

## Chapter 3

# Elements of Language

---

---

### Overview

This chapter presents elements of language, drawing correlations between English parts of speech and words in programming languages. Metalanguages allow languages to describe themselves. Basic structural units, words, sentences, paragraphs, and references, are analogous to the lexical tokens, statements, scope, and comments of programming languages.

---

---

Languages are made of words with their definitions, rules for combining the words into meaningful larger units, and metawords (words for referring to parts of the language). In this section we examine how this is true both of English and of a variety of programming languages.

### 3.1 The Parts of Speech

#### 3.1.1 Nouns

In natural languages nouns give us the ability to refer to objects. People invent names for objects so that they may catalog them and communicate information about them. Likewise, names are used for these purposes in programming languages, where they are given to program objects (functions, memory locations, etc.). A *variable declaration* is a directive to a translator to set aside storage to represent some real-world object, then give a name to that storage so that it may be accessed. Names can also be given to constants, functions, and types in most languages.

## First-Class Objects

One of the major trends throughout the thirty-five years of language design has been to strengthen and broaden the concept of “object”. In the beginning, programmers dealt directly with machine locations. Symbolic assemblers introduced the idea that these locations represented real-world data, and could be named. Originally, each object had a name and corresponded to one storage location. When arrays were introduced in FORTRAN and records in COBOL, these aggregates were viewed as collections of objects, not as objects themselves.

Several years and several languages later, arrays and records began to achieve the status of *first-class objects* that could be manipulated and processed as whole units. Languages from the early seventies, such as Pascal and C, waffled on this point, permitting some whole-object operations on aggregate objects but prohibiting others. Modern languages support aggregate-objects and permit them to be constructed, initialized, assigned to, compared, passed as arguments, and returned as results with the same ease as simple objects.

More recently, the functional object, that is, an executable piece of code, has begun to achieve first-class status in some languages, which are known as “functional languages”. The type object has been the last kind of object to achieve first-class status. A type-object describes the type of other objects and is essential in a language that supports generic code.

## Naming Objects

One of the complex aspects of programming languages that we will study in Chapter 6 involves the correspondence of names to objects. There is considerable variation among languages in the ways that names are used. In various languages a name can:

- Exist without being attached, or *bound*, to an object (LISP).
- Be bound simultaneously to different objects in different scopes (ALGOL, Pascal).
- Be bound to different types of objects at different times (APL, LISP).
- Be bound, through a pointer, to an object that no longer exists (C).

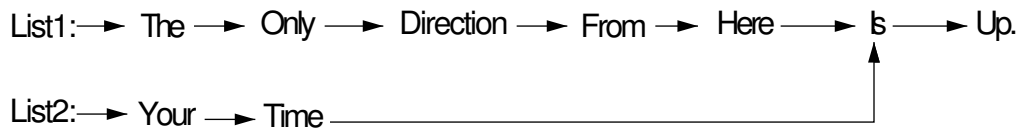
Conversely, in most languages, a single object can be bound to more than one name at a time, producing an *alias*. This occurs when a formal parameter name is bound to an actual parameter during a function call.

Finally, in many languages, the storage allocated for different objects and bound to different names can overlap. Two different list heads may share the same tail section [Exhibit 3.1].

### 3.1.2 Pronouns: Pointers

Pronouns in natural languages correspond roughly to pointers in programming languages. Both are used to refer to different objects (nouns) at different times, and both must be *bound to* (defined to refer to) some object before becoming meaningful. The most important use of pointers in

---

**Exhibit 3.1. Two overlapping objects (linked lists).**


---

programming languages is to label objects that are dynamically created. Because the number of these objects is not known to the programmer before execution time, he cannot provide names for them all, and pointers become the only way to reference them.

When a pointer is bound to an object, the address of that object is stored in space allocated for the pointer, and the pointer refers indirectly to that object. This leads to the possibility that the pointer might refer to an object that has *died*, or ceased to exist. Such a pointer is called a “dangling reference”. Using a dangling reference is a programming error and must be guarded against in some languages (e.g., C). In other languages (e.g., Pascal) this problem is minimized by imposing severe restrictions on the use of pointers. (Dangling references are covered in Section 6.3.2.)

**3.1.3 Adjectives: Data Types**

In English, adjectives describe the size, shape, and general character of objects. They correspond, in a programming language, to the many data type attributes that can be associated with an object by a declaration or by a default. In some languages, a single attribute is declared that embodies a set of properties including specifications for size, structure, and encoding [Exhibit 3.2]. In other languages, these properties are independent and are listed separately, either in variable declarations (as in COBOL) or in type declarations, as in Ada [Exhibit 3.3].

Some of the newer languages permit the programmer to define types that are related hierarchically in a tree structure. Each class of objects in the tree has well-defined properties. Each subclass has properties of its own and also inherits all the properties of the classes above it in the hierarchy. Exhibit 3.4 gives an example of such a type hierarchy in English. The root of this hierarchy is the class “vertebrate”, which is characterized by having a backbone. All subclasses “inherit” this

---

**Exhibit 3.2. Size and encoding bundled in C.**

The line below declares a number that will be represented in the computer using floating-point encoding. The actual number of bytes allocated is usually four, and the precision is approximately seven digits. This declaration is the closest parallel in C to the Ada declaration in Exhibit 3.3.

```
float price;
```

---

---

**Exhibit 3.3. Size and encoding specified separately in Ada.**

- The Ada declarations below create a new type named `REAL` and a `REAL` object, `price`.
- The use of the keyword `digits` indicates that this type is to be derived from some predefined type with floating-point encoding.
- The number seven indicates that the resulting type must have at least seven decimal digits of precision.

```
type REAL is digits 7;
price: REAL;
```

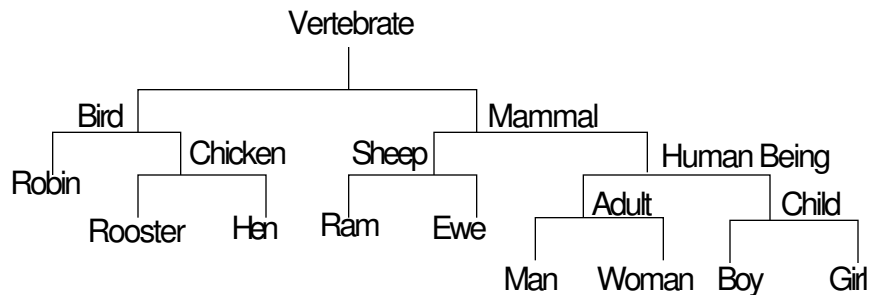
---

property. At the next level are birds, which have feathers, and mammals, which have hair. We can, therefore, conclude that robins and chickens are feathered creatures, and that human beings are hairy. Going down the tree, we see that roosters and hens inherit all properties of chickens, including being good to eat. According to the tree, adults and children are both human (although members of each subclass sometimes dispute this). Finally, at the leaf level, both male and female subclasses exist, which inherit the properties of either adults or children.

