

Part I

About Language

Chapter 1

The Nature of Language

Overview

This chapter introduces the concept of the nature of language. The purpose of language is communication. A set of symbols, understood by both sender and receiver, is combined according to a set of rules, its grammar or syntax. The semantics of the language defines how each grammatically correct sentence is to be interpreted. Using English as a model, language structures are studied and compared. The issue of standardization of programming languages is examined. Nonstandard compilers are examples of the use of deviations from an accepted standard.

This is a book about the structure of programming languages. (For simplicity, we shall use the term “language” to mean “programming language”.) We will try to look beneath the individual quirks of familiar languages and examine the essential properties of language itself. Several aspects of language will be considered, including vocabulary, syntax rules, meaning (semantics), implementation problems, and extensibility. We will consider several programming languages, examining the choices made by language designers that resulted in the strengths, weaknesses, and particular character of each language. When possible, we will draw parallels between programming languages and natural languages.

Different languages are like tools in a toolbox: although each language is capable of expressing most algorithms, some are obviously more appropriate for certain applications than others. (You can use a chisel to turn a screw, but it is not a good idea.) For example, it is commonly understood that COBOL is “good for” business applications. This is true because COBOL provides a large variety of symbols for controlling input and output formats, so that business reports may easily be

made to fit printed forms. LISP is “good for” artificial intelligence applications because it supports dynamically growing and shrinking data. We will consider how well each language models the objects, actions, and relationships inherent in various classes of applications.

Rather than accept languages as whole packages, we will be asking:

- What design decisions make each language different from the others?
- Are the differences a result of minor syntactic rules, or is there an important underlying semantic issue?
- Is a controversial design decision necessary to make the language appropriate for its intended use, or was the decision an accident of history?
- Could different design decisions result in a language with more strengths and fewer weaknesses?
- Are the good parts of different languages mutually exclusive, or could they be effectively combined?
- Can a language be extended to compensate for its weaknesses?

1.1 Communication

A natural language is a symbolic communication system that is commonly understood among a group of people. Each language has a set of symbols that stand for objects, properties, actions, abstractions, relations, and the like. A language must also have rules for combining these symbols. A speaker can communicate an idea to a listener if and only if they have a common understanding of enough symbols and rules. Communication is impaired when speaker and listener interpret a symbol differently. In this case, either speaker and/or listener must use feedback to modify his or her understanding of the symbols until commonality is actually achieved. This happens when we learn a new word or a new meaning for an old word, or correct an error in our idea of the meaning of a word.

English is for communication among people. Programs are written for both computers and people to understand. Using a programming language requires a mutual understanding between a person and a machine. This can be more difficult to achieve than understanding between people because machines are so much more literal than human beings.

The meaning of symbols in natural language is usually defined by custom and learned by experience and feedback. In contrast, programming languages are generally defined by an authority, either an individual language designer or a committee. For a computer to “understand” a human language, we must devise a method for translating both the syntax and semantics of the language into machine code. Language designers build languages that they know how to translate, or that they believe they can figure out how to translate.

On the other hand, if computers were the only audience for our programs we might be writing code in a language that was trivially easy to transform into machine code. But a programmer must be able to understand what he or she is writing, and a human cannot easily work at the level of detail that machine language represents. So we use computer languages that are a compromise between the needs of the speaker (programmer) and listener (computer). Declarations, types, symbolic names, and the like are all concessions to a human's need to understand what someone has written. The concession we make for computers is that we write programs in languages that can be translated with relative ease into machine language. These languages have limited vocabulary and limited syntax. Most belong to a class called *context-free languages*, which can be parsed easily using a stack. Happily, as our skill at translation has increased, the variety and power of symbols in our programming languages have also increased.

The language designer must define sets of rules and symbols that will be commonly understood among both human and electronic users of the language. The *meaning* of these symbols is generally conveyed to people by the combination of a formal semantic description, analogy with other languages, and examples. The meaning of symbols is conveyed to a computer by writing small modules of machine code that define the action to be taken for each symbol. The rules of syntax are conveyed to a computer by writing a compiler or interpreter.

