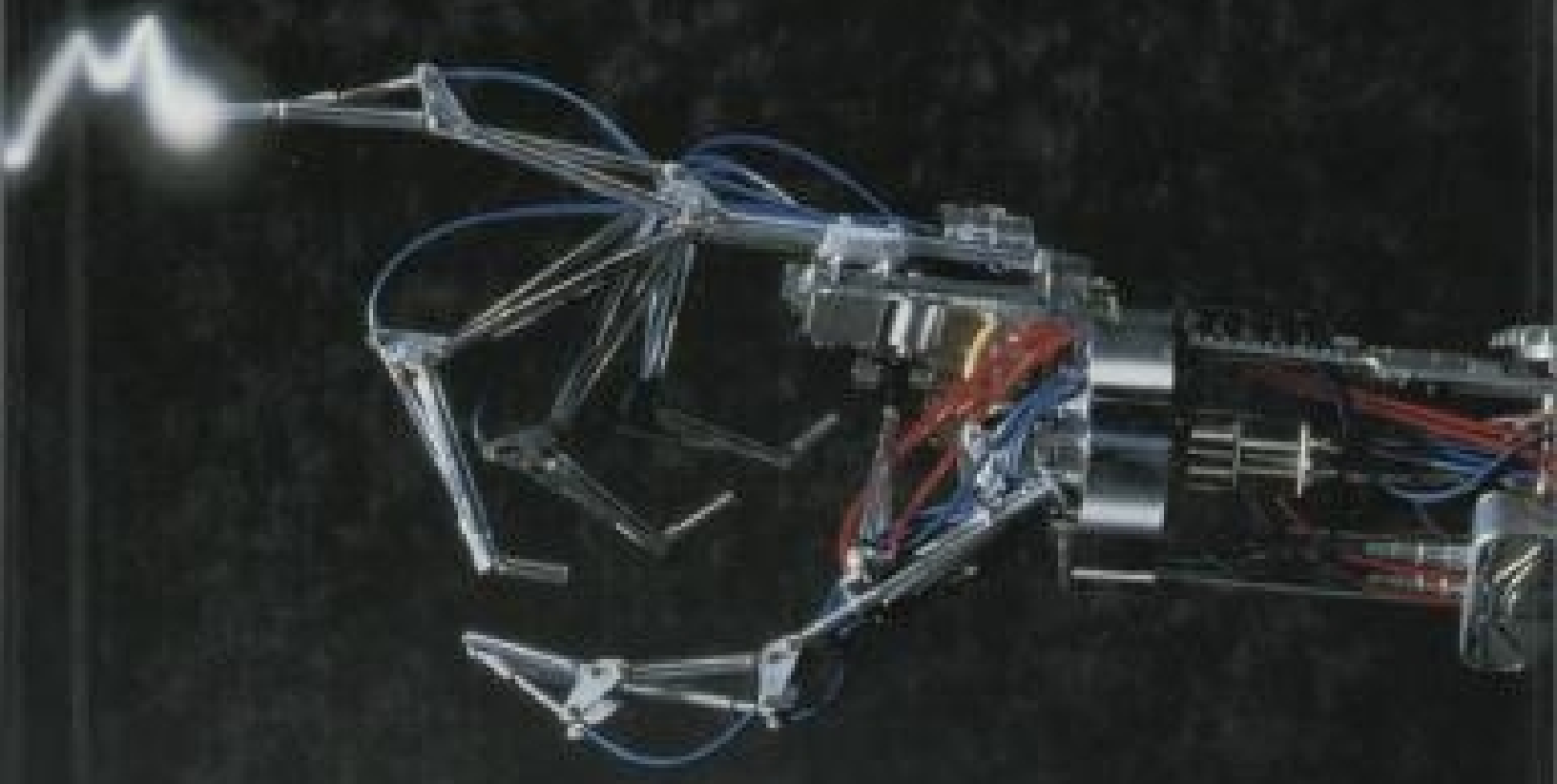
The Anatomy of Programming Languages



Alice E. Fischer • Frances S. Grodzinsky

Contents

Preface

This text is intended for a course in advanced programming languages or the structure of programming language and should be appropriate for students at the junior, senior, or master's level. It should help the student understand the principles that underlie all languages and all language implementations.

This is a comprehensive text which attempts to dissect language and explain how a language is really built. The first eleven chapters cover the core material: language specification, objects, expressions, control, and types. The more concrete aspects of each topic are presented first, followed by a discussion of implementation strategies and the related semantic issues. Later chapters cover current topics, including modules, object-oriented programming, functional languages, and concurrency constructs.

The emphasis throughout the text is on semantics and abstraction; the syntax and historical development of languages are discussed in light of the underlying semantical concepts. Fundamental principles of computation, communication, and good design are stated and are used to evaluate various language constructs and to demonstrate that language designs are improving as these principles become widely understood.

Examples are cited from many languages including Pascal, C, C++, FORTH, BASIC, LISP, FORTRAN, Ada, COBOL, APL, Prolog, Turing, Miranda, and Haskell. All examples are annotated so that a student who is unfamiliar with the language used can understand the meaning of the code and see how it illustrates the principle.

It is the belief of the authors that the student who has a good grasp of the structure of computer languages will have the tools to master new languages easily.

The specific goals of this book are to help students learn:

- To reason clearly about programming languages.
- To develop principles of communication so that we can evaluate the wisdom and utility of the decisions made in the process of language design.
- To break down language into its major components, and each component into small pieces so that we can focus on competing alternatives.
- To define a consistent and general set of terms for the components out of which programming languages are built, and the concepts on which they are based.

- To use these terms to describe existing languages, and in so doing clarify the conflicting terminology used by the language designers, and untangle the complexities inherent in so many languages.
- To see below the surface appearance of a language to its actual structure and descriptive power.
- To understand that many language features that commonly occur together are, in fact, independent and separable. To appreciate the advantages and disadvantages of each feature. To suggest ways in which these basic building blocks can be recombined in new languages with more desirable properties and fewer faults.
- To see the similarities and differences that exist among languages students already know, and to learn new ones.
- To use the understanding so gained to suggest future trends in language design.

Acknowledgement

The authors are indebted to several people for their help and support during the years we have worked on this project. First, we wish to thank our families for their uncomplaining patience and understanding.

We thank Michael J. Fischer for his help in developing the sections on lambda calculus, functional languages and logic. and for working out several sophisticated code examples. In addition, his assistance as software and hardware systems expert and TeX guru made this work possible.

Several reviewers read this work in detail and offered invaluable suggestions and corrections. We thank these people for their help. Special thanks go to Robert Fischer and Roland Lieger for reading beyond the call of duty and to Gary Walters for his advice and for the material he has contributed.

Finally, we thank our students at the University of New Haven and at Sacred Heart University for their feedback on the many versions of this book.

Parts of this manuscript were developed under a grant from Sacred Heart University.

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.

[1] A 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.

[2] Cf. 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 mod 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 >= 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

[3]Cooper [1983], page 3-1.

Exhibit 1.4. The definition of `mod` in Standard Pascal.

- The value of i `mod` j is the value of i−(k*j) for an integer value k, such that 0 <= (i `mod` j) < j. (That is, the value is always between 0 and j.)
- The expression i `mod` 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 `mod` 2	1	Correct.
2.	-5 `mod` 2	-1	Should be 1 (between 0 and the modulus-1).
3.	5 `mod` -2	1	Should be detected as an error.
4.	-5 `mod` -2	-1	Should be detected as an error.
5.	5 `mod` 0	0	Should be detected as an error.
6.	-5 `mod` 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 `mod` 2	1	Correct.
2.	-5 `mod` 2	-1	Should be -1 + 2 = 1.
3.	5 `mod` -2	1	Should be an error.
4.	-5 `mod` -2	-1	Should be an error.
5.	5 `mod` 0	Run-time error	Correct.
6.	-5 `mod` 0	Run-time error	Correct.

The reference manual for Turbo Pascal version 4.0 still does not include `mod` on the list of nonstandard 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 `mod` 0" can be tested by setting a variable, j, to zero, then printing the results of i `mod` 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]

[4]Borland [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?

Chapter 2

Representation and Abstraction

Overview

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

Representation may be explicit or implicit, coherent or diffused.

2.1 What Is a Program?

We can view a program two ways.

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

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

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

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

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

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

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

Exhibit 2.1. Modeling a charge account and relevant processes.

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

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

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

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

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

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

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

Exhibit 2.2. Representations of a date in English.

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

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

2.2 Representation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2.1 Semantic Intent

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

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

Exhibit 2.5. The structure of a table expressed implicitly.

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

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

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

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

2.2.2 Explicit versus Implicit Representation

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

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

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

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

2.2.3 Coherent versus Diffuse Representation

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

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

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

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

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

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

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

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

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

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

Exhibit 2.7. A stack represented coherently in Pascal.

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

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

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

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

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

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

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

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

Exhibit 2.9. Addition represented diffusely or coherently.

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

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

Exhibit 2.10. A coherently but not contiguously represented object.

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

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

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

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

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

2.3 Language Design

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

2.3.1 Competing Design Goals

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

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

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

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

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

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

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

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

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

Exhibit 2.11. A basic type not supported by Pascal.

Basic type implemented by most hardware: bit strings

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

Operations built into most hardware	Symbol in C
a right shift n places	`a >> n`
a left shift n places	`a << n`
a and b	`a & b`
a or b	`a | b`
a exclusive or b	`a ^ b`
complement a	`~ a`

2.3.2 The Power of Restrictions

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

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

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

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

Exhibit 2.12. Bitwise and logical operations.

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

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

Exhibit 2.13. Useful pointer operations supported in C.

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

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

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

code but are prohibited.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.3.3 Principles for Evaluating a Design

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

Principle of Frequency

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

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

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

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

Principle of Locality

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

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

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

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

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

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

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

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

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

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

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

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

Principle of Lexical Coherence

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

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

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

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

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

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

[3] McCarthy et al. [1962].

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

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

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

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

Exhibit 2.16. Syntax for lambda expressions in LISP.

The order of elements in the primitive syntax is:

```
((lambda
    ( ⟨list of dummy parameter names⟩ )
    ( ⟨body of the function⟩ ))
⟨list of actual parameter values⟩)
```

The order of elements in the extended syntax is:

```
(let
    ( ⟨list of dummy name - actual value pairs⟩ )
    ( ⟨body of the function⟩ ))
```

Exhibit 2.17. A LISP function call with poor coherence.

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

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

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

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

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

(′⟨name⟩ (⟨parameter list⟩) ⟨local variable name list⟩′, ′⟨entry label⟩′)

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

[4] Kessler[1988], p. 59.

Exhibit 2.18. A LISP function call with good coherence.

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

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

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

Exhibit 2.19. Poor lexical coherence in SNOBOL.

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

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

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

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

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

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

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

Principle of Distinct Representation

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

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

Exhibit 2.20. BASIC: `GOTO`s and statement ordering both use line numbers.

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

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

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

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

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

---

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

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

1. They define the correct order of lines in a program.
2. They are the targets of `GOTO`s and `IF`s.
3. They define the entry points of subroutines (the targets of `GOSUB`).

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

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

Principle of Too Much Flexibility

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

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

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

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

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

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

Exhibit 2.21. Strange but legal GOTOs in APL.

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

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

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

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

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

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

Exhibit 2.22. A computed GOTO idiom in APL.

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

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

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

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

The Principle of Semantic Power

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

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

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

The Principle of Portability

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

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

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

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

Exhibit 2.23. An archaic language feature in FORTRAN.

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

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

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

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

```
IF J-1 .LT. 0 THEN
    ⟨block 1⟩
ELSE
    ⟨block 2⟩
ENDIF
```

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

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

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

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

2.4 Classifying Languages

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

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

2.4.1 Language Families

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

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

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

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

Structured Languages. Control structures are provided that allow one to write programs without using `GOTO`. Procedures with call-by-value parameters[5] are supported. Note that we call Pascal

[5]See Chapter 9.

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

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

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

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

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

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

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

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

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

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

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

[6] This is the term used in Smalltalk.

[7] See Chapter 8.

Exhibit 2.24. The development of procedural languages.