“Inheritance” means that any function defined for a superclass also applies to all subclasses. Thus if we know that constitutional rights are guaranteed for human beings, we can conclude that girls have these rights.

Using an object-oriented language such as `Smalltalk` a programmer can implement types (classes) with exactly this kind of hierarchical *inheritance* of type properties. (Chapter 17 deals with this topic more fully.)

---

**Exhibit 3.4. A type hierarchy with inheritance in English.**

### 3.1.4 Verbs

In English, verbs are words for actions or states of being. Similarly, in programming languages, we see action words such as `RETURN`, `BREAK`, `STOP`, `GOTO`, and `:=`. Procedure calls, function calls, and arithmetic operators all direct that some action should happen, and are like action verbs. Relational operators (`=`, `>`, etc.) denote states of being—they ask questions about the state of some program object or objects.

In semistandard terminology, a function is a program object that receives information through a list of arguments, performs a prescribed computation on that information, calculates some “answer”, and returns that value to the calling program. In most languages function calls can be embedded within the argument lists of other function calls, and within arithmetic expressions. Function calls are usually denoted by writing the function name followed by an appropriate series of arguments enclosed in parentheses. Expressions often contain more than one function call. In this case each language defines (or explicitly leaves undefined) the order in which the calls will be executed.<sup>1</sup>

A procedure is just like a function except that it does not return a value. Because no value results from executing the procedure, the procedure call constitutes an entire program statement and cannot be embedded in an expression or in the argument list of another call.

An operator is a predefined function whose name is often a special symbol such as “+”. Most operators require either one or two arguments, which are called *operands*. Many languages support infix notation for operators, in which the operator symbol is written between its two operands or before or after its single operand. Rules of precedence and associativity [Chapter 8, Section 8.3.2] govern the way that infix expressions are parsed, and parentheses are used, when necessary, to modify the action of these rules.

We will use the word “function” as a generic word to refer to functions, operators, and procedures when the distinctions among them are not important.

Some languages (e.g., `FORTRAN`, `Pascal`, and `Ada`) provide three different syntactic forms for operators, functions, and procedures [Exhibit 3.5]. Other languages (e.g., `LISP` and `APL`) provide only one [Exhibits 3.6 and 3.7]. To a great extent, this makes languages *appear* to be more different in structure than they are. The first impression of a programmer upon seeing his or her first `LISP` program is that `LISP` is full of parentheses, is cryptic, and has little in common with other languages. Actually, various “front ends”, or preprocessors, have been written for `LISP` that permit the programmer to write using a syntax that resembles `ALGOL` or `Pascal`. This kind of preprocessor changes only cosmetic aspects of the language syntax. It does not add power or supply kinds of statements that do not already exist. The `LISP` preprocessors do demonstrate that `LISP` and `ALGOL` have very similar semantic capabilities.

---

<sup>1</sup>This issue is discussed in Chapter 8.

---

**Exhibit 3.5. Syntax for verbs in Pascal and Ada.**

These languages, like most ALGOL-like languages, have three kinds of verbs, with distinct ways of invoking each.

**Functions:** The function name is written followed by parentheses enclosing the list of arguments.

Arguments may themselves be function calls. The call must be embedded in some larger statement, such as an assignment statement or procedure call. This is a call to a function named “push” with an embedded call on the “sin” function.

```
Success := push(rs, sin(x));
```

**Procedures:** A procedure call constitutes an entire program statement. The procedure name is written followed by parentheses enclosing the list of arguments, which may be function calls.

This is a call on Pascal's output procedure, with an embedded function call:

```
Writeln (Success, sin(x));
```

**Operators:** An operator is written between its operands, and several operators may be combined to form an expression. Operator-expressions are legal in any context in which function calls are legal.

```
Success := push(rs, (x+y)/(x-y));
```

---



---

**Exhibit 3.6. Syntax for verbs in LISP.**

LISP has only one class of verb: functions. There *are* no procedures in LISP, as all functions return a value. In a function call, the function name and the arguments are enclosed in parentheses (first line below). Arithmetic operators are also written as function calls (second line).

```
(myfun arg1 arg2 arg3)
(+ B 1)
```

---

**Exhibit 3.7. Syntax for verbs in APL.**

APL provides a syntax for applying operators but not for function calls or procedure calls. Operators come in three varieties: dyadic (having two arguments), monadic (having one argument), and niladic (having no arguments).

- Dyadic operators are written between their operands. Line [1] below shows “+” being used to add the vector (5 3) to B and add that result to A. (APL expressions are evaluated right-to-left.) Variables A and B might be scalars or length two vectors. The result is a length two vector.
- Monadic operators are written to the left of their operands. Line [2] shows the monadic operator “|”, or absolute value.
- Line [3] shows a call on a *niladic operator*, the read-input operator, “□”. The value read is stored in A.

```
[1] A + 5 3 + B
[2] | A
[3] A ← □
```

The programmer may define new functions but may not use more than two arguments for those functions. Function calls are written using the syntax for operators. Thus a dyadic programmer-defined function named “FUN” would be called by writing:

```
A FUN B
```

When a function requires more than two arguments, they must be packed or encoded into two bunches, sent to the function, then unpacked or decoded within the function. This is awkward and not very elegant.

**The Domain of a Verb**

The definition of a verb in English always includes an indication of the domain of the verb, that is, the nouns with which that verb can meaningfully be used. A dictionary provides this information, either implicitly or explicitly, as part of the definition of each verb [Exhibit 3.8].

Similarly, the domain of a programming language verb is normally specified when it is defined. This specification is part of the program in some languages, part of the documentation in others. The *domain of a function* is defined in most languages by a function header, which is part of the function definition. A header specifies the number of the objects required for the function to operate and the formal names by which those parameters will be known. Languages that implement strong typing also require the types of the parameters to be specified in the header. This information is used to ensure that the function is applied meaningfully, to objects of the correct types [Exhibit 3.9].