To learn to use a new computer language effectively, a user must learn exactly what combinations of symbols will be accepted by a compiler and what actions will be invoked for each symbol in the language. This knowledge is the required common understanding. When the human communicates with a machine, he must modify *his own* understanding until it matches the understanding of the machine, which is embodied in the language translator. Occasionally the translator fails to “understand” a phrase correctly, as specified by the official language definition. This happens when there is an error in the translator. In this case the “understanding” of the translator must be corrected by the language implementor.

1.2 Syntax and Semantics

The *syntax of a language* is a set of rules stating how language elements may be grammatically combined. Syntax specifies how individual words may be written and the order in which words may be placed within a sentence.

The semantics of a language define how each grammatically correct sentence is to be interpreted. In a given language, the *meaning* of a sentence in a compiled language is the object code compiled for that sentence. In an interpreted language, it is the internal representation of the program, which is then evaluated. *Semantic rules* specify the meaning attached to each placement of a word in a sentence, the meaning of omitting a sentence element, and the meaning of each individual word. A speaker (or programmer) has an idea that he or she wishes to communicate. This idea is the speaker's *semantic intent*. The programmer must choose words that have the correct semantics so that the listener (computer) can correctly interpret the speaker's semantic intent.

All languages have syntax and semantics. Chapter 4 discusses formal mechanisms for expressing

the syntax of a language. The rest of this book is primarily concerned with semantics, the semantics of particular languages, and the semantic issues involved in programming.

1.3 Natural Languages and Programming Languages

We will often use comparisons with English to encourage you to examine language structures intuitively, without preconceived ideas about what programming languages can or cannot do. The objects and functions of a program correspond to the nouns and verbs of natural language. (We will use the word “functions” to apply to functions, procedures, operators, and some commands. Objects include variables, constants, records, and so on.)

There are a number of language traits that determine the character of a language. In this section we compare the ways in which these traits are embodied in a natural language (English) and in various programming languages. The differences between English and programming languages are real, but not as great as they might at first seem. The differences are less extreme now than they were ten years ago and will decrease as programming languages continue to evolve. Current programming language research is directed toward:

- Easing the constraints on the order in which statements must be given.
- Increasing the uses of symbols with multiple definitions.
- Permitting the programmer to talk about and use an object without knowing details of its representation.
- Facilitating the construction of libraries, thus increasing the number of words that can be understood “implicitly”.
- Increasing the ability of the language to express varied properties of the problem situation, especially relationships among classes of objects.

1.3.1 Structure

Programs must conform to very strict structural rules. These govern the order of statements and sections of code, and particular ways to begin, punctuate, and end every program. No deviation from these rules is permitted by the language definition, and this is enforced by a compiler.

The structure of English is more flexible and more varied, but rules about the structure of sentences and of larger units do exist. The overall structure of a textbook or a novel is tightly controlled. Indeed, each kind of written material has some structure it must follow. In any situation where the order of events is crucial, such as in a recipe, English sentences must be placed in the “correct” sequence, just like the lines in a program.

Deviation from the rules of structure is permitted in informal speech, and understanding can usually still be achieved. A human listener usually attempts to correct a speaker’s obvious errors.

For example, scrambled words can often be put in the right order. We can correct and understand the sentence: “I yesterday finished the assignment.” Spoonerisms (exchanging the first letters of nearby words, often humorously) can usually be understood. For example, “I kee my sids” was obviously intended to mean “I see my kids”. A human uses common sense, context, and poorly defined heuristics to identify and correct such errors.

Most programming language translators are notable for their intolerance of a programmer’s omissions and errors. A compiler will identify an error when the input text fails to correspond to the syntactic rules of the language (a “syntax error”) or when an object is used in the wrong context (a “type error”). Most translators make some guesses about what the programmer really meant, and try to continue with the translation, so that the programmer gets maximum feedback from each attempt to compile the program. However, compilers can rarely correct anything more than a trivial punctuation error. They commonly make faulty guesses which cause the generation of heaps of irrelevant and confusing error comments.