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

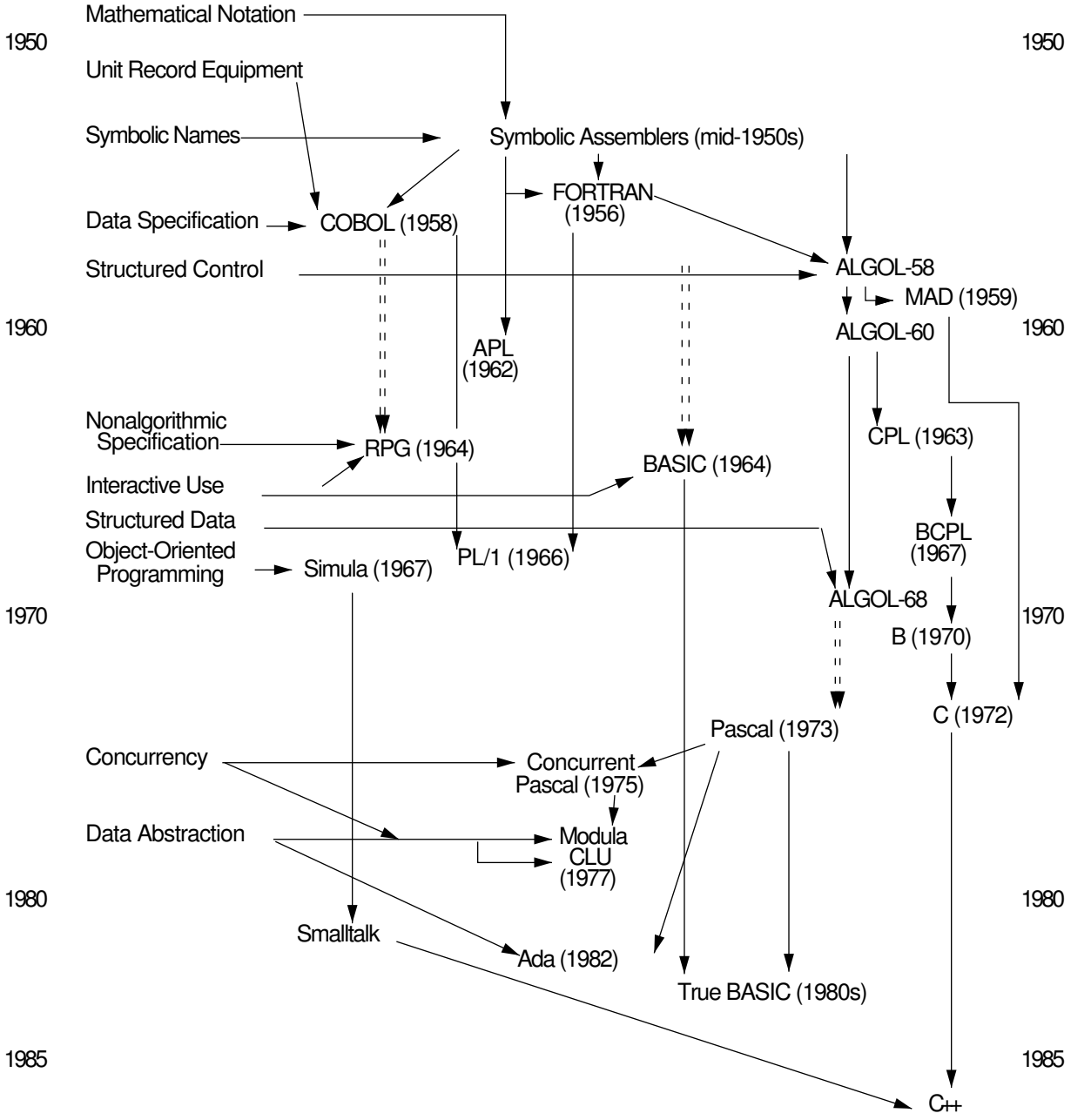
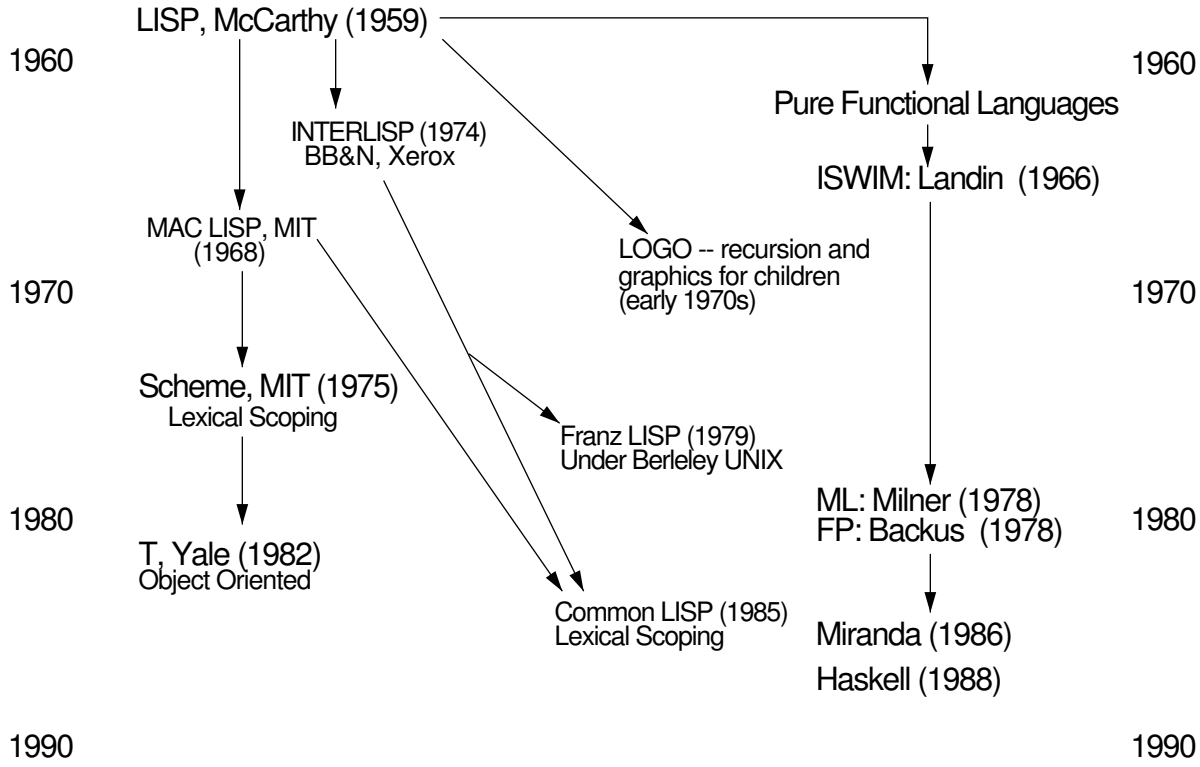


Exhibit 2.25. The development of functional languages.



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

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

Examples: LISP, Common LISP, T, Scheme.

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

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

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

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

Examples: ML, Miranda, Haskell.

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

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

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

[8] Parameter passing is explained fully in Chapter 9.

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

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

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

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

Examples: C, FORTH.

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

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

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

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

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

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

programming.

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

Examples: LISP, T, Scheme, Miranda.

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

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

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

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

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

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

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

Examples: Postscript, TeX, LaTeX.

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

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

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

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

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

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

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

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

2.4.2 Languages Are More Alike than Different

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

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

Exercises

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

15. What is a structured language? Strongly typed language? Object-oriented language? Parallel language? Fourth-generation language?

16. Why are most languages more similar than they are different? From what causes do language differences arise?

17. Discuss two aspects of a language design that make it hard to read, write, or use. Give an example of each, drawn from a language with which you are familiar.

18. Choose three languages from the following list: Smalltalk, BASIC, APL, LISP, C, Pascal, Ada. Describe one feature of each that causes some people to defend it as the "best" language for some application. Choose features that are unusual and do not occur in many languages.

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).

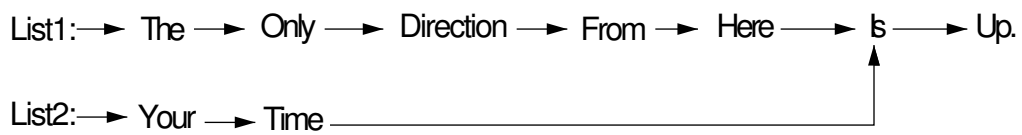
Conversely, in most languages, a single object can be bound to more than one name at a time, producing an *alias*. This occurs when a formal parameter name is bound to an actual parameter during a function call.

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.

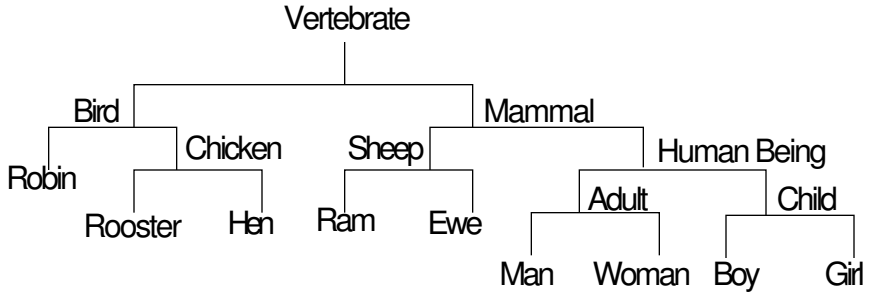
```
type REAL is digits 7;
price:  REAL;
```

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—hey 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.[1]

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.

[1] 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.[a]

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).

[a] 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.[2] 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.[3]
- A control card requesting compilation, an object program listing, an error listing, and a memory map.[4]
- 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[5] of the executable program.

[2]Caution: In a discussion of formal grammars and parsing, the term "sentence" is often used to mean the entire program, not just one statement.

[3]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.

[4]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.

[5]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.[6]

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

[6] 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.

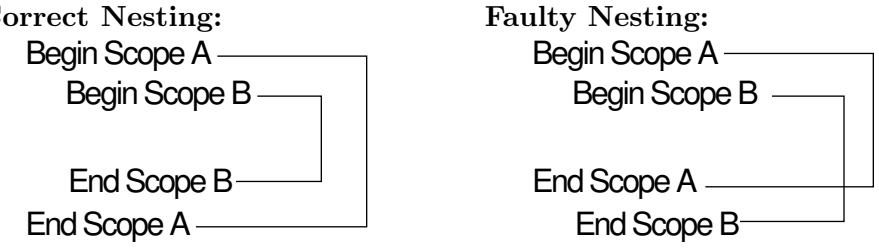


Exhibit 3.13. Tokens used to delimit program scopes.

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.

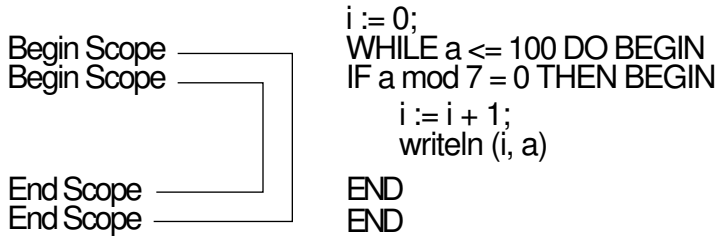
Using one uniform end-scope indicator has the severe disadvantage that a nesting error may not be identified as a syntactic error, but become a logical error which is harder to identify and correct. The programmer has one fewer tool for communicating semantics to the compiler, and the compiler has one 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**.**



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: ⍝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	/* ... */
PL/1	/* ... */
Pascal	(* ... *) or { ... }
FORTH	(...)

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).[7] 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

[7] 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.[8] 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

[8] 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?

Chapter 4

Formal Description of Language

Overview

The syntax of a language is its grammatical rules. These are usually defined through EBNF (Extended Backus-Naur Form) and/or syntax diagrams, both discussed in this chapter. The meaning of a program is represented by p-code (portable code) or by a computation tree. The language syntax defines the computation tree that corresponds to each legal source program.

Semantics are the rules for interpreting the meaning of programming language statements. The semantic specification of a language defines how each computation tree is to be implemented on a machine so that it retains its meaning. Being always concerned with the portability of code, we define the semantics of a language in terms of an implementation-independent model. One such model, the abstract machine, is composed of a program environment, shared environment, stack, and streams. The semantic basis of a language means the specific version of the machine that defines the language, together with the internal data structures and interpretation procedures that implement the abstract semantics. Lambda calculus is an example of a minimal semantic basis.

A language may be extended primarily through its vocabulary and occasionally through its syntax, as in EL/1, or through its semantics, as in FORTH.

4.1 Foundations of Programming Languages

Formal methods have played a critical role in the development of modern programming languages. Formal methods were not available in the mid-1950s when the first higher-level programming languages were being created. The most notable of these efforts was FORTRAN, which survives (in greatly expanded form) to this day. Even though the syntax and semantics of the early FORTRAN were primitive by today's standards, the complexity of the language was at the limit of what could be handled by the methods then available. It was quickly realized that ad hoc methods are severely limited in what they can achieve, and a more systematic approach would be needed to handle languages of greater expressive power and correspondingly greater complexity.

Contemporaneously with the implementation of the FORTRAN language and compiler, a new language, ALGOL, was being defined using a new formal approach for the specification of syntax and semantics. Even though it required several more years of research before people learned how to compile ALGOL efficiently, the language itself had tremendous influence on the design of subsequent programming languages. Concepts such as block structure (cf. Chapter 7) and delayed evaluation of function parameters (cf. Chapter 8), introduced in ALGOL, have reappeared in many subsequent modern programming languages.

ALGOL was the first programming language whose syntax was formally described. A notation called BNF, for Backus-Naur Form, was invented for the purpose. BNF turned out to be equivalent in expressive power to context-free grammars, developed by the linguist Noam Chomsky for describing natural language, but the BNF notation turned out to be easier for people, so variations on it are still used in describing most programming languages. An attempt was made to give a rigorous English-language specification of the semantics of ALGOL. Nevertheless, the underlying model was not well understood at the time, and ALGOL appeared at first to be difficult or impossible to implement efficiently.

Syntax and semantic interpretations were specified informally for early languages. Then, motivated by the new need to describe programming languages, formal language theory flourished. Some of the major developments in the foundations of computer science are shown in Exhibit 4.1. Formal syntax and parsing methods grew from work on automata theory and linguistics [Exhibit 4.1]. Formal methods of semantic specification [Exhibit 4.2] grew from early work on logic and computability and were especially influenced by Church's work on the lambda calculus. In this chapter, we give a brief introduction to some of the formal tools that have been important to the development of modern-day programming languages.

4.2 Syntax

The rules for constructing a well-formed sentence (statement) out of words, a paragraph (module) out of sentences, and an essay (program) out of paragraphs are the syntax of the language. The syntax definitions for most programming languages take several pages of text. A few are very short, a few very long. There is at least one language (ALGOL-68) in which the syntax rules that

Exhibit 4.1. Foundations of computer science.