---

**Exhibit 3.8. The domain of a verb in English.****Verb:** Cry**Definition for the verb “cry”,** paraphrased from the dictionary.<sup>a</sup>

1. To make inarticulate sobbing sounds expressing grief, sorrow, or pain.
2. To weep, or shed tears.
3. To shout, or shout out.
4. To utter a characteristic sound or call (used of an animal).

The domain is defined in definitions (1) through (3) by stating that the object/creature that cries must be able to sob, express feelings, weep, or shout. Definition (4) explicitly states that the domain is an animal. Thus all of the following things can “cry”: human beings (by definitions 1, 2, 3), geese (4), and baby dolls (2).

---

<sup>a</sup>Morris [1969], p. 319.

---

The *range of a function* is the set of objects that may be the result of that function. This must also be specified in the function header (as in Pascal) or by default (as in C) in languages that implement type checking.

**3.1.5 Prepositions and Conjunctions**

In English we distinguish among the parts of speech used to denote time, position, conditionals, and the relationship of phrases in a sentence. Each programming language contains a small number of such words, used analogously to delimit phrases and denote choices and repetition (WHILE, ELSE, BY, CASE, etc.). The exact words differ from language to language. Grammatical rules state how these words may be combined with phrases and statements to form meaningful units.

---

**Exhibit 3.9. The domains of some Pascal functions.**

Predefined functions	Domains
chr	An integer between 0 and 127.
ord	Any object of an enumerated type.
trunc	A real number.

**A user-defined function header** : FUNCTION search (N:name; L:list ): list;

The domain of “search” is pairs of objects, one of type “name”, the other of type “list”. The result of “search” is a “list”; its range is, therefore, the type “list”.

---

By themselves these words have little meaning, and we will deal with them in Chapter 10, where we examine control structures.

## 3.2 The Metalanguage

A language needs ways to denote its structural units and to refer to its own parts. English has sentences, paragraphs, essays, and the like, each with lexical conventions that identify the unit and mark its beginning and end. Natural languages are also able to refer to these units and to the words that comprise the language, as in phrases such as “the paragraph below”, and “USA is an abbreviation for the United States of America”. These parts of a language that permit it to talk about itself are called a *metalanguage*. The metalanguage that accompanies most programming languages consists of an assortment of syntactic delimiters, metawords, and ways to refer to structural units. We consider definitions of the basic structural units to be part of the metalanguage also.

### 3.2.1 Words: Lexical Tokens

The smallest unit of any written language is the *lexical token*—the mark or series of marks that denote one symbol or word in the language. To understand a communication, first the tokens must be identified, then each one and their overall arrangement must be interpreted to arrive at the meaning of the communication. Analogously, one must separate the sounds of a spoken sentence into tokens before it can be comprehended. Sometimes it is a nontrivial task to separate the string of written marks or spoken sounds into tokens, as anyone knows who has spent a day in a foreign country.

This same process must be applied to computer programs. A human reader or a compiler must first perform a *lexical analysis* of the code before beginning to understand the meaning. The portion of the compiler that does this task is called the *lexer*.

In some languages lexical analysis is trivially simple. This is true in FORTH, which requires every lexical token to be *delimited* (separated from every other token) by one or more spaces. Assembly languages frequently define fixed columns for operation codes and require operands to be separated by commas. Operating system command shells usually call for the use of spaces and a half dozen punctuation marks which are tokens themselves and also delimit other tokens. Such simple languages are easy to lexically analyze, or *lex*. Not all programming languages are so simple, though, and we will examine the common lexical conventions and their effects on language.

The lexical rules of most languages define the lexical forms for a variety of token types:

- Names (predefined and user-defined)
- Special symbols
- Numeric literals
- Single-character literals

- Multiple-character string literals

These rules are stated as part of the formal definition of every programming language. A lexer for a language is commonly produced by feeding these rules to a program called a *lexer generator*, whose output is a program (the lexer) that can perform lexical analysis on a source text string according to the given rules. The lexer is the first phase of a compiler. Its role in the compiling process is illustrated in Exhibit 4.3.

Much of the feeling and appearance of a language is a side effect of the rules for forming tokens. The most common rules for delimiting tokens are stated below. They reflect the rules of Pascal, C, and Ada.

- Special symbols are characters or character strings that are nonalphabetic and nonnumeric. Examples are “;”, “+”, and “:=”. They are all predefined by the language syntax. No new special symbols may be defined by the programmer.
- Names must start with an alphabetic character and must not contain anything except letters, digits, and (sometimes) the “\_” symbol.
- Everything that starts with a letter is a name.
- Names end with a space or a special symbol.
- Special symbols generally alternate with names and literals. Where two special symbols or two names are adjacent, they must be separated by a space.
- Numeric literals start with a digit, a “+”, or a “-”. They may contain digits, “.”, and “E” (for exponent). Any other character ends the literal.
- Single-character literals and multiple-character strings are enclosed in matching single or double quotes. If, as in C, a single character has different semantics from a string of length 1, then single quotes may be used to delimit one and double quotes used for the other.

Note that spaces are used to delimit some but not all tokens. This permits the programmer to write arithmetic expressions such as “ $a*(b+c)/d$ ” the way a mathematician would write them. If we insisted on a delimiter (such as a space) after every token, the expression would have to be written “ $a * ( b + c ) / d$ ”, which most programmers would consider to be onerous and unnatural.

Spaces *are* required to delimit arithmetic operators in COBOL. The above expression in COBOL would be written “ $a * ( b + c ) / d$ ”. This awkwardness is one of the reasons that programmers are uncomfortable using COBOL for numeric applications. The reason for this requirement is that the “-” character is ambiguous: COBOL’s lexical rules permit “-” to be used as a hyphen in variable names, for example, “hourly-rate-in”. Long, descriptive variable names greatly enhance the readability of programs.

Hyphenated variable names have existed in COBOL from the beginning. When COBOL was extended at a later date to permit the use of arithmetic expressions an ambiguity arose: the hyphen character and the subtraction operator were the same character. One way to avoid this problem is to use different characters for the two purposes. Modern languages use the “-” for subtraction and the underbar, “\_”, *which has no other function in the language*, to achieve readability.

As you can see, the rules for delimiting tokens can be complex, and they do have varied repercussions. The three important issues here are:

- Code should be readable.
- The language must be translatable and, preferably, easy to lex.
- It is preferable to use the same conventions as are used in English and/or mathematical notation.