Some compilers actually do attempt to correct the programmer’s errors by adding, changing, respelling, or ignoring symbols so that the erroneous statement is made syntactically legal. If the attempted correction causes trouble later, the compiler may return to the line with the error and try a different correction. This effort has had some success. Errors such as misspellings and errors close to the end of the code can often be corrected and enable a successful translation. Techniques have been developed since the mid-1970s and are still being improved. Such error-correcting compilers are uncommon because of the relatively great cost for added time and extra memory needed. Some people feel that the added costs exceed the added utility.

1.3.2 Redundancy

The syntactic structure of English is highly redundant. The same information is often conveyed by several words or word endings in a sentence. If required redundancy is absent, as in the sentence “I finishes the assignment tomorrow”, we can identify that errors have occurred. The lack of agreement between “I” and “finishes” is a syntactic error, and the disagreement of the verb tense (present) with the meaning of “tomorrow” is a semantic error. [Exhibit 1.1]

A human uses the redundancy in the larger context to correct errors. For example, most people would be able to understand that a single letter was omitted in the sentence “The color of my coat is back”. Similarly, if a listener fails to comprehend a single word, she or he can usually use the redundancy in the surrounding sentences to understand the message. If a speaker omits a word, the listener can often supply it by using context.

Programming languages are also partly redundant, and the required redundancy serves as a way to identify errors. For example, the first C declaration in Exhibit 1.2 contains two indications of the intended data type of the variable named `price`: the type name, `int`, and the actual type, `float`, of the initial value. These two indicators conflict, and a compiler can identify this as an error. The second line contains an initializer whose length is longer than the declared size of the array named `table`. This lack of agreement in number is an identifiable error.

Exhibit 1.1. Redundancy in English.

The subject and verb of a sentence must “agree” in number. Either both must be singular or both plural:

Correct:	Mark likes the cake.		Singular subject, singular verb.
Wrong:	Mark like the cake.		Singular subject, plural verb.

The verb tense must agree with any time words in the sentence:

Correct:	I finished the work yesterday.		Past tense, past time.
Wrong:	I finish the work yesterday.		Present tense, past time.

Where categories are mentioned, words belonging to the correct categories must be used.

Correct:	The color of my coat is black.		Black is a color.
Wrong:	The color of my coat is back.		Back is not a color.

Sentences must supply consistent information throughout a paragraph. Pronouns refer to the preceding noun. A pronoun must not suddenly be used to refer to a different noun.

Correct:	The goalie is my son. He is the best. His name is Al.
Wrong:	The goalie is my son. He is the best. He is my father.

These errors in English have analogs in programming languages. The first error above is analogous to using a nonarray variable with a subscript. The second and third errors are similar to type errors in programming languages. The last error is analogous to faulty use of a pointer.

1.3.3 Using Partial Information: Ambiguity and Abstraction

English permits *ambiguity*, that is, words and phrases that have dual meanings. The listener must *disambiguate* the sentence, using context, and determine the actual meaning (or meanings) of the speaker.¹

To a very limited extent, programming languages also permit ambiguity. Operators such as + have two definitions in many languages, *integer+integer* and *real+real*. Object-oriented languages permit programmer-defined procedure names with more than one meaning. Many languages are *block-structured*. They permit the user to define contexts of limited scope, called *blocks*. The same symbol can be given different meanings in different blocks. Context is used, as it is in English, to disambiguate the meaning of the name.

¹A pun is a statement with two meanings, both intended by the speaker, where one meaning is usually funny.

Exhibit 1.2. Violations of redundancy rules in ANSI C.

```
int price = 20.98;           /* Declare and initialize variable. */
int table[3] = {11, 12, 13, 14}; /* Declare and initialize an array. */
```

The primary differences here are that “context” is defined very exactly in each programming language and quite loosely in English, and that most programming languages permit only limited ambiguity.

English supports *abstraction*, that is, the description of a quality apart from an instance. For example, the word “chair” can be defined as “a piece of furniture consisting of a seat, legs, and back, and often arms, designed to accommodate one person.”² This definition applies to many kinds of chairs and conveys some but not all of a particular chair’s properties. Older programming languages do not support this kind of abstraction. They require that all an object’s properties be specified when the name for that object is defined.