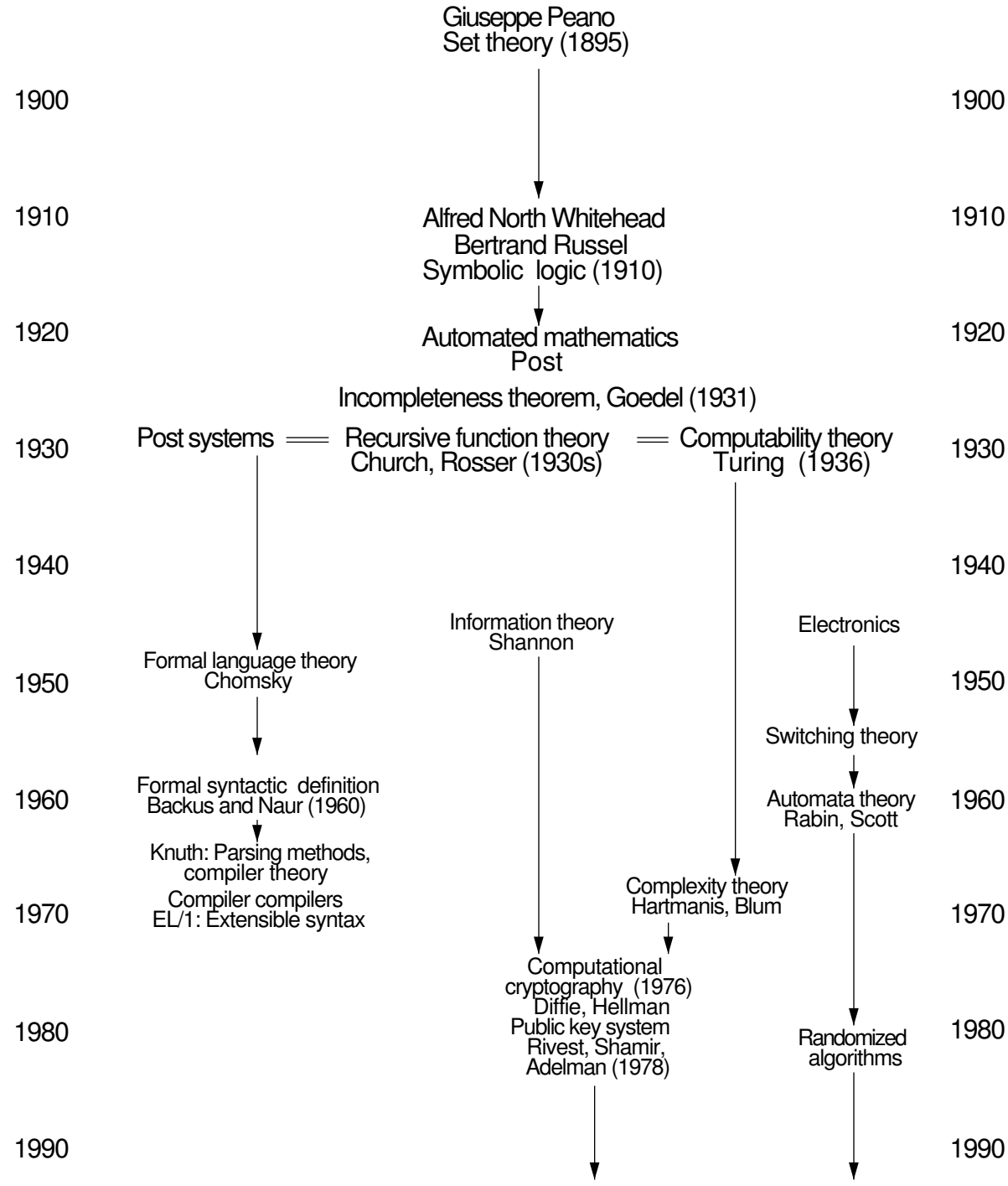
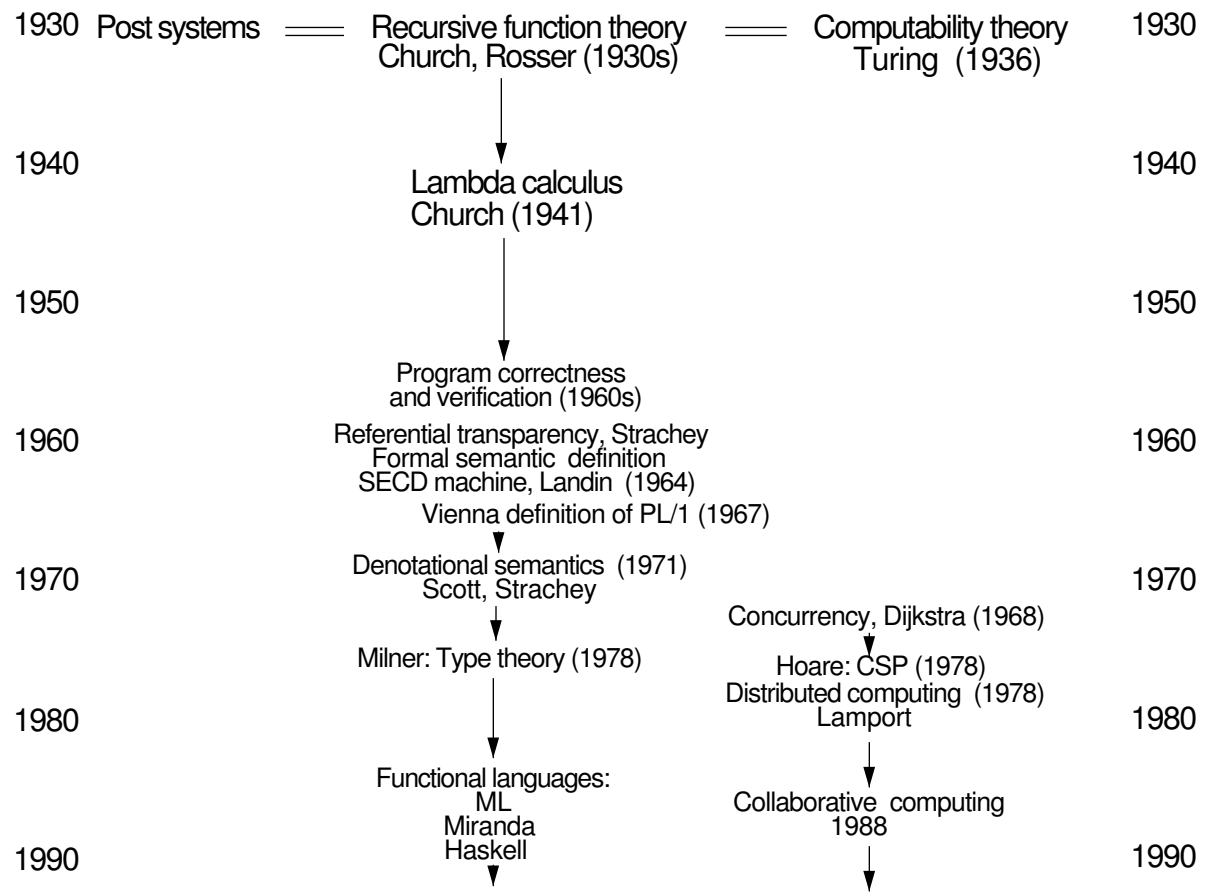


Exhibit 4.2. Formal semantic specification.



determine whether or not a statement should compile are so complicated that only an expert can understand them.

It is usual to define the syntax of a programming language in a *formal language*. A variety of formalisms have been introduced over the years for this purpose. We present two of the most common here: Extended Backus-Naur Form (EBNF) and syntax diagrams.

An EBNF language definition can be translated by a program called a *parser generator*[1] into a program called a *parser* [Exhibit 4.3].[2] A parser reads the user's source code programs and determines the *syntactic category* (part of speech) of every source symbol and combination of

[1]The old term was "compiler compiler". This led to the name of the UNIX parser generator, yacc, which stands for "yet another compiler compiler".

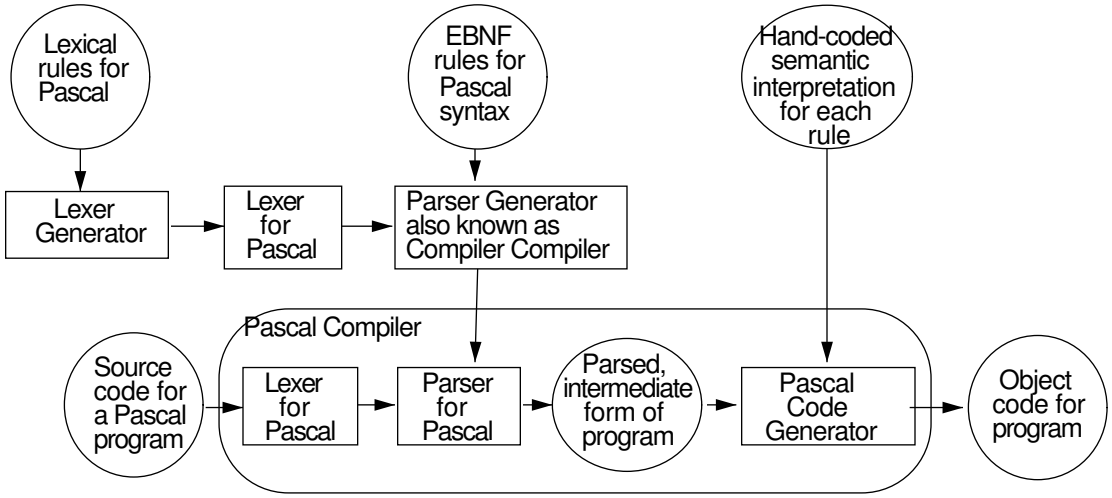
[2]A parser generator can only handle grammars for "context-free" languages. Defining this language class is beyond the scope of this book. Note, though, that the grammars published for most programming languages are context free.

Exhibit 4.3. The compiler is produced from the language definition.

In the following diagram, programs are represented by rectangles and data by circles. The lexer and parser can be automatically generated from the lexical specifications and syntax of a context-free language by a parser generator and its companion lexer generator. This is represented by the vertical arrows in the diagram. The lexer and parser are the output data of these generation steps.

A code generator requires more hand work: the compiler writer must construct an assembly code translation, for every syntax rule in the grammar, which encodes the semantics of that rule in the target machine language.

The lexer, parser, and code generator are programs that together comprise the compiler. The compilation process is represented by the horizontal chain in the diagram.



symbols. Its output is the list of the symbols defined in the program and a *parse tree*, which specifies the role that each source symbol is serving, much like a sentence diagram of an English sentence. The parser forms the heart of any compiler or interpreter for the language.

The study of formal language theory and parsing has strongly affected language design. Older languages were not devised with modern parsing methods in mind. Their syntax was usually developed ad hoc. Consequently, a syntax definition for such a language, for example FORTRAN, is lengthy and full of special cases. By today's standards these languages are also relatively slow and difficult to parse.

Newer languages are designed to be parsed easily by efficient algorithms. The syntax for Pascal is brief and elegant. Pascal compilers are small, as compilers go, and can be implemented on personal computers. The standard LISP translator[3] is only fifteen pages long!

[3]Griss and Hearn [1981].

4.2.1 Extended BNF

"Backus-Naur Form", or *BNF*, is a formal language developed by Backus and Naur for describing programming language syntax. It gained widespread influence when it was used to define ALGOL in the early 1960s. The original BNF formalism has since been extended and streamlined; a generally accepted version, named "Extended BNF", is presented here.

An EBNF grammar consists of:

- A starting symbol.
- A set of terminal symbols, which are the keywords and syntactic markers of the language being defined.
- A set of nonterminal symbols, which correspond to the syntactic categories and kinds of statements of the language.
- A series of rules, called productions, that specify how each nonterminal symbol may be expanded into a phrase containing terminals and nonterminals. Every nonterminal has one production rule, which may contain alternatives.

The Syntax of EBNF

The syntax for EBNF itself is not altogether standardized; several minor variations exist. We define a commonly used version here.

The starting symbol must be defined. One nonterminal is designated as the starting symbol.

Terminal symbols will be written in **boldface** and enclosed in 'single quotes'.

Nonterminal symbols will be written in regular type and enclosed in ⟨angle brackets⟩.

Production rules. The nonterminal being defined is written at the left, followed by a "::=" sign (which we will pronounce as "goes to"). After this is the string, with options, which defines the nonterminal. The definition extends up to but does not include the "." that marks the end of the production. When a nonterminal is *expanded* it is replaced by this defining phrase. Blank spaces between the "::=" and the "." are ignored.

Alternatives are separated by vertical bars. Parentheses may be used to indicate grouping. For example, the rule

s ::= (a | bc) d .

indicates that an 's' may be replaced by an 'ad' or a 'bcd'.

An optional syntactic element is a something-or-nothing alternative—it may be included or not included as needs demand. This is indicated by enclosing the optional element in square brackets, as follows:

s ::= [a] d .

This formula indicates that an 's' may be replaced by an 'ad' or simply by a 'd'.

An unspecified number of repetitions (zero or more) of a syntactic unit is indicated by enclosing the unit in curly brackets. For example, the rule

s ::= {a}d .

indicates that an 's' may be replaced by a 'd', an 'ad', an 'aad', or a string of any number of 'a's followed by a single 'd'. A frequently occurring pattern is the following:

s ::= t{t}

This means that 's' may be replaced by *one or more* copies of 't'.

Recursive rules. Recursive production rules are permitted. For example, this rule is directly recursive because its right side contains a reference to itself:

s ::= asz | w .

This expands into a single 'w', surrounded on the left and right by any number of matched pairs of 'a' and 'z': awz, aawzz, aaawzzz, etc.

Tail recursion is a special kind of recursion in which the recursive reference is the last symbol in the string. Tail recursion has the same effect as a loop. This production is tail recursive:

s ::= as | b .

This expands into a string of any number of 'a's followed by a 'b'.

Mutually recursive rules are also permitted. For example, this pair of rules is mutually recursive because each rule refers to the other:

s ::= at | b .
t ::= bs | a .

A single 's' could expand into any of the following: b, aa, abb, abaa, ababb, ababaa, etc.

Combinations of alternatives, optional elements, recursions, and repetitions often occur in a production, as follows:

s ::= {a | b} [c] d .

This rule indicates that an 's' may be replaced by any of the following: d, ad, bd, cd, acd, bcd, aad, abd, aacd, abcd, bd, bad, bbd, bcd, bacd, bbcd, and many more.

Using EBNF

To illustrate the EBNF rules, we give part of the syntax for Pascal, taken from the ISO standard [Exhibit 4.4]. The first few rules of the grammar are given, followed by several rules from the middle of the grammar which define what a "statement" is. The complete set of EBNF grammar rules cannot be given here because it is too long.[4] Following are brief explanations of the meaning

[4]It occupies nearly six pages in Cooper [1983].

Exhibit 4.4. EBNF production rules for parts of Pascal.

```
program ::= ⟨program-heading⟩ ';' ⟨program-block⟩ '.' .
program-heading ::= 'program' ⟨identifier⟩ [ '(' ⟨program-parameters⟩ ')' ].
program-parameters ::= ⟨identifier-list⟩ .
identifier-list ::= ⟨identifier⟩ { ',' ⟨identifier⟩ } .
program-block ::= ⟨block⟩ .
block ::= ⟨label-declaration-part⟩ ⟨constant-declaration-part⟩
       ⟨type-declaration-part⟩ ⟨variable-declaration-part⟩
       ⟨procedure-and-function-declaration-part⟩ ⟨statement-part⟩ .
variable-declaration-part ::= [ 'var' { ⟨identifier-list⟩ ':' ⟨typename⟩ ';' } ] .
statement-part ::= compound statement .
compound-statement ::= 'begin' ⟨statement-sequence⟩ 'end' .
statement-sequence ::= ⟨statement⟩ { ';' ⟨statement⟩ } .
statement ::= [ ⟨label⟩ ':' ] ( ⟨simple-statement⟩ | ⟨structured-statement⟩ ).
simple-statement ::= ⟨empty-statement⟩ | ⟨assignment-statement⟩ |
                     ⟨procedure-statement⟩ | ⟨goto-statement⟩ .
structured-statement ::= ⟨compound-statement⟩ | ⟨conditional-statement⟩ |
                     ⟨repetitive-statement⟩ | ⟨with-statement⟩ .
```

of these rules.