The examples given show that a familiar, readable language may contain an ambiguous use of symbols. A few language designers have chosen to sacrifice familiarity and readability altogether in order to achieve lexical simplicity. LISP, APL, and FORTH all have simpler lexical and syntactic rules, and all are considered unreadable by some programmers because of the conflict between their prior experience and the lexical and syntactic forms of the language.

Let us examine the simple lexical rule in FORTH and its effects. In other languages the decision was made to permit arithmetic expressions to be written without delimiters between the variable names and the operators. A direct consequence is that special symbols (nonalphabetic, nonnumeric, and nonunderbar) must be prohibited in variable names. It may seem natural to prohibit the use of characters like “+” and “(” in a name, but it is not at all necessary.

FORTH requires one or more space characters or carriage returns between every pair of tokens, and because of this rule, it can permit special characters to be used in identifiers. It makes no distinction between user-defined names and predefined tokens: either may contain any character that can be typed and displayed. The string “#\$\$” could be used as a variable or function name if the programmer so desired. The token “ab\*” could never be confused with an arithmetic problem because the corresponding arithmetic problem, “a b \*”, contains three tokens separated by spaces. Thus the programmer, having a much larger alphabet to use, is far freer to invent brief, meaningful names. For example, one might use “a+” to name a function that increments its argument (a variable) by the value of a.

Lexical analysis is trivially easy in FORTH. Since its lexical rules treat all printing characters the same way and do not distinguish between alphabetic characters and punctuation marks, FORTH needs only three classes of lexical tokens:

- Names (predefined or user-defined).
- Numeric literals.
- String literals. These can appear only after the string output command, which is “. ” (pronounced “dot-quote”). A string literal is terminated by the next “” (pronounced “quote”).

These three token types correspond to semantically distinct classes of objects that the interpreter handles in distinct ways. Names are to be looked up in the dictionary and executed. Numeric literals are to be converted to binary and put on the stack. String literals are to be copied to the output stream. The lexical rules of the language thus correspond directly to its semantics, and the interpreter is very short and simple.

The effect of these lexical rules on people should also be noted. Although the rules are simple and easy to learn, a programmer accustomed to the conventions in other languages has a hard time learning to treat the space character as important.

### 3.2.2 Sentences: Statements

The earliest high-level languages reflected the linguistic idea of sentences: a FORTRAN or COBOL program is a series of sentencelike statements.<sup>2</sup> COBOL statements even end in periods. Most statements, like sentences, specify an action to perform and some object or objects on which to perform the action. A language is called “procedural”, if a program is a sequence statements, grouped into procedures, to be carried out using the objects specified.

In the late 1950s when FORTRAN and COBOL were developed, the punched card was the dominant medium for communication from human to computer. Programs, commands to the operating system, and data were all punched on cards. To compile and (one hoped) run a program, the programmer constructed a “deck” usually consisting of:

- An ID control card, specifying time limits for the compilation and run.<sup>3</sup>
- A control card requesting compilation, an object program listing, an error listing, and a memory map.<sup>4</sup>
- A series of cards containing the program.
- A control card requesting loading and linking of the object program.
- A control card requesting a run and a core dump<sup>5</sup> of the executable program.

---

<sup>2</sup>Caution: In a discussion of formal grammars and parsing, the term “sentence” is often used to mean the entire program, not just one statement.

<sup>3</sup>Historical note: Some of the items on the control cards are hard to understand in today’s environment. Limiting the time that a job would be allowed to run (using a job time limit) was important then because computer time was very costly. In 1962, time on the IBM 704 (a machine comparable in power to a Commodore 64) cost \$600 per hour at the University of Michigan. For comparison, Porterhouse steak cost about \$1 per pound. Short time limits were specified so that infinite loops would be terminated by the system as soon as possible.

<sup>4</sup>The memory map listed all variable names and their memory addresses. The map, object listing, and core (memory) dump together were indispensable aides to debugging. They permitted the programmer to reconstruct the execution of the program manually.

<sup>5</sup>Most debugging was done in those days by carefully analyzing the contents of a core (memory) dump. The kind of trial and error debugging that we use today was impractical because turnaround time for a trial run was rarely less than a few hours and sometimes was measured in days. In order to glean as much information as possible from each run, the programmer would analyze the core dump using the memory maps produced by the compiler and linker.

---

**Exhibit 3.10. Field definitions in FORTRAN.**

Columns	Use in a FORTRAN program
1	A “C” or a “*” here indicates a comment line.
1–5	Statement labels
6	Statement continuation mark
7–72	The statement itself
73–80	Programmer ID and sequence numbers.

**End of statement.** At end of line, unless column 6 on the *next* card is punched to indicate a continuation of the statement.

**Indenting convention.** Start every statement in column 7. (Indenting is not generally used.)

---

- A control card marking the beginning of the data.
- A series of cards containing the data.
- A JOB END control card marking the end of the deck.

Control cards had a special character in column 1 by which they could be recognized. Because a deck of cards could easily become disordered by being dropped, columns 73 through 80 were conventionally reserved for identification and sequence numbers. The JOB END card had a different special mark. This made it easy for the operating system to abort remaining segments of a job after a fatal error was discovered during compilation or linking.

Because punched cards were used for programs as well as data, the physical characteristics of the card strongly influenced certain aspects of the early languages. The program statement, which was the natural program unit, became tightly associated with the 80-column card, which was the natural media unit. Many programmers wrote their code on printed coding forms, which looked like graph paper with darker lines marking the fields. This helped keypunch operators type things in the correct columns.

The designers of the FORTRAN language felt that most FORTRAN statements would fit on one line and so chose to require that each statement be on a separate card. The occasional statement that was too long to fit could be continued on another card by placing a character in column six of the second card. Exhibit 3.10 lists the fields that were defined for a FORTRAN card.

COBOL was also an early fixed-format language, with similar but different fixed fields. Due to the much longer variable names permitted in COBOL, and the wordier and more complex syntax,

---

The programmer would trace execution of the program step by step and compare the actual contents of each memory location to what was supposed to be there. Needless to say, this was slow, difficult, and beyond the capabilities of many people. Modern advances have made computing much more accessible.

many statements would not fit on one line. A convention that imitated English was introduced: the end of each statement was marked by a period. A group of statements that would be executed sequentially was called a “paragraph”, and each paragraph was given an alphanumeric label. Within columns 13–72, indenting was commonly used to clarify the meaning of the statements.