Some current languages support very limited forms of abstraction. For example, Ada permits names to be defined for *generic objects*, some of whose properties are left temporarily undefined. Later, the generic definition must be *instantiated* by supplying actual definitions for those properties. The instantiation process produces fully specified code with no remaining abstractions which can then be compiled in the normal way.

Smalltalk and C++ are current languages whose primary design goal was support for abstraction. A Smalltalk declaration for a class “chair” would be parallel to the English definition. Languages of the future will have more extensive ability to define and use partially specified objects.

1.3.4 Implicit Communication

English permits some things to be understood even if they are left unsaid. When we “read between the lines” in an English paragraph, we are interpreting both explicit and implicit messages. Understanding of the explicit message is derived from the words of the sentence. The implicit message is understood from the common experience of speaker and listener. People from different cultures have trouble with implicit communication because they have inadequate common understanding.

Some things may be left implicit in programming languages also. Variable types in FORTRAN and the type of the result of a function in the original Kernighan and Ritchie C may or may not be defined explicitly. In these cases, as in English, the full meaning of such constructs is defined by having a mutual understanding, between speaker and listener, about the meaning of things left unspecified. A programmer learning a new language must learn its implicit assumptions, more commonly called *defaults*.

Unfortunately, when a programmer relies on defaults to convey meaning, the compiler cannot tell the difference between the purposeful use of a default and an accidental omission of an important declaration. Many experienced programmers use explicit declarations rather than rely on defaults. Stating information explicitly is less error prone and enables a compiler to give more helpful error comments.

²Cf. Morris [1969].

1.3.5 Flexibility and Nuance

English is very *flexible*: there are often many ways to say something. Programming languages have this same flexibility, as is demonstrated by the tremendous variety in the solutions handed in for one student programming problem. As another example, APL provides at least three ways to express the same simple conditional branch.

Alternate ways of saying something in English usually have slightly different meanings, and subtlety and nuance are important. When different statement sequences in a programming language express the same algorithm, we can say that they have the same meaning. However, they might still differ in subtle ways, such as in the time and amount of memory required to execute the algorithm. We can call such differences *nuances*.

The nuances of meaning in a program are of both theoretical and practical importance. We are content when the work of a beginning programmer has the correct result (a way of measuring its meaning). As programmers become more experienced, however, they become aware of the subtle implications of alternative ways of saying the same thing. They will be able to produce a program with the same meaning as the beginner's program, but with superior clarity, efficiency, and compactness.

1.3.6 Ability to Change and Evolve

Expressing an idea in any language, natural or artificial, can sometimes be difficult and awkward. A person can become “speechless” when speaking English. Words can fail to express the strength or complexity of the speaker's feelings. Sometimes a large number of English words are required to explain a new concept. Later, when the concept becomes well understood, a word or a few words suffice.

English is constantly evolving. Old words become obsolete and new words and phrases are added. Programming languages, happily, also evolve. Consider FORTRAN for example. The original FORTRAN was a very limited language. For example, it did not support parameters and did not have an `IF . . . THEN . . . ELSE` statement. Programmers who needed these things surely found themselves “speechless”, and they had to express their logic in a wordy and awkward fashion. Useful constructs were added to FORTRAN because of popular demand. As this happened, some of the old FORTRAN words and methods became obsolete. While they have not been dropped from the language yet, that may happen someday.

As applications of computers change, languages are extended to include words and concepts appropriate for the new applications. An example is the introduction of words for sound generation and graphics into Commodore BASIC when the Commodore-64 was introduced with sound and graphics hardware.

One of the languages that evolves easily and constantly is FORTH. There are several public domain implementations, or dialects, used by many people and often modified to fit a user's hardware and application area. The modified dialect is then passed on to others. This process works like the process for adding new meanings to English. New words are introduced and become “common

knowledge” gradually as an increasing number of people learn and use them.

Translators for many dialects of BASIC, LISP, and FORTH are in common use. These languages are not fully *standardized*. Many dialects of the original language emerge because implementors are inspired to add or redesign language features. Programs written in one dialect must be modified to be used by people whose computer “understands” a different dialect. When this happens we say that a program is *nonportable*. The cost of rewriting programs makes nonstandardized programming languages unattractive to commercial users of computers. Lack of standardization can also cause severe difficulties for programmers and publishers: the language specifications and reference material must be relearned and rewritten for each new dialect.