- The production for the starting symbol states that a `program` consists of a heading, a semicolon, a block and a period. The semicolon and period are terminal symbols and will form part of the finished program. The symbols "program-heading" and "program-block" are nonterminals and need further expansion.
- The program-heading starts with the terminal symbol "program", which is followed by the name of the program and an optional, parenthesized list of parameters, used for file names.
- The program parameters, if they are used, are just a list of identifiers, that is, a series of one or more identifiers separated by commas.
- The program block consists of a series of declarations followed by a single compound statement.
- The production for "compound statement" forms an indirectly recursive cycle with the rules for statement sequence, and statement. That is, a statement can be a structured statement,

which can be a compound statement, which contains a statement-sequence, which contains a statement, completing the cycle.

- The rule for "statement" contains an optional label field and the choice between "simple-statement" and "structured-statement".
- The rules for simple-statement and structured-statement define all of Pascal's control structures.

Generating a Program. To *generate* a program (or part of a program) using a grammar, one starts with the specified starting symbol and expands it according to its production rule. The starting symbol is replaced by the string of symbols from the right side of its production rule. If the rule contains alternatives, one may use whichever option seems appropriate. The resulting expansion will contain other nonterminal symbols which then must be expanded also. When all the nonterminals have been expanded, the result is a grammatically correct program.

We illustrate this derivation process by using the EBNF grammar for ISO Standard Pascal to generate a ridiculously simple program named "little". Parts, but not all, of this grammar are given in Exhibit 4.4.[5]

The starting symbol is ⟨program⟩. Wherever possible, more than one nonterminal symbol is reduced on each line, in order to shorten the derivation.

⟨program⟩
⟨program-heading⟩ ; ⟨program-block⟩ .
program ⟨identifier⟩ ; ⟨block⟩ .
program little ; ⟨label-declaration-part⟩ ⟨constant-declaration-part⟩
⟨variable-declaration-part⟩ ⟨procedure-and-function-declaration-part⟩
⟨statement-part⟩ .

program little ; var ⟨variable-declaration⟩ ; ⟨compound-statement⟩ .

program little ; var ⟨identifier-list⟩ : ⟨type-denoter⟩ ;
begin ⟨statement-sequence⟩ **end .**

program little ; var ⟨identifier⟩ : ⟨type-denoter⟩ ;
begin ⟨statement⟩ ; ⟨statement⟩ **end .**

program little ; var x : integer ;
begin ⟨simple-statement⟩ ; ⟨simple-statement⟩ **end .**

program little ; var x : integer ;
begin ⟨assignment-statement⟩ ; ⟨procedure-statement⟩ **end .**

program little ; var x : integer ;
begin ⟨variable-access⟩ := ⟨expression⟩ ;
⟨procedure-identifier⟩ **(** ⟨writeln-parameter-list⟩ **) end .**

[5]The complete grammar can be found in Cooper [1983], pp 153–58.

program little ; var x : integer ;
begin ⟨entire-variable⟩ **:=** ⟨simple-expression⟩ **;**
writeln (⟨write-parameter⟩ **) end .**

program little ; var x : integer ; begin ⟨variable-identifier⟩**:=** ⟨term⟩ **;**
writeln (⟨expression⟩ **) end .**

program little ; var x : integer ; begin ⟨identifier⟩ **:=** ⟨factor⟩ **;**
writeln (⟨simple-expression⟩ **) end.**

program little ; var x : integer ; begin x := ⟨unsigned-constant⟩ **;**
writeln (⟨term⟩ **) end .**

program little ; var x : integer ; begin x := ⟨unsigned-number⟩ **;**
writeln (⟨factor⟩ **) end .**

program little ; var x : integer ; begin x := ⟨unsigned-integer⟩ **;**
writeln (⟨variable-access⟩ **) end .**

program little ; var x : integer ; begin x := 17 ;
writeln (⟨entire-variable⟩ **) end .**

program little ; var x : integer ; begin x := 17 ;
writeln (⟨variable-identifier⟩ **) end .**

program little ; var x : integer ; begin x := 17 ; writeln (⟨identifier⟩ **) end .**

program little ; var x : integer ; begin x := 17 ; writeln (x) end .

Parsing a Program. The process of *syntactic analysis* is the inverse of this generation process. Syntactic analysis starts with source code. The parsing routines of a compiler determine how the source code corresponds to the grammar. The output from the parse is a tree-representation of the grammatical structure of the code called a *parse tree*.

There are several methods of syntactic analysis, which are usually studied in a compiler course and are beyond the scope of this book. The two broad categories of parsing algorithms are called "bottom-up" and "top-down". In top-down parsing, the parser starts with the grammar's starting symbol and tries, at each step, to generate the next part of the source code string. A brief description of a "bottom-up" method should serve to illustrate the parsing process. In a "bottom-up" parse, the parser searches the source code for a string which occurs as one alternative on the right side of some production rule. Ambiguity is resolved by looking ahead k input symbols. The matching string is replaced by the nonterminal on the left of that rule. By repeating this process, the program is eventually reduced, phrase by phrase, back to the starting symbol. Exhibit 4.5 illustrates the steps in forming a parse tree for the body of the program named "little".

All syntactically correct programs can be reduced in this manner. If a compiler cannot do the reduction successfully, there is some error in the source code and the compiler produces an error

comment containing some guess about what kind of syntactic error was made. These guesses are usually close to being correct when the error is discovered near where it was made. Their usefulness decreases rapidly as the compiler works on and on through the source code without discovering the error, as often happens.

4.2.2 Syntax Diagrams

Syntax diagrams were developed by Niklaus Wirth to define the syntax of Pascal. They are also called "railroad diagrams", because of their curving, branching shapes. This is the form in which Pascal syntax is usually presented in textbooks. Syntax diagrams and EBNF can express exactly the same class of languages, but they are used for different purposes. Syntax diagrams provide a graphic, two-dimensional way to communicate a grammar, so they are used to make grammatical relationships easier for human beings to grasp.

EBNF is used to write a grammar that will be the input to a parser generator. Corresponding to each production is code for the semantic action that the compiler should take when that production is parsed. The rules of an EBNF syntax are often more broken up than seems necessary, in order to provide "hooks" for all the semantic actions that a compiler must perform. When a grammar for the same language is presented as syntax diagrams, several EBNF productions are often condensed into one diagram, making the entire grammar shorter, less roundabout, and easier to comprehend.

A Wirth syntax diagram definition has the same elements as an EBNF grammar, as follows:

- A starting symbol.
- Terminal symbols, written in **boldface** but without quotes, sometimes also enclosed in round or oval boxes.
- Nonterminal symbols, written in regular type.
- Production rules are written using arrows (as in a flow chart) to indicate alternatives, options, and indefinite repetition. Each rule starts with a nonterminal symbol written at the left and ends where the arrow ends on the right.

Nonterminal symbols are like subroutine calls. To expand one, you go to the correct diagram, follow the arrows through the diagram until it ends, and return to the calling point to finish the calling production. Branch points correspond to alternatives and indicate that any appropriate choice can be made. Repetition is encoded by backward-pointing arrows which form explicit loops. Direct and indirect recursion are both allowed.

Syntax diagrams are given in Exhibits 4.6 and 4.7, which correspond exactly to the EBNF grammar fragments in Exhibit 4.4.

In spite of the simplicity and visual appeal of syntax diagrams, though, the official definition of Pascal grammar is written in EBNF, not syntax diagrams. EBNF is a better input language for a parser generator and provides a clearer basis for a formal definition of the semantics of the language.

Revision 1.8 1992/06/09 17:15:02 fischer

Exhibit 4.5. Parsing a simple Pascal program.

We perform a bottom-up parse of part of the program named "little", using standard Pascal syntax, part of which is shown in Exhibit 4.4. Starting with the expression at the top, we identify a single token or a consecutive series of tokens that correspond to the right side of a syntactic rule. This series is then "reduced", or replaced by the left side of that rule. The final reduction is shown at the bottom of the diagram.

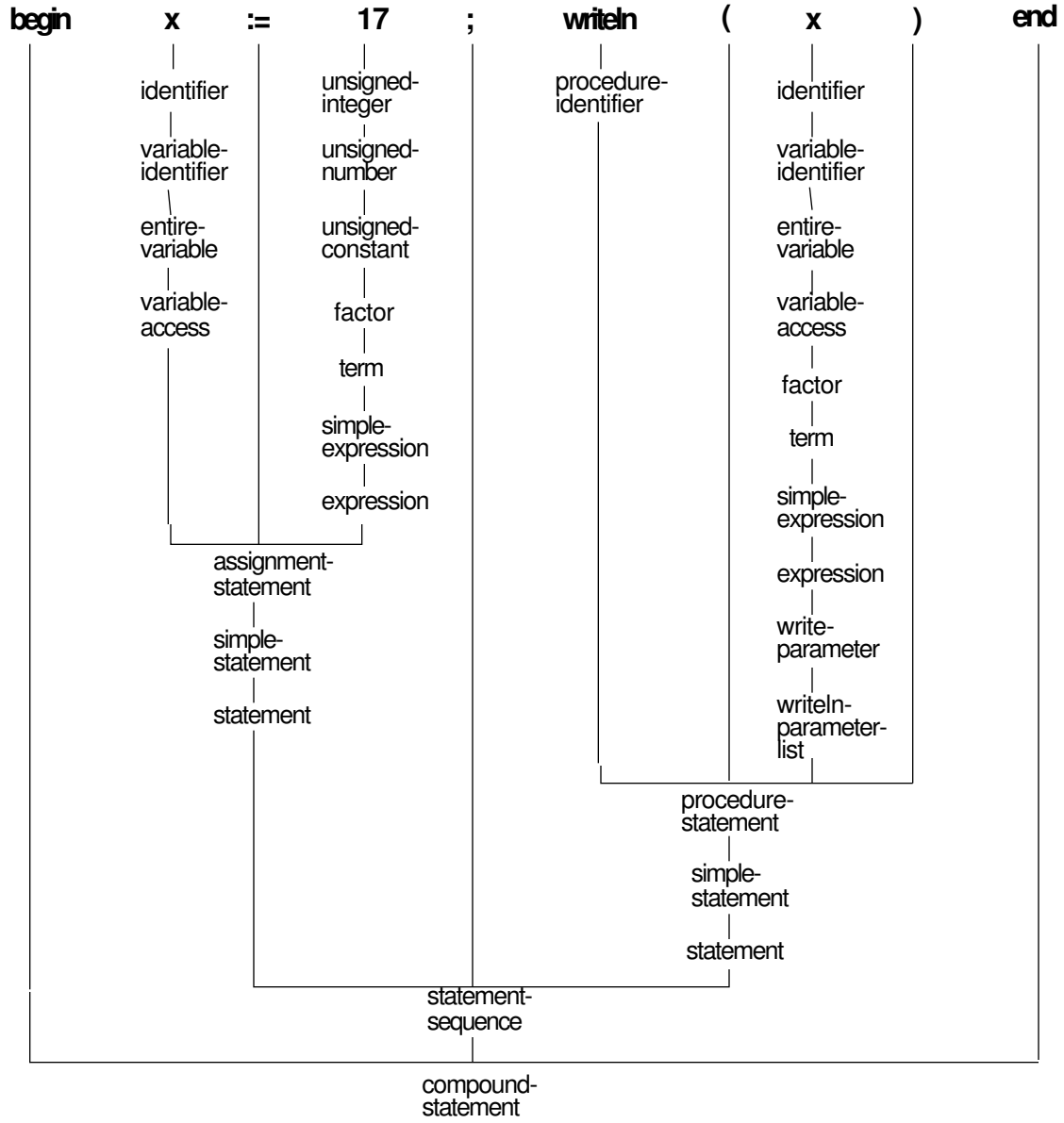


Exhibit 4.6. Syntax diagram for "program".

This diagram corresponds to the EBNF productions for program, program-heading, program-parameters, and identifier list. The starting symbol is "program" .

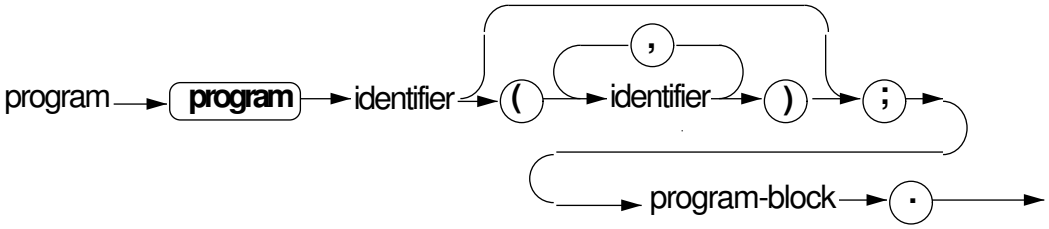


Exhibit 4.7. Syntax diagrams for "statement".

These diagrams correspond to the EBNF productions for statement, simple-statement, structured-statement, compound-statement, and statement-sequence.