Two inventions in the late 1960s combined to make the use of punched cards for programs obsolete. The remote-access terminal and the on-line, disk-based file system made it both unnecessary and impractical to use punched cards. Languages that were designed after this I/O revolution reflect the changes in the equipment used. Fixed fields disappeared, the use of indentation to clarify program structure became universal, and a character such as “;” was used to separate statements or terminate each statement.<sup>6</sup>

### 3.2.3 Larger Program Units: Scope

English prepositions and conjunctions commonly control a single phrase or clause. When a larger scope of influence is needed in English, we indicate that the word pertains to a paragraph. In programming languages, units that correspond to such paragraphs are called *scopes* and are commonly marked by a pair of matched opening and closing marks. Exhibits 3.11, 3.13, and 3.15 show the tremendous variety of indicators used to mark the beginning and end of a scope.

In FORTRAN the concept of “scope” was not well abstracted, and scopes were indicated in a variety of ways, depending on the context. As new statement types were added to the language over the years, new ways were introduced to indicate their scopes. FORTRAN uses five distinct ways to delimit the scopes of the `DATA` statement, `DO` loop, `implied DO` loop, `logical IF` (true action only), and `block IF` (true and false actions) [Exhibit 3.11]. This nonuniformity of syntax does not occur in the newer languages.

Two different kinds of ways to end scopes are shown in Exhibit 3.11. The labeled statement at the end of a `DO` scope ends a specific `DO`. Each `DO` statement specifies the statement label of the line which terminates its scope. (Two `DO`s are allowed to name the same label, but that is not relevant here.) We say that `DO` has a *labeled scope*. In contrast, all `block IF` statements are ended by identical `ENDIF` lines. Thus an `ENDIF` could end any `block IF` statement. We say that `block IF` statements have *unlabeled scopes*.

The rules of FORTRAN do not permit either `DO` scopes or `block IF` scopes to overlap partially. That is, if the beginning of one of these scopes, say B, comes between the beginning and end of another scope, say A, then the end of scope B must come before the end of scope A. Legal and illegal nestings of labeled scopes are shown in Exhibit 3.12.

All languages designed since 1965 embody the abstraction “scope”. That is, the language supplies a single way to delimit a paragraph, and that way is used uniformly wherever a scope is needed in the syntax, for example, with `THEN`, `ELSE`, `WHILE`, `DO`, and so on. For many languages, this is accomplished by having a single pair of symbols for begin-scope and end-scope, which are

---

<sup>6</sup>Most languages did not use the “.” as a statement-mark because periods are used for several other purposes (decimal points and record part selection), and any syntax becomes hard to translate when symbols become heavily ambiguous.

**Exhibit 3.11. Scope delimiters in FORTRAN.**

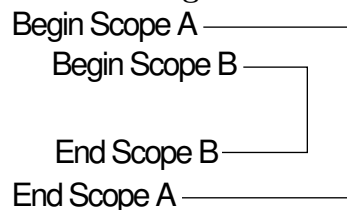
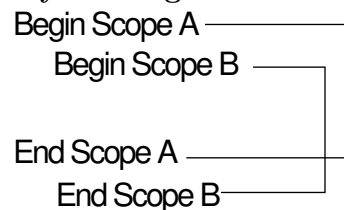
The following program contains an example of each linguistic unit that has an associated scope. The line numbers at the left key the statements to the descriptions, in the table that follows, of the scope indicators used.

```

1      INTEGER A, B, C(20), I
2      DATA A, B /31, 42/
3      READ* A, B, ( C(I), I=1,10)
4      DO 80 I= 1, 10
5      IF (C(I) .LT. 0) C(I+10)=0
6      IF (C(I) .LT. 100) THEN
7      C(I+10) = 2 * C(I)
8      ELSE
9      C(I+10) = C(I)/2
10     ENDIF
11 80  CONTINUE
12     END

```

Scope of	Begins at	Ends at	Line #s
Dimension list	“(” after array name	The next “)”	1
DATA values	First “/”	Second “/”	2
Implied DO	“(” in I/O list	I/O loop control	3
Subscript list	“(” after array name	Matching “)”	3
DO loop	Line following DO	Statement with DO label	5 - 11
Logical IF	After ((condition))	End of line	5
Block IF (true)	After THEN	ELSE, ELSEIF, or ENDIF	7
Block IF (false)	After ELSEIF or ELSE	ELSE, ELSEIF, or ENDIF	9

**Exhibit 3.12. Labeled scopes.****Correct Nesting:****Faulty Nesting:**

**Exhibit 3.13. Tokens used to delimit program scopes.**


---

Language	Beginning of Scope	End of Scope
C	{	}
LISP	(	)
Pascal	BEGIN	END
	RECORD	END
	CASE	END
PL/1	DO;	END;
	DO <loop control>;	END;

---

used to delimit any kind of scope [Exhibit 3.13]. In these languages it is not possible to nest scopes improperly because the compiler will always interpret the nesting in the legal way. A compiler will match each end-scope to the nearest unmatched begin-scope. This design is attractive because it produces a language that is simpler to learn and simpler to translate.

If an end-scope is omitted, the next one will be used to terminate the open scope regardless of the programmer's intent [Exhibit 3.14]. Thus an end-scope that was intended to terminate an IF may instead be used to terminate a loop or a subprogram. A compiler error comment may appear on the next line because the program element written there is in an illegal context, or error comments may not appear until the translator reaches the end of the program and finds that the wrong number of end-scopes was included. If an extra end-scope appears somewhere else, improper nesting might not be detected at all.

Using one uniform end-scope indicator has the severe disadvantage that a nesting error may not be identified as a syntactic error, but become a logical error which is harder to identify and correct. The programmer has one fewer tool for communicating semantics to the compiler, and the compiler has one fewer way to help the programmer achieve semantic validity. Many experienced programmers use comments to indicate which end-scope belongs to each begin-scope. This practice makes programs more readable and therefore easier to debug, but of course does not help the compiler.

A third, intermediate way to handle scope delimiters occurs in *Ada*. Unlike *Pascal*, each kind of scope has a distinct end-scope marker. Procedures and blocks and labeled loops have fully labeled end-scopes. Unlike *FORTRAN*, a uniform syntax was introduced for delimiting and labeling scopes. An end-scope marker is the word "end" followed by the word and label, if any, associated with the beginning of the scope [Exhibit 3.15].

It is possible, in *Ada*, for the compiler to detect many (but not all) improperly nested scopes and often to correctly deduce where an end-scope has been omitted. This is important, since a misplaced or forgotten end-scope is one of the most common kinds of compile-time errors.