1.4 The Standardization Process

Once a language is in widespread use, it becomes very important to have a complete and precise definition of the language so that compatible implementations may be produced for a variety of hardware and system environments. The standardization process was developed in response to this need. A language standard is a formal definition of the syntax and semantics of a language. It must be a complete, unambiguous statement of both. Language aspects that are defined must be defined clearly, while aspects that go beyond the limits of the standard must be designated clearly as “undefined”. A language translator that implements the standard must produce code that conforms to all defined aspects of the standard, but for an undefined aspect, it is permitted to produce any convenient translation.

The authority to define an unstandardized language or to change a language definition may belong to the individual language designer, to the agency that sponsored the language design, or to a committee of the American National Standards Institute (ANSI) or the International Standards Organization (ISO). The FORTRAN standard was originated by ANSI, the Pascal standard by ISO. The definition of Ada is controlled by the U.S. Department of Defense, which paid for the design of Ada. New or experimental languages are usually controlled by their designers.

When a standards organization decides to sponsor a new standard for a language, it convenes a committee of people from industry and academia who have a strong interest in and extensive experience with that language. The standardization process is not easy or smooth. The committee must decide which dialect, or combination of ideas from different dialects, will become the standard. Committee members come to this task with different notions of what is good or bad and different priorities. Agreement at the outset is rare. The process may drag on for years as one or two committee members fight for their pet features. This happened with the original ISO Pascal standard, the ANSI C standard, and the new FORTRAN-90 standard.

After a standard is adopted by one standards organization (ISO or ANSI), the definition is considered by the other. In the best of all worlds, the new standard would be accepted by the second organization. For example, ANSI adopted the ISO standard for Pascal nearly unchanged. However, smooth sailing is not always the rule. The new ANSI C standard is not acceptable to some ISO committee members, and when ISO decides on a C standard, it may be substantially

different from ANSI C.

The first standard for a language often clears up ambiguities, fixes some obvious defects, and defines a better and more portable language. The ANSI C and ANSI LISP standards do all of these things. Programmers writing new translators for this language must then conform to the common standard, as far as it goes. Implementations may also include words and structures, called *extensions*, that go beyond anything specified in the standard.

1.4.1 Language Growth and Divergence

After a number of years, language extensions accumulate and actual implementations diverge so much that programs again become nonportable. This has happened now with Pascal. The standard language is only minimally adequate for modern applications. For instance, it contains no support for string processing or graphics. Further, it has design faults, such as an inadequate `case` statement, and design shortcomings, such as a lack of static variables, initialized variables, and support for modular compilation. Virtually all implementations of Pascal for personal computers extend the language. These extensions are similar in intent and function but differ in detail. A program that uses the extensions is nonportable. One that doesn't use extensions is severely limited. We all need a new Pascal standard.

When a standardized language has several divergent extensions in common use, the sponsoring standards agency may convene a new committee to reexamine and restandardize the language. The committee will consider the collection of extensions from various implementations and decide upon a new standard, which usually includes all of the old standard as a subset.

Thus there is a constant tension between standardization and diversification. As our range of applications and our knowledge of language and translation techniques increase, there is pressure to extend our languages. Then the dialects in common use become diversified. When the diversity becomes too costly, the language will be restandardized.

1.5 Nonstandard Compilers

It is common for compilers to deviate from the language standard. There are three major kinds of deviations: extensions, intentional changes, and compiler bugs. The list of differences in Exhibit 1.3 was taken from the Introduction to the *Turbo Pascal Reference Manual, Version 2.0*. With each new version of Turbo, this list has grown in size and complexity. Turbo Pascal version 5 is a very different and much more extensive language than Standard Pascal.

An *extension* is a feature added to the standard, as string operations and graphics primitives are often added to Pascal. Items marked with a “+” in Exhibit 1.3 are true extensions: they provide processing capabilities for things that are not covered by the standard but do not change the basic nature of the language.