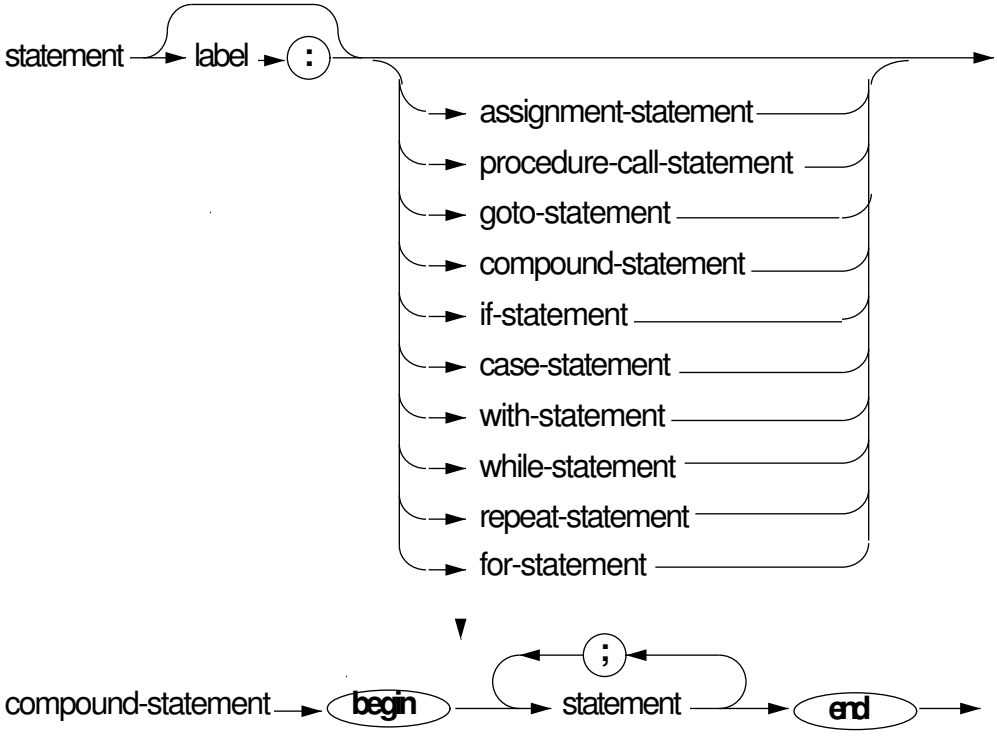


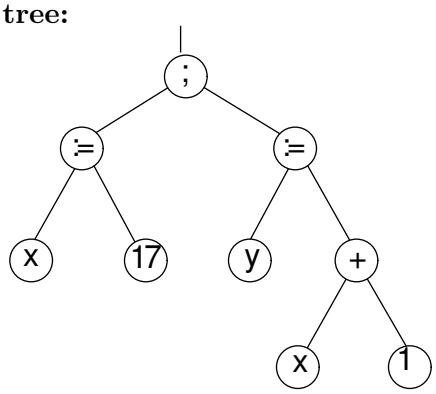
Exhibit 4.8. Translation: from source to machine code.

The object code was generated by OSS Pascal for the Atari ST.

Source code:

```
begin
x := 17;
y:= x+1
end
```

P-code tree:



Object code:

```
moveq #17,d0
move d0,x
addq #1,d0
move d0,y
```

4.3 Semantics

4.3.1 The Meaning of a Program

A modern language translator converts a program from its source form into a tree representation. This tree representation is sometimes called *p-code*, a shortening of *portable code*, because it is completely independent of hardware. This tree represents the *structure* of the program. The formal syntax of the language defines the kinds of nodes in the tree and how they may be combined. In this tree, the nodes represent objects and computations, and the structure of the tree represents the (partial) order in which the computations must be done. If any part of this tree is undefined or missing, the tree may have no meaning.

The formal semantics defines the meaning of this tree and, therefore, the *meaning* of the program. A language implementor must determine how to convert this tree to machine code for a specific machine so that his or her translation will have the same meaning as that defined by the formal semantics. This two-step approach is used because the conversion from source text to tree form can be the same for all implementations of a language. Only the second step, code generation, is hardware-dependent [Exhibit 4.8].

4.3.2 Definition of Language Semantics

The rules for interpreting the meaning of statements in a language are the semantics of the language. In order for a language to be meaningful and useful, the language designers, compiler writers, and programmers must share a common understanding of those semantics. If no single semantic standard exists, or no common understanding of the standard exists, various compiler writers will implement the language differently, and a programmer's knowledge of the language will not be transferable from one implementation to another. This is indeed the situation with both BASIC

and LISP; many incompatible versions exist.

Knowing the full syntax of a programming language is enough to permit an experienced person to make a guess about the semantics, but such a guess is at best rough, and it is likely to be wrong in many details and in some major ways. This is because highly similar syntactic forms in similar languages often have different semantics.

The syntax of a programming language needs only to describe all strings of symbols that comprise legal programs. To define the semantics, one must either define the results of some real or abstract computer executing the program, or write a complete set of mathematical formulas that axiomatize the operation of the program and the expected results. Either way, the definition must be complete, precise, correct, and nonambiguous. Neither kind of definition is easy to make.

The semantics of a language must thus define a highly varied set of things, including but not limited to:

- What is the "correct" interpretation of every statement type?
- What do you mean when you write a name?
- What happens during a function call?
- In what order are computations done?
- Are there syntactically legal expressions that are not meaningful?
- In what ways does a compiler writer have freedom?
- To what extent must all compilers produce code that computes the same answers?

In general, answering such questions takes many more pages than defining the syntax of a language. For example, syntax diagrams for Pascal can be printed in eight pages, three of which also contain extensive semantic information.[6] In contrast, a complete semantic description of Pascal, at a level that can be understood by a well-educated person, takes 142 pages.[7] Part of the reason for this difference is the dissimilarity between the meta-languages in which syntax and semantics are defined.

The semantics of natural languages are communicated to learners by a combination of examples and attempts to describe the meaning. The examples are required because an English description of semantics will lack precision and be as ambiguous as English. Similarly, English alone is not adequate to define the semantics of a programming language because it is too vague and too ambiguous to define highly complex things in such a way that no doubt remains about their meaning.

Just as it is possible to create a formal system such as EBNF to define language syntax, it is possible to create a formal system to define programming language semantics.[8] There is a major

[6] Dale and Lilly [1985], pages A1–A8.

[7] Cooper [1983].

[8] Historical note: The "Vienna Definition of PL/1" defined a new language for expressing semantics and defined the semantics of PL/1 in it. ALGOL-68 also had its own, impenetrable, formal language that tried to eliminate most of the need for a semantic definition by including semantics in the syntax. The result was a book-length syntax.

difference, though. The languages used to express syntax are relatively easy to learn and can be mastered by any student with a little effort. The languages used to express semantics are very difficult to read and extremely difficult to write.

The primary use for a formal semantic definition is to establish a single, unambiguous standard for the semantics of the language, to which all other semantic descriptions must conform. It defines all details of the meaning of the language being described and provides a precise answer to any question about details of the language, even details that were never considered by the language designer or semantics writer. Precision and completeness are more important for this purpose than readability, and formal semantic definitions are not easy to read.

A definition which only experts can read can serve as a standard to determine whether a compiler implements the standard language, but it is not really adequate for general use. Someone must study the definition and provide additional explanatory material so that educated nonexperts can understand it. Following is a quote from Cooper's Preface[9] which colorfully expresses the role of his book in providing a usable definition of Pascal semantics:

> The purpose of this manual is to provide a correct, comprehensive, and comprehensible reference for Pascal. Although the official Standard promulgated by the International Standards Organization (ISO) is 'correct' by definition, the precision and terseness required by a formal standard makes it quite difficult to understand. This book is aimed at students and implementors with merely human powers of understanding, and only a modest capacity for fasting and prayer in the search for the syntax or semantics of a *domain-type* or *variant selector*.

Cooper's book includes the definitions from the ISO standard and provides added explanatory material and examples. Compiler writers and textbook authors, in turn, can (but too many do not) use books such as *Standard Pascal* to ensure that their translations, explanations, and examples are correct.

4.3.3 The Abstract Machine

In order to make language definitions portable and not dependent on the properties of any particular hardware, the semantics of a computation tree must be defined in terms of an abstract model of a computer, rather than some specific hardware. Such a model has elements that represent the computer hardware, plus a facility for defining and using symbols. It forms a bridge between the needs of the human and computer. On one hand, it can represent symbolic computation, and on the other hand, the elements of the model are chosen so that they can be easily implemented on real hardware.

We describe an *abstract machine* here which we will use to discuss the semantics of many languages. It has five elements: the program environment, the stack, streams, the shared environment, and the control.

[9] Cooper [1983], p. ix.

This abstract machine resembles both the abstract machine underlying FORTH[10] and the SECD machine that Landin used to formalize the semantics of LISP.[11] Landin's SECD machine also has a stack and a control. Its environment component is our program environment, and our streams replace Landin's dump.

The FORTH model contains a dictionary which implements our program environment. FORTH has two stacks (for parameters and return addresses) which together implement our stack, except that no facility is provided for parameter names or local names.[12] The FORTH system defines input and output from files (our streams) and how a stream may be attached to a program. Finally, FORTH has an interpreter and a compiler which together define our control element.

Our abstract machine has one element, the shared environment, not present in either the FORTH model or the SECD machine, as those models did not directly support multitasking.

Program Environment. This environment is the context internal to the program. It includes global definitions and dynamically allocated storage that can be reached through global objects. It is the part of the abstract machine that supports communication between any nonhierarchically nested modules in a single program. Each function, F, exists in some symbolic context. Names are defined outside of F for objects and other functions. If these names are in F's program environment, they are known to F and permit F to refer to those objects and call those functions.

The program environment is implemented by a symbol table ("oblist" in LISP, "dictionary" in FORTH). When a symbol is defined, its name is placed in the symbol table, which connects each name to its meaning. Predefined symbols are also part of the environment. The meaning of a name is stored in some memory location, either when the name is defined or later. Either this space itself (as in FORTH) or a pointer to it (as in LISP) is kept adjacent to the name in the symbol table. Depending on the language, the meaning may be stored into the space by binding and initialization and/or it may be changed by assignment.

Shared Environment. This is the context provided by the operating system or program development "shell". It is the part of the abstract machine that supports communication between a program and the outside world. A model for a language that supports multitasking must include this element to enable communication between tasks. Shared objects are in the environment of two or more tasks but do not "belong" to any of them.

Objects that can be directly accessed by the separate, asynchronous tasks that form a job are part of the shared environment. Intertask messages are examples.

The Stack. The stack is the part of the computation model that supports communication between the enclosing and enclosed function calls that form an expression. It is a segmented structure of

[10] Brodie [1987], Chapter 9.

[11] Landin [1964].

[12] The dictionary in FORTH 83 is structured as a list of independent vocabularies, giving some support for local names.

theoretically unlimited size. The top stack segment, or frame, provides a local environment and temporary objects for the currently active function. This local environment consists of local names for objects outside the function (parameters) and for objects inside the function (local variables). Local environments for several functions can exist simultaneously and will not interfere with each other. Suspension of one function in order to execute another is possible, with later reactivation of the first in the same state as when it was suspended.

The stack is implemented by a stack. A stack pointer is used to point at the stack frame (local environment) for the current function, which points back to a prior frame. A frame for a function F is created above the prior frame upon entry to F, and is destroyed when F exits. Storage for function parameters and a function return address are allocated in this frame and initialized (and possibly later removed) by the calling program.

Upon entry to F, the names of its parameters are added to the local environment by binding them to the stack locations that were set up by the calling program. The local symbols defined in F are also added to the environment and bound to additional locations allocated in F's stack frame. The symbol table is managed in such a way as to permit these names to be removed from the environment upon function exit.

Streams. Streams are one medium of communication between different tasks that are parts of a job. A program exists in the larger context of a computer system and its files. The abstract machine, therefore, must reflect mass storage and ways of achieving data input and output. A *stream* is a model of a sequential file, as seen by a program. It is a sequence, in time, of data objects, which can be either read or written. Symbolic names for streams and for the files to which they are bound must be part of the program environment.

The concept of a stream is actually more general than the concept of a sequential file. Suppose two tasks are running concurrently on a computer system, and the output stream of one becomes the input stream of the other. A small buffer to hold the output until it is reprocessed can be enough to implement both streams.

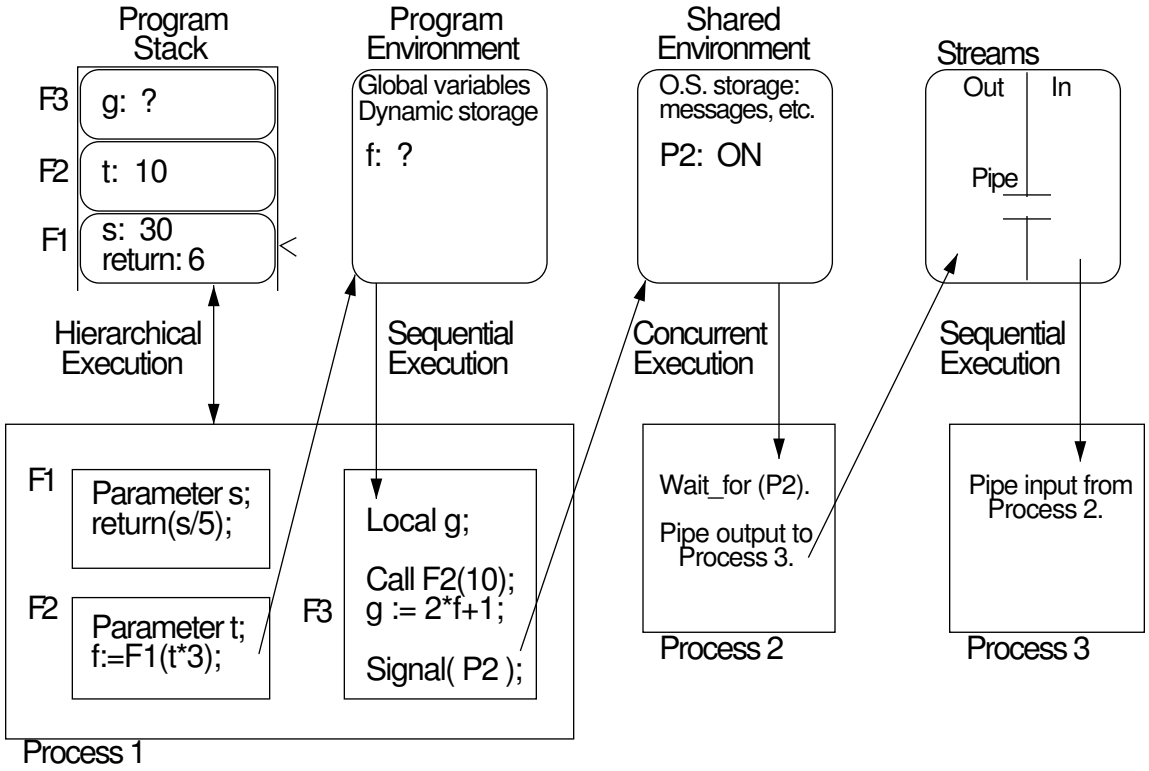
Control. The control section of the abstract model implements the semantic rules of the language that define the order in which the pieces of the abstract computation tree will be evaluated. It defines how execution of statements and functions is to begin, proceed, and end, including the details of sequencing, conditional execution, repetition, and function evaluation. (Chapter 8 deals with expressions and function evaluation, and Chapter 10 deals with control statements.)