A good technique for avoiding errors with paired delimiters is to type the END marker when the BEGIN is typed, and position the cursor between them. This is the idea behind the *structured*

**Exhibit 3.14. Nested unlabeled scopes in Pascal.**

Begin Scope	—	i := 0;
Begin Scope		
	—	IF a mod 7 = 0 THEN BEGIN
		i := i + 1;
		writeln (i, a)
End Scope	—	END
End Scope	—	END

*editors.* When the programmer types the beginning of a multipart control unit, the editor inserts all the keywords and scope markers necessary to complete that unit meaningfully. This prevents beginners and forgetful experts from creating malformed scopes.

### 3.2.4 Comments

Footnotes and bibliographic citations in English permit us to convey general information about the text. Analogously, comments, interspersed with program words, let us provide information about a program that is not part of the program. With comments, as with statements, we have the problem of identifying both the beginning and end of the unit. Older languages (COBOL, FORTRAN) generally restrict comments to separate lines, begun by a specific comment mark in a fixed position on the line [Exhibit 3.16]. This convention was natural when programs were typed on punch cards. At the same time it is a severe restriction because it prohibits the use of brief comments placed out of the way visually. It therefore limits the usefulness of comments to explain obscure items that are embedded in the code.

The newer languages permit comments and code to be interspersed more freely. In these

**Exhibit 3.15. Lexical scope delimiters in Ada.**

Begin-scope markers	End-scope markers
<block_name>: <declarations> BEGIN	END <block_name>
PROCEDURE <proc_name>	END <proc_name>;
LOOP	END LOOP;
<label>:LOOP	END LOOP <label>;
CASE	END CASE
IF <condition> THEN or	
ELSIF <condition> THEN	ELSIF, ELSE, or END IF
ELSE	END IF

**Exhibit 3.16. Comment lines in older languages.**

In these languages comments must be placed on a separate line, below or above the code to which they apply.

Language	Comment line is marked by
FORTRAN	A “C” in column 1
COBOL	A “*” in column 7
original APL	The “lamp” symbol: $\circ$ at the beginning of a line
BASIC	REM at the beginning of a line

languages, statements can be broken onto multiple lines and combined freely with short comments in order to do a superior job of clarifying the intent of the programmer. Both the beginning and end of a comment are marked [Exhibit 3.17]. Comments are permitted to appear anywhere within a program, even in the middle of a statement.

A nearly universal convention is to place the code on the left part of the page and comments on the right. Comments are used to document the semantic intent of variables, parameters, and unusual program actions, and to clarify which end-scope marker is supposed to match each begin-scope marker. Whole-line comments are used to mark and document the beginning of each program module, greatly assisting the programmer’s eye in finding his or her way through the pages of code. Some comments span several lines, in which case only the beginning of the first line and end of the last line need begin- and end-comment marks. In spite of this, many programmers mark the beginning and end of every line because it is aesthetically nicer and sets the comment apart from code.

With all the advantages of these partial-line comments, one real disadvantage was introduced by permitting begin-comment and end-comment marks to appear anywhere within the code. It is not unusual for an end-comment mark to be omitted or typed incorrectly [Exhibit 3.18]. In this case all the program statements up to the end of the next comment are taken to be part of the nonterminated comment and are simply “swallowed up” by the comment.

**Exhibit 3.17. Comment beginning and end delimiters.**

These languages permit a comment and program code to be placed on the same line. Both the beginning and end of the comment is marked.

Language	Comments are delimited by
C	<code>/* ... */</code>
PL/1	<code>/* ... */</code>
Pascal	<code>(* ... *)</code> or <code>{ ... }</code>
FORTH	<code>( ... )</code>

**Exhibit 3.18. An incomplete comment swallowing an instruction.**

The following Pascal code appears to be ok at first glance, but because of the mistyped end-comment mark, the computation for `tot_age` will be omitted. The result will be a list of family members with the wrong average age!

A “`person_list`” is an array of person cells, each containing a name and an age.

```

PROCEDURE average_age(p: person_list);
VAR famsize, tot_age, k:integer;
BEGIN
  readln(famsize); (* Get the number of family members to process.*)
  tot_age := 0;
  FOR k := 1 TO famsize DO BEGIN
    writeln( p[k].name ); (* Start with oldest family member. * )
    tot_age := tot_age + p[k].age; (* Sum ages for average. *)
  END;
  writeln('Average age of family = ', tot_age/famsize)
END;
```

The translator may not ever detect this violation of the programmer’s intent. If the next comment is relatively near, and no end-scope markers are swallowed up by the comment, the program may compile with no errors but run strangely. This can be a very difficult error to debug, since the program looks correct but its behavior is inconsistent with its appearance! Eventually the programmer will decide that he or she has clearly written a correct instruction that the compiler seems to have ignored. Since compilers do not just ignore code, this does not make sense. Finally the programmer notices that the end-comment mark that should be at the end of some prior line is missing.

This problem is an example of the cost of over-generality. Limiting comments to separate lines was too restrictive, that is, not general enough. Permitting them to begin and end anywhere on a line, though, is more general than is needed or desired. Even in languages that permit this, comments usually occupy either a full line or the right end of a line. A more desirable implementation of comments would match the comment-scope and comment placement rules with the actual conventions that most programmers use, which are:

- whole-line comments
- partial-line comments placed on the right side of the page
- multiple-line comments

Thus comments should be permitted to occur on the right end of any line, but they might as well be terminated by the end of the line. Permitting multiple-line comments to be written is important,

**Exhibit 3.19. Comments terminating at end of line.**

These languages permit comments to occupy entire lines or the right end of any line. Comments start with the comment-begin mark listed and extend to the carriage return on the right end of the line. (TeX is a text processing language for typesetting mathematical text and formulas. It was used to produce this book.)

Language	Comment-begin mark
Ada	--
LISP	; (This varies among implementations.)
TeX	%
UNIX command shell	#
C++	//

but it is not a big burden to mark the beginning of every comment line, as many programmers do anyway to improve the appearance of their programs. The payoff for accepting this small restriction is that the end-of-line mark can be used as a comment-end mark. Since programmers do not forget to put carriage returns in their programs, comments can no longer swallow up entire chunks of code. Some languages that have adopted this convention are listed in Exhibit 3.19.

