new parallel languages will certainly be developed as extensions of functional languages.

Examples: Co-Pascal, in a restricted sense. LINDA, OCCAM, FORTRAN-90.

Languages Specialized for Some Application. These languages all contain a complete general-purpose programming language as their basis and, in addition, contain a set of specialized primitives designed to make it convenient to process some particular data structure or problem area. Most contain some sophisticated and powerful higher-level commands that would require great skill and long labor to program in an unspecialized language like Pascal. An example is dBASE III which contains a full programming language similar to BASIC and, in addition, powerful screen handling and file management routines. The former expedites entry and display of information, the latter supports a complex indexed file structure in which key fields can be used to relate records in different files.

Systems programming languages must contain primitives that let the programmer manipulate the bits and bytes of the underlying machine and should be heavily standardized and widely available so that systems, once implemented, can be easily ported to other machines.

Examples: C, FORTH.

Business data processing languages must contain primitives that give fine and easy control over details of input, output, file handling, and precision of numbers. The standard floating-point representations are not adequate to provide this control, and some form of fixed-point numeric representation must be provided. The kind of printer or screen output formatting provided in FORTRAN, C, and Pascal is too clumsy and does not provide enough flexibility. A better syntax and more options must be provided. Similarly, a modern language for business data processing must have a good facility for defining screens for interactive input. A major proportion of these languages is devoted to I/O.

Higher-level commands should be included for common tasks such as table handling and sorting. Finally, the language should provide good support for file handling, including primitives for handling sequential, indexed, and random access files.

Examples: RPG (limited to sequential files), COBOL, Ada.

Data base languages contain extensive subsystems for handling *internal* files, and relationships among files. Note that this is quite independent of a good subsystem for screen and printer I/O.

Examples: dBASE, Framework, Structured Query Language (SQL).

List processing languages contain primitive definitions for a linked list data type and the important basic operations on lists. This structure has proven to be useful for artificial intelligence

programming.

Implementations must contain powerful operations for direct input and output of lists, routines for allocation of dynamic heap storage, and a *garbage collection* routine for recovery of dynamically allocated storage that is no longer accessible.

Examples: LISP, T, Scheme, Miranda.

Logic languages are interactive languages that use symbolic logic and set theory to model computation. Prolog was the first logic language and is still the best known. Its dominant characteristics define the language class. A Prolog “program” is a series of statements about logical relations that are used to establish a data base, interspersed with statements that query this data base. To evaluate a query, Prolog searches that data base for any entries that satisfy all the constraints in the query. To do this, the translator invokes an elaborate expression evaluator which performs an exhaustive search of the data base, with backtracking. Rules of logical deduction are built into the evaluator.

Thus we can classify a logic language as an interactive data base language where both operations and the data base itself are highly specialized for dealing with the language of symbolic logic and set theory. Prolog is of particular interest in the artificial intelligence community, where deductive reasoning on the basis of a set of known facts is basic to many undertakings.

Examples: HASL, FUNLOG, Templog (for temporal logic), Uniform (unifies LISP and Prolog), Fresh (combines the functional language approach with logic programming), etc.

Array processing languages contain primitives for constructing and manipulating arrays and matrices. Sophisticated control structures are built in for mapping simple operations onto arrays, for composing and decomposing arrays, and for operating on whole arrays.

Examples: APL, APL-2, VisiCalc, and Lotus.

String processing languages contain primitives for input, output, and processing of character strings. Operations include searching for and extracting substrings specified by complex patterns involving string functions. Pattern matching is a powerful higher-level operation that may involve exhaustive search by backtracking. The well-known string processing languages are SNOBOL and its modern descendant, ICON.

Typesetting languages were developed because computer typesetting is becoming an economically important task. Technical papers, books, and drawings are, increasingly, prepared for print using a computer language. A document prepared in such a language is an unreadable mixture of commands and ordinary text. The commands handle files, set type fonts, position material, and control indexing, footnotes, and glossaries. Drawings are specified in a language of their own, then integrated with text. The entire finished product is output in a language that a laser printer can handle. This book was prepared using the languages mentioned below, and a drafting package named Easydraw whose output was converted to Postscript.

Examples: Postscript, TeX, LaTeX.