Three kinds of control patterns exist: functional, sequential, and asynchronous.[13] These patterns are supported in various combinations in different languages. Each kind of control pattern is associated with its own form of communication, as diagrammed in Exhibit 4.9.

Functional control elements communicate with each other by putting parameters on the stack and leaving results in a return register. In the diagram, functions F1, F2, and F3 are all part of Process_1 and have associated stack frames on the stack for Process_1. When F3 is entered, its

[13] Developed fully in Chapter 8.

Exhibit 4.9. Communication between modules.



stack frame is created. Then when F3 calls F2 and F2 calls F1, frames for F2 and F1 are created on the stack. The frame for F1, indicated by a "<", is the "current" frame. Parameters are initialized during the function-calling process. When F1 returns it will return a 6 to F2.

Functions within the same process share access to global variables in the program environment for that process. Sequential constructs in these functions communicate by assigning values to these variables. Function F2 communicates with F3, and sequential statements in F3 communicate with each other through the global variable named "f" in the program environment. F1 will return the value 6 to F2, which will assign it to a global variable, f. This variable is accessible to F3, which will use its value to compute g.

Concurrent tasks communicate through the shared environment. Process_1 and Process_2 share asynchronous, concurrent execution and synchronize their operations through signals left in the shared environment.

Sequential tasks communicate through streams. The output from Process_2 becomes the input for Process_3. To implement this, the operating system has connected their output and input

streams through an operating system "pipe". This pipe could be implemented either by conveying the data values to Process_3 as soon as they are produced by Process_2 or by storing the output in a buffer or a file, then reading it back when the stream is closed.

A Semantic Basis. The formal semantic definition of a language must include specific definitions of the details of the abstract machine that implements its semantics. Different language models include and exclude different elements of our abstract machine. Many languages do not support a shared environment. The new functional languages do not support a program environment, except for predefined symbols. The control elements, in particular, differ greatly from one language to the next.

We define the term *semantic basis of a language* to mean the specific version of the abstract machine that defines the language, together with the internal data structures and interpretation procedures that implement the abstract semantics. Layered on top of the semantic basis is the syntax of the language, which specifies the particular keywords, symbols, and order of elements to be used to denote each semantic unit it supports.

The semantic basis of a language must define the kinds of objects that are supported, the primitive actions, and the control structures by which the objects and actions are linked together, and the ways that the language may be extended by new definitions. The features included in a semantic basis completely determine the power of a language; items left out cannot be defined by the programmer or added by using macros. Where two different semantic units provide roughly the same power, the choice of which to include determines the character of the language and the style of programs that will be written in it. Thus a wise language designer gives careful thought to the semantic basis before beginning to define syntax.

4.3.4 Lambda Calculus: A Minimal Semantic Basis

It is perhaps surprising that a very small set of semantic primitives, *excluding goto and assignment*, can form an adequate semantic basis for a language. This was proven theoretically by Church's work on lambda calculus.[14]

Lambda calculus is not a programming language and is not directly concerned with computers. It has no programs or objects or execution as we understand them. It is a symbolic, logical system in which formulas are written as strings of symbols and manipulated according to logical rules.

We need to be knowledgeable about lambda calculus for three reasons. First, it is a *complete* system: Church has shown that it is capable of representing any computable function. Thus any language that can implement or emulate lambda calculus is also complete.

Second, lambda calculus gives us a starting point by defining a minimal semantic basis for computation that is mathematically clean. As we examine real computer languages we want to distinguish between necessary features, nice features (extras), nonfeatures (things that the language

[14] Church [1941].

Exhibit 4.10. Lambda calculus formulas.

Formulas	Comments
x	Any variable is a formula.
$(\lambda x.((yy)x))$	Lambda expressions are formulas.
$(\lambda z.(y(\lambda z.z)))$	The body of this lambda expression is an application.
$((\lambda z.(zy))x)$	Why is this formula an application?

would be better off without), and missing features which limit the power of the language. The lambda calculus gives us a starting point for deciding which features are necessary or missing.

Finally, an extended version of lambda calculus forms the semantic basis for the modern functional languages. The Miranda compiler translates Miranda code into tree structures which can then be interpreted by an augmented lambda calculus interpreter. Lambda calculus has taken on new importance because of the recent research on functional languages. These languages come exceedingly close to capturing the essence of lambda calculus in a real, translatable, executable computer language. Understanding the original formal system gives us some grasp of how these languages differ from C, Pascal, and LISP, and supplies some reason for the aspects of functional languages that seem strange at first.

Symbols, Functions, and Formulas

There are two kinds of symbols in lambda calculus:

- A single-character symbol, such as y, used to name a parameter and called a *variable*.
- Punctuation symbols '(', ')', '.', and 'λ'.

These symbols can be combined into strings to form *formulas* according to three simple rules:

1. A variable is a formula.
2. If y is a variable and F is a formula, then $(\lambda y.F)$ is a formula, which is called a *lambda expression*; y is said to be the *parameter* of the lambda expression, and F is its *body*.
3. If F and G are formulas, then (FG) is a formula, which is called an *application*.

Thus every lambda calculus formula is of one of three types: a variable, a lambda expression, or an application. Examples of formulas are given in Exhibit 4.10.

Lambda calculus differs from programming languages in that its programs and its semantic domain are the same. Formulas can be thought of as programs or as the data upon which programs operate. A lambda expression is like a function: it specifies a parameter name and has a body that usually refers to that parameter.[15] An application whose first formula is a lambda expression is like

[15]The syntax defined here supports only one-argument functions. There is a common variant which permits multiargument functions. This form can be mechanically converted to the single-argument syntax.

Exhibit 4.11. Lambda calculus names and symbols.

Formulas	Comments
x, y, z, etc.	Single lowercase letters are variables.
G = $(\lambda x.(y(yx)))$	A symbolic name may be defined to stand for a formula.
H = (GG)	Previously defined names may be used in describing formulas.

a function call—the function represented by the lambda expression is called with the second formula as an argument. Thus $((\lambda x.F)G)$ intuitively means to call the function $(\lambda x.F)$ with argument G. However, not all formulas can be interpreted as programs. Formulas such as (xx) or $(y(\lambda x.z))$ do not specify a computation; they can be thought of as data.

In order to talk about lambda formulas, we will often give them symbolic names. To avoid confusing our names, which we use to talk *about* formulas, with variables, which *are* formulas, we use uppercase letters when naming formulas. As a shorthand for the statement, "let F be the formula $(\lambda x.(yx))$", we will write simply $F = (\lambda x.(yx))$. If we then write a phrase like, "the formula (Fz) is an application", the formula we are talking about is $((\lambda x.(yx))z)$. In general, wherever F appears, it should be replaced by its definition. Since names are just a shorthand for formulas, a circular "definition" such as $F = (\lambda x.(yF))$ is meaningless. Examples of symbols and definitions are shown in Exhibit 4.11.

As another shorthand, when talking about formulas, we may omit unnecessary parentheses. Thus we may write $\lambda x.y$ instead of $(\lambda x.y)$. In general, there may be more than one way to insert parentheses to make a meaningful formula. For example, $\lambda x.yx$ might mean either $(\lambda x.(yx))$ or $((\lambda x.y)x)$. We use the rules that the body of a lambda expression extends as far to the right as possible, and sequences associate to the left. Thus, in the above example, the body of the lambda expression is yx, so the fully parenthesized form is $(\lambda x.(yx))$. Examples of these rules are given in Exhibit 4.12.

Free and Bound Variables. A parameter name is a purely local name. It *binds* all occurrences of that name on the right side of the lambda expression. A symbol on the right side of a lambda

Exhibit 4.12. Omitting parentheses when writing lambda calculus formulas.

Shorthand	Meaning
fxy	$((fx)y)$
$\lambda x.\lambda y.x$	$(\lambda x.(\lambda y.x))$
$\lambda x.x\lambda y.y$	$(\lambda x.(x(\lambda y.y)))$
$(\lambda x.(xx))(zw)$	$((\lambda x.(xx))(zw))$
$\lambda x.\lambda y.yzw$	$(\lambda x.(\lambda y.((yz)w)))$

Exhibit 4.13. Lambda expressions for TRUE and FALSE.

Expressions	Comments
$T = \lambda x.\lambda y.x$	The symbol "T" represents the logical value TRUE. You should read the definition of T as follows: T is a function of parameters x and y. Its body ignores y and returns x. (We say the argument y is "dropped".)
$F = \lambda x.\lambda y.y$	"F" names the lambda expression which represents FALSE.

expression is *bound* if it occurs as a parameter, immediately following the symbol λ, on the left side of the same expression or of an enclosing expression. The scope of a binding is the entire right side of the expression. In Exhibit 4.14, the λx defines a local name and binds all occurrences of x in the expression. We say that each bound occurrence of x refers to the particular λx that binds it.

An occurrence of a variable x in F is *free* if x is not bound. Thus the occurrence of p in $(\lambda y.py)$ is free, but the occurrence of y in that same formula is bound (to λy). In the formula $(x(\lambda x.((\lambda x.x)x)))$, the variable x occurs five times. The second and third occurrences are bindings; the other three occurrences are uses. The first occurrence is free, since it does not lie within the scope of any λx-expression. The fourth occurrence is bound to the third occurrence, and the fifth occurrence is bound to the second occurrence.

These binding rules are the familiar scoping rules of block-structured programming languages such as Pascal. The operator λx declares a new instance of x. All occurrences of x within its scope refer to that instance, unless x is redeclared by a nested λx. In other words, an occurrence of a variable is always bound to the innermost enclosing block in which x is declared.

Representing Computation

Church invented a way to use lambda formulas to represent computation. He assigned interpretations to certain formulas, making them represent the basic elements of computation. (Some, but not all, lambda expressions have useful interpretations.) The formulas shown in this chapter are some of the most basic in Church's system, including formulas that represent truth values [Exhibit 4.13], the integers [Exhibit 4.15], and simple computations on them [Exhibit 4.16]. More advanced formulas are able to represent recursion. As you work through these examples the purpose and mechanics of these basic definitions should become clearer.

Now that we know what lambda calculus formulas are, we need to talk about what they do. Evaluation rules allow one formula to be transformed to another. A formula which cannot be transformed further is said to be in *normal form*. The meaning of a formula is its normal form, if it has one; otherwise, the formula is undefined. An undefined formula corresponds to a nonterminating computation. Exhibit 4.14 dissects an expression and looks at its parts.

Exhibit 4.14. Dissection of a lambda expression.

A lambda expression, with name: $2 = \lambda x.\lambda y.x(xy)$

Useful interpretation: the number two

Breakdown of elements

$2 =$	Declares the symbol "2" to be a name for the following expression.
$\lambda x.$	The function header names the parameter, "x". Everything that follows this "." is the expression body.
$\lambda y.x(xy)$	The body of the original expression is another expression with a parameter named "y". Parameter names are purely arbitrary; this expression would still have the same meaning if it were rewritten with a different parameter name, as in: $\lambda q.x(xq)$
$x(xy)$	This is the body of the inner expression. It contains a reference to the parameter "y" and also references to the parameter "x" from the enclosing expression.

Reduction. Consider a lambda expression which represents a function. At the abstract level, the meaning, or semantics, of the expression is the mathematical function that it computes when applied to an argument. Intuitively, we want to be able to freely replace an expression by a simpler expression that has the same meaning. The rules for beta and eta reduction permit us to do so.

The main evaluation rule for lambda calculus is called *beta reduction* and it corresponds to the action of calling a function on its argument. A *beta reducible expression* is an application whose left part is a lambda expression. We also use the term *beta redex* as a shortening of "reducible expression". When a lambda expression is applied to an argument, the argument formula is substituted for the bound variable in the body of the expression. The result is a new formula.

A second reduction rule is called *eta reduction*. Eta reduction lets us eliminate one level of binding in an expression of the form $\lambda x.f(x)$. In words, this is a special case in which the lambda argument is used only once, at the end of the body of the expression, and the rest of the body is a lambda expression applied to this parameter. If we apply such an expression to an argument, one beta reduction step will result in the simpler form $f(x)$. Eta reduction lets us make this transformation *without supplying an argument.* Specifically, eta reduction permits us to replace any expression of the form $\lambda x.f(x)$, where f represents a function, by the single symbol f.

After a reduction step, the new formula may still contain a redex. In that case, a second reduction step may be done. When the result does not contain a beta-redex or eta-redex, the reduction process is complete. We say such a formula is in *normal form.*

Many lambda expressions contain nested expressions. When such an expression is fully parenthesized it is clear which arguments belong to which function. When parentheses are omitted, remember that *function application associates to the left*; that is, the leftmost argument is substituted first for the parameter in the outermost expression.

We now describe in more detail how reduction works. When we reduce a formula (or subformula) of the form $H = ((\lambda x.F)G)$, we replace H by the formula F', where F' is obtained from F by

Exhibit 4.15. Lambda calculus formulas that represent numbers.

$$\begin{aligned} 0 = &\quad \lambda x.\lambda y.y \\ 1 = &\quad \lambda x.\lambda y.xy \\ 2 = &\quad \lambda x.\lambda y.x(xy) \end{aligned}$$

The formula for zero has no occurrences of its first parameter in its body. Note that it is the same as the formula for F. Zero and False are also represented identically in many programming languages.

The formula for the integer one has a single x in its body, followed by a y. The formula for two has two x's. The number n will be represented by a formula in which the first parameter occurs n times in succession.

substituting G for each reference to x in F. Note that if F contains another binding λx, the references to *that* binding are not replaced. For example, $((\lambda x.xy)(zw))$ reduces to $((zw)y)$ and $((\lambda x.x(\lambda x.(xy)))(zz))$ reduces to $(zz)(\lambda x.(xy))$.