Some languages support two kinds of comment delimiters. This permits the programmer to use the partial-line variety to delimit explanatory comments. The second kind of delimiter (with matched begin-comment and end-comment symbols) is reserved for use during debugging, when the programmer often wants to “comment out”, temporarily, large sections of code.

**3.2.5 Naming Parts of a Program**

In order to refer to the parts of a program, we need meta-words for those parts and for whatever actions are permitted. For example, C permits parts of a program to be stored in separate files and brought into the compiler together by using “#include <file\_name>”. The file name is a metaword denoting a section of the program, and “#include” is a metaword for the action of combining it with another section.

Most procedural languages provide a GOTO instruction which transfers control to a specific labeled statement somewhere in the program. The statement label, whether symbolic or numeric, is thus a metaword that refers to a part of the program. Since the role of statement labels cannot be fully understood apart from the control structures that use them, labels are discussed with the GOTO command in Section 11.1.

**3.2.6 Metawords That Let the Programmer Extend the Language**

There are several levels on which a language may be extended. One might extend:

- The list of defined words (nouns, verbs, adjectives).

- The syntax but not the semantics, thus providing alternative ways of writing the same meanings one could write without the extension.
- The actual semantics of the language, with a corresponding extension either of the syntax or of the list of defined words recognized by the compiler.

Languages that permit the third kind of extension are rare because extending the semantics requires changing the translator to handle a new category of objects. Semantic extension is discussed in the next chapter.

### Extending the Vocabulary

Every declaration extends the language in the sense that it permits a compiler to “understand” new words. Normally we are only permitted to declare a few kinds of things: nouns (variables, constants, file names), verbs (functions and procedures), and sometimes adjectives (type names) and metawords (labels). We cannot normally declare new syntactic words or new words such as “array”. The compiler maintains one combined list or several separate lists of these definitions. This list is usually called the “symbol table”, but it is actually called the “dictionary” in FORTH. New symbols added to this list always belong to some previously defined syntactic category with semantics defined by the compiler.

Each category of symbol that can be declared must have its own keyword or syntactic marker by which the compiler can recognize that a definition of a new symbol follows. Words such as `TYPE`, `CONST`, and `PROCEDURE` in Pascal and `INTEGER` and `FUNCTION` in FORTRAN are metawords that mean, in part, “extend the language by putting the symbols that follow into the symbol table.”

As compiler technology has developed and languages have become bigger and more sophisticated, more kinds of declarable symbols have been added to languages. The original BASIC permitted no declarations: all two-letter variable names could be used without declaration, and no other symbols, even subroutine names, could be defined. The newest versions of BASIC permit use of longer variable names, names for subroutines, and symbolic labels. FORTRAN, developed in 1954–1958, permitted declaration of names for variables and functions. FORTRAN 77 also permits declaration of names for constants and COMMON blocks. ALGOL-68 supported type declarations as a separate abstraction, not as part of some data object. Pascal, published in 1971, brought type declarations into widespread use. Modula, a newer language devised by the author of Pascal, permits declaration and naming of semantically separate modules. Ada, one of the newest languages in commercial use, permits declaration of several things missing in Pascal, including the range and precision of real variables, support for concurrent tasks, and program modules called “generic packages” which contain data and function declarations with type parameters.

---

**Exhibit 3.20. Definition of a simple macro in C.**

In C, macro definitions start with the word `#define`, followed by the macro name. The string to the right of the macro name defines the meaning of the name.

The `#define` statements below make the apparent syntax of C more like Pascal. They permit the faithful Pascal programmer to use the familiar scoping words `BEGIN` and `END` in a C program. (These words are not normally part of the C language.) During preprocessing, `BEGIN` will be replaced by “{” and `END` will be replaced by “}”.

```
#define BEGIN {
#define END }
```

---

**Syntactic Extension without Semantic Extension**

Some languages contain a macro facility (in C, it is part of the preprocessor).<sup>7</sup> This permits the programmer to define short names for frequently used expressions. A macro definition consists of a name and a string of characters that becomes the meaning of the name [Exhibit 3.20]. To use a macro, the programmer writes its name, like a shorthand notation, in the program wherever that string of characters is to be inserted [Exhibit 3.21].

A preprocessor scans the source program, searching for macro names, before the program is

---

<sup>7</sup>The C preprocessor supports various compiler directives as well as a general macro facility.

---

**Exhibit 3.21. Use of a simple macro in C.**

**Macro Calls.** The simple macros defined in Exhibit 3.20 are called in the following code fragment. Unfortunately, the new scope symbols, `BEGIN` and `END`, and the old ones, “{” and “}”, are now interchangeable. Our programmer can write the following code, defining two well-nested scopes. It would work, but it isn’t “pretty” or clear.

```
BEGIN    x = y+2;
         if (x < 100) { x += k; y = 0; END
         else x = 0; }
```

**Macro Expansion.** During macro expansion the macro call is replaced by the defining string. The C translator never sees the word `BEGIN`.

```
{    x = y+2;
   if (x < 100) { x += k; y = 0; }
   else x = 0; }
```

---

parsed. These macro names are replaced by the defining strings. The expanded program is then parsed and compiled. Thus the preprocessor commands and macro calls form a separate, primitive, language. They are identified, expanded, and eliminated before the parser for the main language even begins its work.

The syntax for a macro language, even one with macro parameters, is always simple. However, piggy-backing a macro language on top of a general programming language causes some complications. The source code will be processed by two translators, and their relationship must be made clear. Issues such as the relationship of macro calls to comments or quoted strings must be settled.

In C, preprocessor commands and macro definitions start with a “#” in column 1.<sup>8</sup> This distinguishes them from source code intended for the compiler. Custom (but not compiler rules) dictates that macro names be typed in uppercase characters and program identifiers in lowercase. Case does not matter to the translator, but this custom helps the programmer read the code.

Macro *calls* are harder to identify than macro *definitions*, since they may be embedded anywhere in the code, including within a macro definition. Macro names, like program identifiers, are variable-length strings that need to be identified and separated from other symbols. Lexical analysis must, therefore, be done before macro expansion. Since the result of expansion is a source string, lexical analysis must be done again after expansion. Since macro definitions may contain macro calls, the result of macro expansion must be rescanned for more macro calls. Control must thus pass back and forth between the lexer and the macro facility. The lexical rules for the preprocessor language are necessarily the same as the rules for the main language.