Command languages are little languages frequently created by extending a system's user interface. First simple commands are provided; these are extended by permitting arguments and variations. More useful commands are added. In many cases these command interfaces develop their own syntax (usually ad hoc and fairly primitive) and truly extensive capabilities. For example, entire books have been written about UNIX shell programming. Every UNIX system includes one or several "shells" which accept, parse, and interpret commands. From these shells, the user may call system utilities and other small systems such as `grep`, `make`, and `flex`. Each one has its own syntax, switches, semantics, and defaults.

Command languages tend to be arcane. In many cases, little design effort goes into them because their creators view them as simple interfaces, not as languages.

Fourth-generation Languages . This curious name was applied to diverse systems developed in the mid-1980s. Their common property was that they all contained some powerful new control structures, statements, or functions by which you could invoke in a few words some useful action that would take many lines to program in a language like Pascal. These languages were considered, therefore, to be especially easy to learn and "user friendly", and the natural accompaniments to "fourth-generation hardware", or personal computers.

Lotus 1-2-3 and SuperCalc are good examples of fourth-generation languages. They contain a long list of commands that are very useful for creating, editing, printing, and extracting information from a two-dimensional data base called a *spreadsheet*, and subsystems for creating several kinds of graphs from that data.

HyperCard is a data base system in which it is said that you can write complex applications without writing a line of code. You construct the application with the mouse, not with the keyboard.

The designers of many fourth-generation languages viewed them as *replacements* for programming languages, not as *new* programming languages. The result is that their designs did not really profit as much as they could have from thirty years of experience in language design. Like COBOL and FORTRAN, these languages are ad hoc collections of useful operations.

The data base languages such as dBASE are also called "fourth-generation languages", and again their designers thought of them as replacements for computer languages. Unfortunately, these languages do not eliminate the need for programming. Even with lots of special report-generating features built in, users often want something a little different from the features provided. This implies a need for a general-purpose language within the fourth-generation system in which users can define their own routines. The general-purpose language included in dBASE is primitive and lacks important control structures. Until the newest version, dBASE4, procedures did not even have parameters, and when they were finally added, the implementation was unusual and clumsy.

The moral is that there is no free lunch. An adaptable system must contain a general-purpose language to cover applications not supported by predefined features. The whole system will be better if this general-purpose language is carefully designed.

2.4.2 Languages Are More Alike than Different

Viewing languages as belonging to “language families” tends to make us forget how similar all languages are. This basic similarity happens because the purpose of all languages is to communicate models from human to machine. All languages are influenced by the innate abilities and weaknesses of human beings, and are constrained by the computer’s inability to handle irreducible ambiguity. Most of the differences among languages arise from the specialized nature of the objects and tasks to be communicated using a given language.

This book is not about any particular family of languages. It is primarily about the concepts and mechanisms that underlie the design and implementation of all languages, and only secondarily about the features that distinguish one family from another. Most of all, it is not about the myriad variations in syntax used to represent the same semantics in different languages. The reader is asked to try to forget syntax and focus on the underlying elements.

Exercises

1. What are the two ways to view a program?
2. How will languages supporting these views differ?
3. What is a computer representation of an object? A process?
4. Define semantic intent. Define semantic validity. What is their importance?
5. What is the difference between explicit and implicit representation? What are the implications of each?
6. What is the difference between coherent and diffuse representation?
7. What are the advantages of coherent representation?
8. How can language design goals conflict? How can the designer resolve this problem?
9. How can restrictions imposed by the language designer both aid and hamper the programmer?
10. Why is the concept of locality of effect so important in programming language design?
11. What are the dangers involved when using global variables?
12. What is lexical coherence? Give an example of poor lexical coherence.
13. What is portability? Why does it limit flexibility?
14. Why is it difficult to classify languages according to their most salient characteristics?

15. What is a structured language? Strongly typed language? Object-oriented language? Parallel language? Fourth-generation language?
16. Why are most languages more similar than they are different? From what causes do language differences arise?
17. Discuss two aspects of a language design that make it hard to read, write, or use. Give an example of each, drawn from a language with which you are familiar.
18. Choose three languages from the following list: Smalltalk, BASIC, APL, LISP, C, Pascal, Ada. Describe one feature of each that causes some people to defend it as the “best” language for some application. Choose features that are unusual and do not occur in many languages.