When an expression containing an unbound symbol is used as an argument to another lambda expression, special care must be taken. Any occurrence of a variable in the argument that was free before the substitution must remain free after the substitution. It is not permitted for a variable to be "captured" by an unrelated λ during substitution. For example, it is *not* permitted to apply the reduction rule to the formula $((\lambda x.(\lambda y.x))(zy))$, since y is free in (zy), but after substitution, that occurrence of y would not be free in $(\lambda y.(zy))$. To avoid this problem, the parameter must be renamed, and all of its bound occurrences must be changed to the new name. Thus $((\lambda x.(\lambda y.x))(zy))$ could be rewritten as $((\lambda x.(\lambda w.x))(zy))$, after which the reduction step would be legal.

Examples of Formulas and Their Reductions

The formulas T and F in Exhibit 4.13 accomplish the equivalent of branching by manipulating their parameters. They take the place of the conditional statement in a programming language. T (true) returns its first argument and discards the second. Thus it corresponds to the `IF..THEN` statement which evaluates the `THEN` clause when the condition is true. Similarly, the formula F (false) corresponds to the `IF..ELSE` clause. It returns its second parameter just as an `IF` statement evaluates the second, or `ELSE` clause, when the condition is false.

The *successor function*, S, applied to any integer, gives us the next integer. Exhibit 4.16 shows the lambda formula that computes this function. Given any formula for a number n, it returns the formula for $n + 1$. The function *ZeroP* (zero predicate) tests whether its argument is equal to the formula for zero. If so, the result is T, if not, F. Exhibit 4.17 shows how we would call S and *ZeroP*. The process of carrying out these computations will be explained later.

Church was able to show that lambda calculus can represent all computation, by representing

Exhibit 4.16. Basic arithmetic functions.

The successor function for integers. Given the formula for any integer, n, this formula adds one x and returns the formula for the next larger integer.

$$S = \lambda n.(\lambda x.\lambda y.nx(xy))$$

Zero predicate. This function returns T if the argument = 0 and F otherwise.

$$ZeroP = \lambda n.n(\lambda x.F)T$$

numbers, conditional evaluation, and recursion. Crucial to the power of his system is that there is no distinction between objects and functions. In fact, "objects", in the sense of data objects, were not defined at all. Expressions called "normal forms" take their place as concrete things that exist and can be tested for identity. A formula is in normal form if it contains no redexes. Not all formulas have a normal form; some may be reduced infinitely many times. These formulas, therefore, do not represent objects. They are the analog of infinite recursions in computer languages.

For example, let us define the symbol "*twin*" to be a lambda expression that duplicates its parameter:

$$twin = \lambda x.xx$$

The function "twin" can be applied to itself as an argument. The application looks like this:

$$(twin \;\; twin)$$

The preceding line shows this application symbolically. Now we rewrite this formula with the

Exhibit 4.17. Some lambda applications.

An application consists of a function followed by an argument. The first three applications listed here use the number symbols defined in Exhibit 4.15 and the function symbols defined in Exhibit 4.16. These three applications are evaluated step-by-step in Exhibits 4.18, 4.19, and 4.20.

$(S \;\; 1)$	Apply the successor function to the function 1.
$(ZeroP \;\; 0)$	Apply $ZeroP$ to 0 (Does 0 = zero?)
$(ZeroP \;\; 1)$	Does 1 = zero?
$((GH) \;\; x)$	Apply formula G to formula H, and apply the result to x.

The last application has the same meaning when written without the parentheses: "GHx".

name of the function replaced by its definition. Parentheses are used, for clarity, to separate expressions:

$$((\lambda x.xx)(twin))$$

This formula contains a redex and so it is not in normal form. When we apply the reduction rule, the function, $\lambda x.xx$, makes two copies of its parameter, giving:

$$(twin \ \ twin)$$

Thus the result of reduction is the same as the formula we started with! Clearly, a normal form can never be reached.

Higher-Order Functions

If lambda calculus were a programming language, we would say that it treats functions as *first-class objects* and supports *higher-order functions*. This means that functions may take functions as parameters and return functions as results. With this potential we can do some highly powerful things.

We can define a lambda expression, F, to be the *composition* of two other expressions, say G and H. (This means that F is the expression produced by applying G to the result of H.) This cannot be done in most programming languages. C, for example, permits you to *execute* a function G on the result of *executing* H. But C does not let you write a function that takes two functional parameters, G and H, and returns a function, F, that will later accept some argument and apply first H to it and then apply G to the result.

A formula that implements recursion can be defined as the composition of two higher-order functions. Thus lambda calculus does not need to have recursion "built in"; it can be defined within the system. In contrast, recursion is, and must be, "built into" C and Pascal.

A language with higher-order functions also permits one to *curry* a function. G is a *currying* of F if G has one fewer parameter than F and computes its result by calling F with a constant in place of the omitted parameter. Currying, combined with generic dispatching,[16] is one way to implement functions with optional arguments.

Evaluation / Reduction

Any model of computation must represent action as well as objects. Actions are represented in the lambda calculus by applying the *reduction rule*, which requires applying the renaming and substitution rules.

To reduce a formula, F, one finds a subformula, S, anywhere within F, that is reducible. To be reducible, S must consist of a lambda expression, L, followed by an argument, A. The reduction process then consists of two steps: renaming and substitution.

[16] See Chapter 18.

Exhibit 4.18. Reducing $(S\ 1)$**.**

Compute the Successor of 1. The answer should be 2. For clarity, the formula for one has been written using p and q instead of x and y. (This is, of course, permitted. The symbols that are used for *bound* variables may be renamed any time.)

Write out S.	$(\lambda n.(\lambda x.\lambda y.nx(xy))1)$
Substitute 1 for n, reduce.	$(\lambda x.\lambda y.1x(xy))$
Write out the definition of 1.	$(\lambda x.\lambda y.(\lambda p.\lambda q.pq)x(xy))$
Substitute x for p, and reduce.	$(\lambda x.\lambda y.(\lambda q.xq)(xy))$
Substitute (xy) for q, reduce.	$(\lambda x.\lambda y.x(xy))$

The answer is the formula for 2, which is, indeed, the successor of 1.

Renaming. Renaming is required only if unbound symbols occur in A. They must not have the same name as L's parameter. If such a name conflict occurs, the parameter in L must be renamed so that the unbound symbol will not be "captured" by L's parameter. The new name may be any symbol whatsoever. The formula for L is simply rewritten with the new symbol in place of the old one.

Substitution. After renaming, each parameter reference on the right side of L is replaced by a copy of the entire argument-expression, and the resulting string replaces the subexpression S. The λ, the dummy parameter, and the "." are dropped.

Exhibits 4.18, 4.19, and 4.20 illustrate the reduction process. Three simple formulas are given and reduced until they are in normal form. The comments on the left in these exhibits document each choice of redex and the corresponding substitution process. The following explanations are given so that you may develop some intuition about how these functions work.

Successor. Intuitively, the successor function must take a numeric argument (a nest of two lambda expressions) and insert an additional copy of the outermost parameter into the middle of the formula. This is accomplished as follows:

- On the first reduction step, the formula for S embeds its argument, n, in the middle of a nested lambda expression. The symbols x and y in the formula for S are bound by the lambdas at the left. We rename the bound variables in the formula for n to avoid confusion; during the reduction process, this p and q will be eliminated.
- The formula for n now forms a redex with the x in the tail end of the formula for S. Reducing this puts as many copies of x into the result as there were copies of p in n. Remember, we want to end up with exactly one additional copy of x.
- This added x comes from the (xy) at the right of the formula for S. The result of the preceding

Exhibit 4.19. Reducing (*ZeroP* 0)**.**

Apply *ZeroP* to 0, that is, determine whether 0 equals zero. The answer should be T.

Write out *ZeroP* followed by 0.	$((\lambda n.n(\lambda x.F)T)0)$
Substitute 0 for n in the body of *ZeroP* and reduce.	$(0(\lambda x.F)T)$
Write out the formula for zero.	$((\lambda x.\lambda y.y)(\lambda x.F)T)$
Substitute $(\lambda x.F)$ for x, and reduce.	$((\lambda y.y)T)$
Substitute T for y, reduce.	T

So 0 does equal 0. Note that the argument, $(\lambda x.F)$, was dropped in the fourth step because the parameter, x, was not referenced in the body of the function.

reduction forms a redex with this (xy). When we reduce, this final x is sandwiched between the other x's and the y, as desired.

Essentially, the y in a number is a "growth bud" that permits any number of x's to be appended to the string. It would be easy, now, to write a definition for the function "plus2".

Zero predicate. Remember, 0 and F are represented by the same formula. Thus the zero predicate must turn F into T and any other numeric formula into F. (The behavior of *ZeroP* on nonnumeric arguments is undefined. Applying *ZeroP* to a nonnumber is like a type error.) Briefly, the mechanics of this computation work as follows:

- An integer is represented by a formula that is a nest of two lambda expressions.
- *ZeroP* takes its argument, n, and appends two expressions, $\lambda x.F$ and T, to n. These two

Exhibit 4.20. Reducing (*ZeroP* 1)**.**

Write out *ZeroP* followed by 1.	$((\lambda n.n(\lambda x.F)T)1)$
Substitute 1 for n, reduce.	$(1(\lambda x.F)T)$
Write out the formula for 1.	$((\lambda x.\lambda y.xy)(\lambda x.F)T)$
Substitute $(\lambda x.F)$ for x, reduce.	$((\lambda y.(\lambda x.F)y)T)$
Substitute T for y, reduce.	$((\lambda x.F)T)$
Substitute T for x and reduce.	F

On the last line, the parameter x does not appear in the body of the function, so the argument, T, is simply dropped. So 1 does not equal 0.

Applying *ZeroP* to any nonzero number would give the same result, but involve one more reduction step for each x in the formula.

Exhibit 4.21. A formula with three redexes.

Assume that $P3$ (which adds 3 to its argument) and $*$ (which computes the product of two arguments) have already been defined. (They can be built up out of the successor function.) Then the formula

$$(* \ (P3 \ 4) \ (P3 \ 9))$$

has three reducible expressions: $(P3\ 4)$, $(P3\ 9)$, and $(* \ (P3\ 4)\ (P3\ 9))$.

expressions form arguments for the two lambda expressions in n. The entire unit forms two nested applications.

- We reduce the outermost lambda expression first, using the argument $\lambda x.F$. If n is 0, this argument is "discarded" because the formula for zero does not contain a reference to its parameter. For nonzero arguments, this expression is kept.
- The inner expression (from the original argument, n) forms an application with the argument T. If n was zero, this reduces immediately to T. If n was nonzero, there is one more reduction step and the result is F.

The Order of Reductions

Not every expression has a normal form; some can be reduced forever. But if a normal form exists it can always be reached by some chain of reductions. When each lambda expression in a formula is nested fully within another, only one order of reduction is possible—from the outside in. But it is possible to have a formula with two reducible lambda expressions at the same level, side by side [Exhibit 4.21]. Further, whatever redex you select next, the normal form can still be reached. Put informally, you cannot back yourself into a corner from which you cannot escape. This important result is named the "Church-Rosser Theorem" after the logicians who formally proved it.

Some expressions that do have normal forms contain subexpressions that cannot be reduced to normal form. This seems like a contradiction until you realize that, in the process of evaluation, whole sections of a formula may be "discarded". For example, in a conditional structure, either the "then part" or the "else part" will be skipped. The computation enclosing the conditional can still terminate successfully, even if the part that is skipped contains an infinite computation.

By the Church-Rosser theorem, a normal form, if it exists, can be reached by reducing subformulas in any order until there are no reducible subformulas left. However, although you cannot get "blocked" in reducing such an expression, you can waste an infinite amount of effort if you persist in reducing a nonterminating part of the formula. Since any subformula may be discarded by a conditional, and never need to be evaluated, it is wiser to postpone evaluating a sub-expression until it is needed. If, eventually, a non-terminating sub-formula must be evaluated, then the formula has no normal form. If, on the other hand, it is "discarded", the formula in which this infinite

computation was embedded can still be computed (reduced to normal form).

A further theorem proves that *if* a normal form can be reached, then it can be reached using the outside-in order of evaluation. That is, at each step the outermost possible redex is chosen. (The formulas in Exhibits 4.20, 4.19, and 4.18 were all reduced in outside-in order.) This order is called the *normal order of evaluation* in lambda calculus and corresponds to *call-by-name reduction order* in a programming language.[17] It may not be a unique order, since sometimes the outermost formula is not reducible, but may contain more than one redex side-by-side. In that case, either may be reduced first.

The Relevancy of Lambda Calculus

Lambda calculus has been proven to be a fully general way to symbolize any computable formula. Its semantic basis contains representations of objects (normal forms) and functions (λ expressions). Because functions *are* objects, and higher-order functions can be constructed, the system is able to represent conditional branching, function composition, and recursion. Computation is represented by the process of reduction, which is defined by the rules for renaming, parameter substitution, and formula rewriting.

Although lambda calculus is a formal logical system for manipulating formulas and symbols, it provides a model of computation that can be and has been used as a starting point for defining programming languages. LISP was originally designed to be an implementation of lambda calculus, but it did not capture the outside-in evaluation semantics.

4.4 Extending the Semantics of a Language

Let us define an *extension* to be a set of definitions which augment a language with an entirely new facility that can be used in the same way that preexisting facilities are used. Some of the earliest languages were not very extensible at all. The original FORTRAN allowed variables to be defined but not types or functions (in a general sense). Function definitions were limited to one line. All modern languages are extensible in many ways. Any time we define a new object, a new function, or a new data type, we are extending the language. Each such definition extends the list of words that are meaningful and adds new expressive power. Pascal, LISP, and the like. are extensible in this sense: by building up a vocabulary of defined functions and/or procedures, we ultimately write programs in a language that is much more extensive and powerful than the bare language provided by the compiler.

Historically, we have seen that extensibility depends on uniform, general treatment of a language feature. Any time a translator is designed to recognize a specific, fixed set of keywords or defined symbols, that portion of the language is not extensible. The earliest BASIC was not extensible at all; even variable names were all predefined (only two-letter names were permitted). FORTRAN, one of the earliest computer languages, can help us see how the design of a language and a translator

[17] See chapter 9, Section 9.2.

can create barriers to extensibility. We will look at types and functions in early FORTRAN and contrast them to the extension facilities in more modern languages.