Sometimes compiler writers believe that a language, as it is officially defined, is defective; that is, some part of the design is too restrictive or too clumsy to use in a practical application environment. In these cases the implementor often redefines the language, making it nonstandard

Exhibit 1.3. Summary of Turbo Pascal deviations from the standard.

syntactic extensions	semantic extensions	semantic changes	
		!	Absolute address variables
		!	Bit/byte manipulation
		!	Direct access to CPU memory and data ports
	+		Dynamic strings
*			Free ordering of sections within declaration part
	+		Full support of operating system facilities
*			In-line machine code generation
*			Include files
		!	Logical operations on integers
*			Program chaining with common variables
	+		Random access data files
	+		Structured constants
	+		Type conversion functions (to be used explicitly)

and incompatible with other translators. This is an *intentional change*. Items marked with a “!” in Exhibit 1.3 change the semantics of the language by circumventing semantic protection mechanisms that are part of the standard. Items marked by a “*” are extensions and changes to the syntax of the language that do not change the semantics but, if used, do make Turbo programs incompatible with the standard.

A *compiler bug* occurs where, unknown to the compiler writer, the compiler implements different semantics than those prescribed by the language standard. Examples of compiler bugs abound. One Pascal compiler for the Commodore 64 required a semicolon after every statement. In contrast, the Pascal standard requires semicolons only as separators between statements and *forbids* a semicolon before an ELSE. A program written for this nonstandard compiler cannot be compiled by a standard compiler and vice versa.

An example of a common “bug” is implementation of the mod operator. The easy way to compute $i \bmod j$ is to take the remainder after using integer division to calculate i/j . According to the Pascal standard, quoted in Exhibit 1.4,³ this computation method is correct if both i and j are positive integers. If i is negative, though, the result must be adjusted by adding in the modulus, j . The standard considers the operation to be an error if j is negative. Note that mod is only the same as the mathematical remainder function if $i \geq 0$ and $j > 0$.

Many compilers ignore this complexity, as shown in Exhibits 1.5 and 1.6. They simply perform an integer division operation and return the result, regardless of the signs of i and j . For example, in OSS Pascal for the Atari ST, the mod operator is defined in the usual nonstandard way. The OSS

³Cooper [1983], page 3-1.

Exhibit 1.4. The definition of mod in Standard Pascal.

- The value of $i \bmod j$ is the value of $i - (k*j)$ for an integer value k , such that $0 \leq (i \bmod j) < j$. (That is, the value is always between 0 and j .)
 - The expression $i \bmod j$ is an error if j is zero or negative.
-

Pascal reference manual (page 6-26) describes `mod` as follows:

The modulus is the remainder left over after integer division.

Compiling and testing a few simple expressions [Exhibit 1.5] substantiates this and shows how OSS Pascal differs from the standard. Expression 2 gives a nonstandard answer. Expressions (3) through (6) compile and run, but shouldn't. They are designated as errors in the standard, which requires the modulus to be greater than 0. These errors are not detected by the OSS Pascal compiler or run-time system, nor does the OSS Pascal reference manual state that they will not be detected, as required by the standard.

In defense of this nonstandard implementation, one must note that this particular deviation is common and the function it computes is probably more useful than the standard definition for `mod`.

The implementation of `mod` in Turbo Pascal is different, but also nonstandard, and may have been an unintentional deviation. It was not included on the list of nonstandard language features. [Exhibit 1.3] The author of this manual seems to have been unaware of this nonstandard nature of `mod` and did not even describe it adequately. The partial information given in the Turbo reference manual (pages 51–52) is as follows:

```
mod is only defined for integers
its result is an integer
12 mod 5 = 2
```

Exhibit 1.5. The definition of mod in OSS Pascal for the Atari ST.

	Expression	OSS result	Answer according to Pascal Standard
1.	$5 \bmod 2$	1	Correct.
2.	$-5 \bmod 2$	-1	Should be 1 (between 0 and the modulus-1).
3.	$5 \bmod -2$	1	Should be detected as an error.
4.	$-5 \bmod -2$	-1	Should be detected as an error.
5.	$5 \bmod 0$	0	Should be detected as an error.
6.	$-5 \bmod 0$	-1	Should be detected as an error.