In the original definition of C, the relationship among the lexer, preprocessor, and parser was not completely defined. Existing C translators thus do different things with macros, and all are “correct” by the language definition. Some C translators simply insert the expanded macro text back into the source text without inserting any blanks or delimiters. The effect is that characters outside a macro can become adjacent to characters produced by the macro expansion. The program line containing the expanded macro is then sent back to the lexer. When the lexer processes this, it forms a single symbol from the two character strings. This “gluing” action can produce strange and unexpected results.

The ANSI standard for C has clarified this situation. It states that no symbol can bridge a macro boundary. Lexical analysis on the original source string is done, and symbols are identified, before macro expansion. The source string that defines the macro can also be lexed before expansion, since characters in it can never be joined with characters outside it. These rules “clean up” a messy situation. The result of expanding a macro still must be rescanned for more macro calls, but it does not need to be re-lexed. The definition and call of a macro within a macro are illustrated in Exhibits 3.22 and 3.23.

A general macro facility also permits the use of parameters in macro definitions [Exhibit 3.24]. In a call, macro arguments are easily parsed, since they are enclosed in parentheses and follow the macro name [Exhibit 3.25]. To expand a macro, formal parameter names must be identified in the definition of the macro. To do this, the tokens in the macro definition must first be identified. Any

---

<sup>8</sup>Newer C translators permit the “#” to be anywhere on the line as long as it is the first nonblank character.

---

**Exhibit 3.22. A nest of macros in C.**

The macros defined here are named `PI` and `PRINTX`. `PRINTX` expands into a call on the library function that does formatted output, `printf`. The first parameter for `printf` must be a format string, the other parameters are expressions denoting items to be printed. Within the format, a `%` field defines the type and field width for each item on the I/O list. The “`\t`” prints out a tab character.

```
#define PI      3.1415927
#define PRINTX printf("Pi times x = %8.5f\t", PI * x)
```

---



---

**Exhibit 3.23. Use of the simple PRINTX macro.**

The macro named `PRINTX` is used below in a `for` loop.

```
for (x=1; x<=3; x++) PRINTX;
```

Before compilation begins, the macro name is replaced by the string “`printf("Pi times x = %8.5f\t", PI * x)`”, giving a string that still contains a macro call:

```
for (x=1; x<=3; x++) printf("Pi times x = %8.5f\t", PI * x);
```

This string is re-scanned, and the call on the macro `PI` is expanded, producing macro-free source code. The compiler then compiles the statement:

```
for (x=1; x<=3; x++) printf("Pi times x = %8.5f\t", 3.1415927 * x);
```

At run time, this code causes `x` to be initialized to 1 before the loop is executed. On each iteration of the loop, the value of `x` is compared to 3. If `x` does not exceed 3, the words “`Pi times x =`” are printed, followed by the value of `3.1415927 * x` as a floating-point number with five decimal places (`%8.5f`), followed by a tab character (`\t`). The counter `x` is then incremented. The loop is terminated when `x` exceeds 3. Thus a line with five fields is printed, as follows:

```
Pi times x = 3.14159      Pi times x = 6.28319      Pi times x = 9.42477
```

---

---

**Exhibit 3.24. A macro with parameters in C.**

The macro defined here is named PRINT. It is similar to the PRINTX macro in Exhibit 3.22, but it has a parameter.

```
#define PRINT(yy) printf(#yy " = %d\t", yy)
```

The definition for PRINT is written in ANSI C. References to macro parameters that occur within quoted strings are not recognized by the preprocessor. However, the “#” symbol in a macro definition causes the parameter following it to be converted to a quoted string. Adjacent strings are concatenated by the translator. Using both these facts, we are able to insert a parameter value into a quoted format string.

---

token that matches a parameter name is replaced by the corresponding argument string. Finally, the entire string of characters, with parameter substitutions, replaces the macro call.

The original definition of C did not clearly define whether tokens were identified before or after macro parameters were processed. This is important because a comment or a quoted string looks like many words but forms a single program token. If a preprocessor searches for parameter names before identifying tokens, quoted strings will be searched and parameter substitution will happen within them. Many C translators work this way; others identify tokens first. The ANSI C standard clarifies this situation. It decrees that tokenization will be done uniformly before parameter substitution.

Macro names are syntactic extensions. They are words that may be written in the program and will be recognized by the compiler. Unlike variable declarations they may stand for arbitrarily complex items, and they may expand into strings that are not even syntactically legal units when used alone. Macros can be used to shorten code with repetitive elements, to redefine the compiler words such as BEGIN, or to give symbolic names to constants. What they *do not* do is extend the

---

**Exhibit 3.25. Use of the print macro with parameters.**

The macro named PRINT is used here, with different variables supplied as parameters each time.

```
PRINT(x); PRINT(y); PRINT(z);
```

These macro calls will be expanded and produce the following compilable code:

```
printf("x = %d\t", x);  
printf("y = %d\t", y);  
printf("z = %d\t", z);
```

Assume that at run time the variables x, y, and z contain the values 1, 3, and 10, respectively. Then executing this code will cause one line to be printed, as follows:

```
x = 1      y = 3      z = 10
```

---

semantics of the language. Since all macro calls must be expanded into compilable code, anything written with a macro call could also be written without it. No “power” is added to the language by a macro facility.

## Exercises

1. Why are function calls considered verbs?
2. What is the domain of a verb? Define the domain and range of a function.
3. What is a data type? Inheritance?
4. What is a metalanguage?
5. What is a lexical token? How are lexical tokens formed? Use a language with which you are familiar as an example. What are delimiters?
6. How are programming language statements analogous to sentences?
7. What is the scope of a programming language unit? How is it usually denoted?
8. How is it possible to improperly nest scopes? How can this be avoided by designers of programming languages?
9. What is the purpose of a comment? How are comments traditionally handled within programs? What is the advantage of using a carriage return as a comment delimiter?
10. The language C++ is an extension of C which supports generic functions and type checking. For the most part, C++ is C with additions to implement things that the C++ designers believed are important and missing from C. One of the additions is a second way to denote a comment. In C, a comment can be placed almost anywhere in the code and is delimited at both ends. In this program fragment two comments and an assignment statement are intermingled:

```
x=y*z    /* Add the product of y and z */+x;    /* to x. */
```

C++ supports this form but also a new form which must be placed on the right end of the line and is only delimited at the beginning by “//”:

```
x=y*z + x // Add the product of y and z to x.
```

Briefly explain why the original comment syntax was so inadequate that a new form was needed.

11. How can we extend a language through its vocabulary? Its syntax?
12. What is a macro? How is it used within a program?