Early FORTRAN supported a list of predefined mathematical functions. The translator recognized calls on those predefined functions, but users could not define their own. This probably happened because the designers/implementors of FORTRAN provided a static, closed list of function names instead of simply permitting a list that could grow. The mechanics of translating a function call are also simpler if only one- and two-argument functions have to be supported, rather than argument lists of unlimited size.

In contrast, consider early LISP. Functions were considered basic (as lambda expressions are basic in lambda calculus), and the user was expected to define many of them. The language as a whole was designed to accept and translate a series of definitions and enter each into an extensible table of defined functions. The syntax for function calls was completely simple and modeled after lambda calculus, which was known to be completely general. LISP was actually easier to translate than FORTRAN.

Consider type extensions. In FORTRAN, there were two recognized data types, real and integer. These were "hard wired" into the language: variables whose names started with letters "I" through "N" were integers, all other variables were real. On the implementation level, FORTRAN parsers were written to look at each variable name and deduce the type from it. This was certainly a convenient system, since it made declarations unnecessary, but it was not extensible. The system fell apart when FORTRAN was extended to support alphabetic data and double-precision arithmetic.

In contrast, look at Pascal. Pascal has four primitive data types and several ways to build new simple and aggregate types out of the primitive types. The language has a clear notion of what a type is, and when a new type is or is not constructed. Each time the programmer uses a type constructor, a new type is added to the list of defined types. Thereafter, the programmer may use the new type name in exactly the same ways that primitive type names may be used.

Although Pascal types are extensible, there are predefined, nonextensible relationships among the predefined types, just as there are in FORTRAN. Integers may be converted to reals, and vice versa, under specific, predefined circumstances. These conversion relationships are nonextensible; the triggering circumstances cannot be modified, and similar conversion relationships for other types cannot be defined. Object-oriented languages carry type-extensibility one step farther, permitting the programmer to define relationships between types and extend the set of situations in which a conversion will take place. This is accomplished, in C++ for example, by introducing the notion of a "constructor function", which builds a value of the target type out of components of the original type. The programmer may define her or his own constructors. The translator will use those constructors to avoid a type error under specified circumstances, by converting an argument of the original type to one of the target type.

In all the cases described here, extension is accomplished by allowing the programmer to define new examples of a semantic category that already exists in the translator. To enable extension, a new syntax is provided for defining new instances of existing categories. However, the programmer writes the same syntax for using an extension as for using a predefined facility. Old categories are extended; entirely new things are not added. Some languages, those with macro facilities, allow

the programmer to extend the language by supplying new notation for existing facilities. However, very few languages support additions or changes to the basic syntactic structure or the semantic basis of the language. Changing the syntactic structure would involve changing the parser, which is normally fixed. Changing the semantic basis would involve adding new kinds of tables or procedures to the translator to implement the new semantics.

What would it mean to extend the syntactic structure of a language? Consider the `break` instruction in C and the `EXIT` in Ada. These highly useful statements enable controlled exits from the middle of loops. Pascal does not have a similar statement, and an exit from the middle of a loop can be done only with a `GOTO`. But the `GOTO` lacks the safely controlled semantics of `break` and `EXIT`. Because it is so useful, `EXIT` is sometimes added to Pascal as a nonstandard extension. Doing this involves extending the parsing phase of the compiler to recognize a new keyword and modifying the code generation phase to generate a branch from the middle of a loop to the first statement after the loop. Of course, a programmer cannot extend a Pascal compiler like this. It can only be done when the compiler is being written.

The ANSI C dialect and the language C++ are both semantic extensions of C. ANSI C extended the original language by adding type checking for function calls and some coherent operations on structured data. C++ adds, in addition, semantically protected modules (classes), virtual functions, and polymorphic domains. This kind of semantic extension is implemented by changing the compiler and having it do work of a different nature than is done by an old C compiler. These extensions mentioned required modifying the process of translating a function call, adding new information to the symbol table, implementing new restrictions on visibility, and adding type checking and type conversion algorithms.

The code and tables of a compiler are normally "off-limits" to the ordinary language user. In most languages, a programmer cannot access or change the compiler's tables. The languages EL/1, FORTH, and T break this rule; EL/1[18] permitted additions to the compiler's syntactic tables, with accompanying semantic extensions, and FORTH permits access to the entire compiler, including the symbol table and the semantic interpretation mechanisms.

EL/1 (Extensible Language 1) actually permitted the programmer to supply new EBNF syntax rules and their associated interpretations. The translator included a preprocessor and a compiler generator which combined the user-supplied syntax rules with the built-in ones and produced a compiler for the extended language. The semantic interpretations for the new syntactic rules, supplied by the user, were then used in the code generation phase.

A very similar thing can be done in T. T is a semantic extension of Scheme which includes data structuring primitives, object classes, and a macro preprocessor which can be used to extend the syntax of the language. Each preprocessor symbol is defined by a well-formed T expression. With these tools, extensions can be constructed that are not possible in C, Pascal, or Scheme. We could, for example, use the macro facility to define the syntax for a `for loop` expression and define the semantics to be a complex combination of initializations, statement executions, increments, and result-value construction.

[18] Wegbreit [1970].

4.4.1 Semantic Extension in FORTH

We use FORTH to demonstrate the kind of extension that can be implemented by changing the parser and semantic interpretation mechanisms of a translator. Two kinds of limited semantic extension are possible in FORTH:

- We may add new kinds of information to the symbol table, with accompanying extensions to the interpreter.
- We may modify the parser to translate new control structures.

We shall give an example of each kind of extension below. In both cases, the extension is accomplished by using knowledge of the actual implementation of the compiler and accessing tables that would (in most compilers) be protected from user tampering. FORTH has several unusual features that make it possible to do this kind of extension.

First, like LISP, FORTH is a small, simple language with a totally simple structure. FORTH books explain the internal structure of the language and details of the operation of the compiler and interpreter. Second, the designers of FORTH anticipated the desire to extend the rather rudimentary language and included extension primitives, the words "`CREATE`" and "`DOES>`", that denote a compiler extension, and the internal data structures to implement them.

Finally, FORTH is an interpretive language. The compiler produces an efficient intermediate representation of the code, not native machine code. Control changes from the interpreter to the compiler when the interpreter reaches the ":" at the beginning of a definition, and switches back to the interpreter when the compiler reaches the ";" at the end of the definition. Words are also included that permit one to suspend a compilation in the middle, interpret some code, and return to the compilation. Thus variable declarations, ordinary function definitions, segments of code to be interpreted, and extensions to the compiler can be freely intermixed. The only requirement is that everything be defined before it is used.

New Types. Unextended, FORTH has three semantic categories, or data types, for items in the dictionary (symbol table): constant, variable, and function. By using the words `CREATE` and `DOES>` inside what otherwise looks like a normal function definition, more types can be added. `CREATE` enters the name of the new type category into the dictionary. Following it must be FORTH code for any compile-time actions that must be taken to allocate and/or initialize the storage for this new type. This compile-time section is terminated by the `DOES>`, which marks this partial entry as a new semantic category. Finally, the definition includes FORTH code for the semantic routine that should be executed at run time when items in this category are referenced [Exhibit 4.22].

Having added a type, the FORTH interpreter can be extended to check the type of a function parameter and dispatch (or execute) one of several function methods, depending on the type. New data types are additional examples of a category that was built into the language. However, type checking was not built into FORTH in any way. When we implement type checking, we add a semantic mechanism to the language that did not previously exist. This is true semantic extension.

Exhibit 4.22. Definition in FORTH of the semantics for arrays.

```
0   :  2by3array    ( The ":" marks the beginning of a definition.  )
1     create        ( Compile time actions for type declarator 2by3array.  )
2       2 , 3 ,     ( Store dimensions in the dictionary with the object.  )
3       12 allot    ( Allocate 12 bytes for 6 short integers.  )
4     does>         ( Run time actions to do a subscripted fetch.  )
5       rangecheck  ( Function call to check that both subscripts are )
6                   ( within the legal range.  )
7       linearsub   ( Function call to compute the effective memory )
8                   ( address, given base address of array and subscripts.)
9   ;               ( End of data type definition.  )
10
11  2by3array box   ( Declare and allocate an array variable named box.  )
12  10 1 2 box !    ( Store the number 10 in box[1,2].  )
```

Program Notes

- Comments are enclosed in parentheses.
- The definition of the new type declarator goes from line 0 to line 9.
- "," stores the prior number in the dictionary.
- Lines 5 and 7 are calls on the functions `rangecheck` and `linearsub`, which the programmer must define and compile before this can be compiled. `Linearsub` must leave its result, the desired memory address, on the stack.
- Line 11 declares a 2by3array variable named `box`. When this line is compiled, the code on lines 2 and 3 is run to allocate and initialize storage for the new array variable.
- Line 12 puts the value 10 on the stack, then the subscripts 1 and 2. When the interpreter processes the reference to `box`, the semantic routine for 2by3array (lines 5–8) is executed. This checks that the subscripts are within range, then computes a memory address and leaves it on the stack.
- Finally, that address is used by "!" to store the 10 that was put on the stack earlier. "!" is the assignment operation. It expects a value and an address to be on the stack and stores the value in that address.

Adding a new control structure. CREATE and DOES> provide semantic extension without corresponding syntactic extension. They permit us to extend the data structuring capabilities of the language but not to add things like new loops that would require modifying the syntax. To the extent that the FORTH compiler's code is open and documented, though, the clever programmer can even extend the syntax in a limited way. We have code that adds a BREAK instruction to exit from the middle of a FIG FORTH loop. This code uses a compiler variable that contains the address of the end of the loop during the process of compiling the loop. The code for BREAK cannot be added to FORTH 83. Many compiler variables that were documented in FIG FORTH are kept secret in the newer FORTH 83. These machine- and implementation-dependent things were taken out of the language documentation in order to increase the portability of programs written in FORTH, and the portability of the FORTH translator itself. Providing no documentation about the internal operation of the compiler prevents the syntax from being extended.

Exercises

1. Briefly define EBNF and syntax diagrams. How are they used, and why are they necessary?

2. Describe the compilation process from source code to object code.

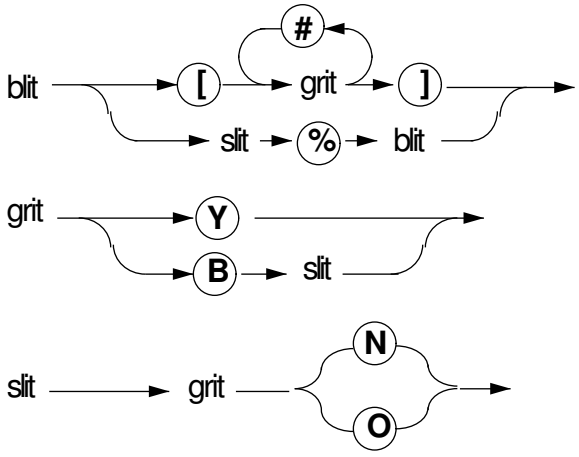
3. Consider the following EBNF syntax. Rewrite this grammar as syntax diagrams.

```
sneech ::=   '*' |
             ( '(' ⟨sneech⟩ ')' ) |
             [  ⟨bander⟩ ] '*' ⟨sneech⟩
bander ::=   { '+$+' | '#' } | ( '%' ⟨bander⟩ )
```

4. Which of the following "sentences" are not legal according to the syntax for sneeches, given in question 3? Why?

a.	(*)	f.	#######*
b.	(+$+*)	g.	(+$+#
c.	*	h.	+$+#*
d.	*****	i.	*+$+#
e.	%%%**	j.	%#*+$+**

5. Rewrite the following syntax diagrams as an EBNF grammar.



6. What is the difference between a terminal and nonterminal symbol in EBNF?

7. What is a production? How are alternatives denoted in EBNF? Repetitions?

8. Using the production rules in Exhibit 4.4, generate the program called "easy" which has two variables: a, an integer, and b, a real. The program initializes a to 5. Then b gets the result of multiplying a by 2. Finally, the value of b is written to the screen followed by a new line.

9. What are the EBNF productions for the conditional statement in Pascal? Show the corresponding syntax diagrams from a standard Pascal reference.

10. Show the syntax diagram for the `For` statement in Pascal. List several details of the meaning of the `For` that are not defined by the syntax diagram.

11. What are semantics?

12. What is the difference between a program environment and a shared environment?

13. What is a stream?

14. Why is lambda calculus relevant in a study of programming language?

15. Show the result of substituting u for x in the following applications. Rename bound variables where necessary.

    a. $((\lambda x.\lambda y.x)u)$

    b. $((\lambda x.\lambda y.z)u)$

    c. $((\lambda x.\lambda u.ux)u)$

    d. $((\lambda x.\lambda x.ux)u)$

16. Each item below is a lambda application. We have used a lot of parentheses to help you parse the expressions. Reduce each formula, until no redex remains. One of the items requires renaming of a bound variable.

    a. $((\lambda x.\lambda y.x(xy))(pq)q)$

    b. $((\lambda x.\lambda y.y)(pq)q)$

    c. $((\lambda z.(\lambda y.yz))(\lambda x.xy))$

    d. $((\lambda x.\lambda y.y(xy))(\lambda p.pp)q)$

17. Verify the following equality. Start with the left-hand side and substitute the formula for *twice*. Then reduce the formula until it is in normal form. This may look like a circular reduction, but the formula reaches normal form after eight reduction steps.

    Let $twice = \lambda f.\lambda x.f(fx)$.
    Show that $twice\ \ twice\ \ gz = g(g(g(gz)))$.

    Hints: Write out the formula for *twice* only when you are using it as a function; keep arguments in symbolic form. Each time you write out *twice*, use new names for the bound variables. Be careful of the parentheses. Remember that function application associates to the left.

18. Show that 3 is the successor of 2, using the lambda calculus representations defined for integers.

19. Define the function "plus2" using a lambda formula. Demonstrate that your formula works by applying it to the formula for 1.

20. Construct a lambda formula to express the following conditional expression. (Assume that x is a Boolean value, T or F.) Verify the correctness of your expression by applying it to T and F and reducing to get 0 or 2.

    If x is true then return 0 else return 2.

21. How do EL/1 and FORTH allow the semantics of the languages to be extended?