Exhibit 1.6. The definition of mod in Turbo Pascal for the IBM PC.

	Expression	Turbo result	Answer according to Pascal Standard
1.	$5 \bmod 2$	1	Correct.
2.	$-5 \bmod 2$	-1	Should be $-1 + 2 = 1$.
3.	$5 \bmod -2$	1	Should be an error.
4.	$-5 \bmod -2$	-1	Should be an error.
5.	$5 \bmod 0$	Run-time error	Correct.
6.	$-5 \bmod 0$	Run-time error	Correct.

The reference manual for Turbo Pascal version 4.0 still does not include `mod` on the list of non-standard features. However, it does give an adequate definition (p. 240) of the function it actually computes for `mod`:

“the `mod` operator returns the remainder from dividing the operands:

$$i \bmod j = i - (i/j) * j.$$

The sign of the result is the sign of i . An error occurs if $j = 0$.”

Compiling and testing a few simple expressions [Exhibit 1.6] substantiates this definition. Expression 2 gives a nonstandard answer. Expressions (3) and (4) are designated as errors in the standard, which requires the modulus to be greater than 0. These errors are not detected by the Turbo compiler. Furthermore, its reference manual does not state that they will not be detected, as required by the standard.

While Turbo Pascal will not compile a `div` or `mod` operation with 0 as a constant divisor, the result of “ $i \bmod 0$ ” can be tested by setting a variable, j , to zero, then printing the results of $i \bmod j$. This gives the results on lines (5) and (6).

Occasionally, deviations from the standard occur because an implementor believes that the standard, although unambiguous, defined an item “wrong”; that is, some other definition would have been more efficient or more useful. The version incorporated into the compiler is intended as an *improvement* over the standard. Again, the implementation of `mod` provides an example here. In many cases, the programmer who uses `mod` really wants the arithmetic remainder, and it seems foolish for the compiler to insert extra lines of code in order to compute the unwanted standard Pascal function. At least one Pascal compiler (for the Apollo workstation) provides a switch that can be set either to compile the standard meaning of `mod` or to compile the easy and efficient meaning. The person who wrote this compiler clearly believed that the standard was “wrong” to include the version it did rather than the integer remainder function.

The implementation of input and output operations in Turbo Pascal version 2.0 provides another example of a compiler writer who declined to implement the standard language because he believed his own version was clearly superior. He explains this decision as follows:⁴

⁴Borland [1984], Appendix F.

The standard procedures GET and PUT are not implemented. Instead, the READ and WRITE procedures have been extended to handle all I/O needs. The reason for this is threefold: Firstly, READ and WRITE gives much faster I/O, secondly variable space overhead is reduced, as file buffer variables are not required, and thirdly the READ and WRITE procedures are far more versatile and easier to understand than GET and PUT.

The actual Turbo implementation of READ did not even measure up to the standard in a minimal way, as it did not permit the programmer to read a line of input from the keyboard one character at a time. (It is surely inefficient to do so but essential in some applications.) Someone who did not know that this deviation was made on purpose would think that it was simply a compiler bug. This situation provides an excellent example of the dangers of “taking the law into your own hands”.

Whether or not we agree with the requirements of a language standard, we must think carefully before using nonstandard features. Every time we use a nonstandard feature or one that depends on the particular bit-level implementation of the language, it makes a program harder to port from one system to another and decreases its potential usefulness and potential lifetime. Programmers who use nonstandard “features” in their code should segregate the nonstandard segments and thoroughly document them.

Exercises

1. Define natural language. Define programming language. How are they different?
2. How are languages used to establish communication?
3. What is the syntax of a language? What are the semantics?
4. What are the traits that determine the character of a language?
5. How do these traits appear in programming languages?
6. What need led to standardization?
7. What is a “standard” for a language?
8. What does it mean when a language standard defines something to be “undefined”?
9. How does standardization lead to portability?
10. What three kinds of deviations are common in nonstandard compilers?
11. What are the advantages and disadvantages of using nonstandard language features?