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.

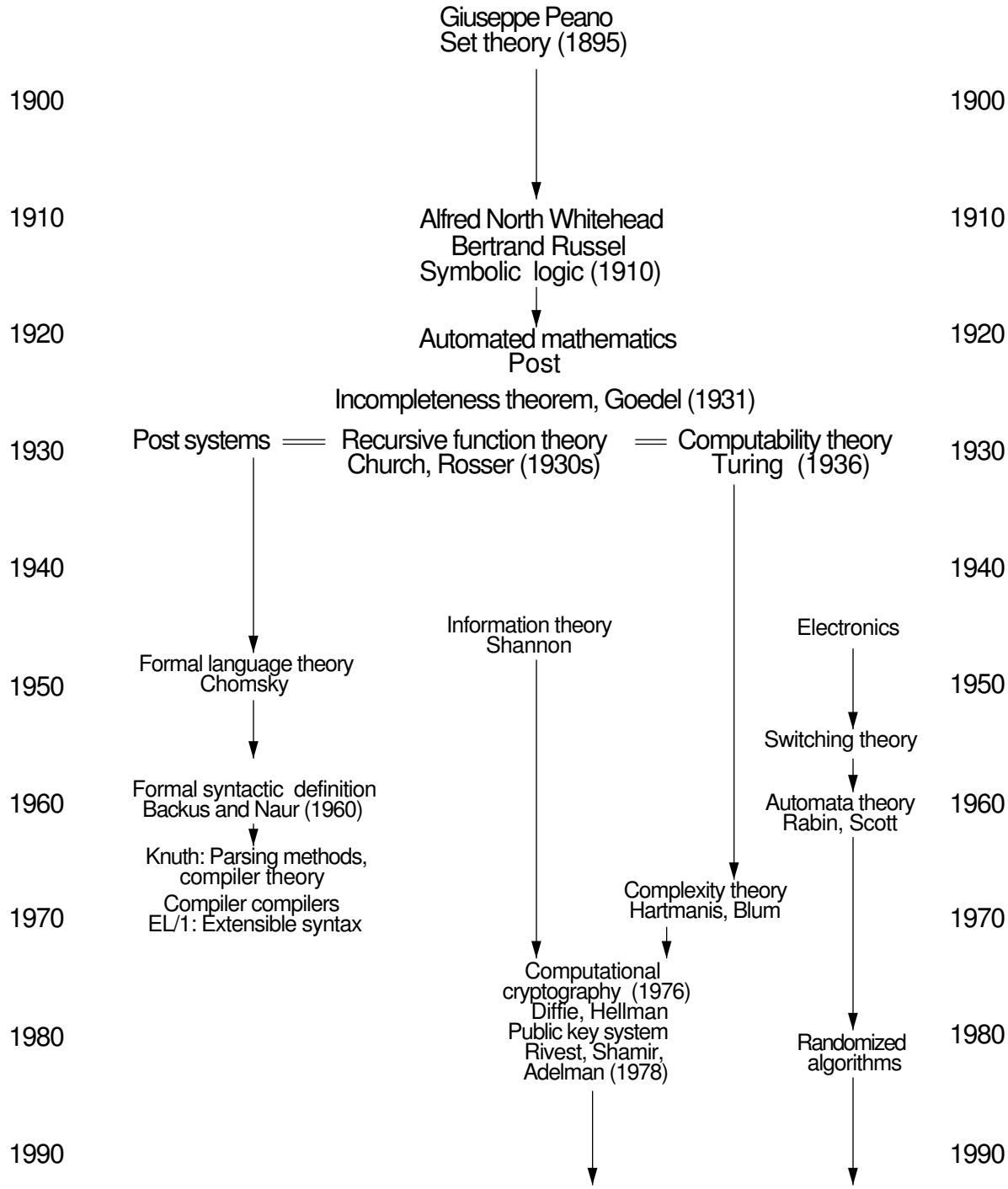
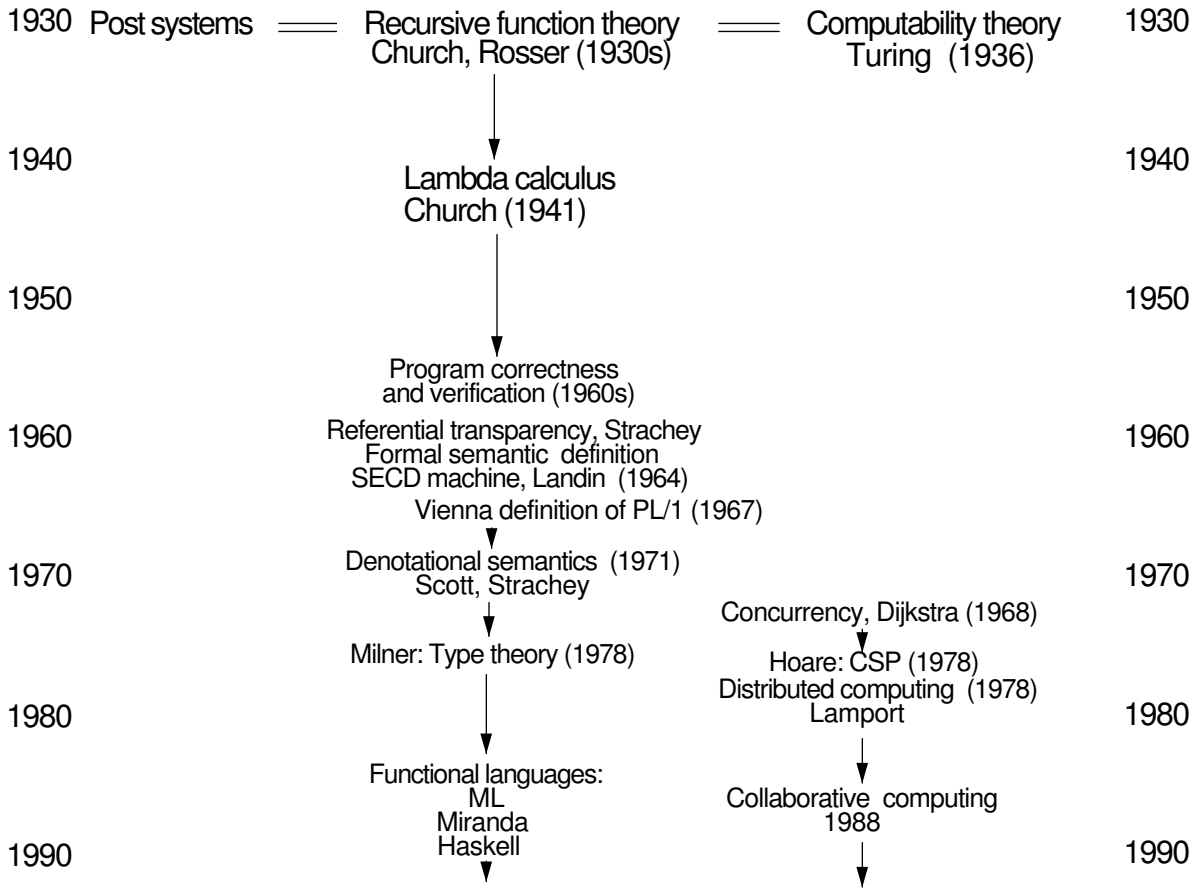


Exhibit 4.2. Formal semantic specification.



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".

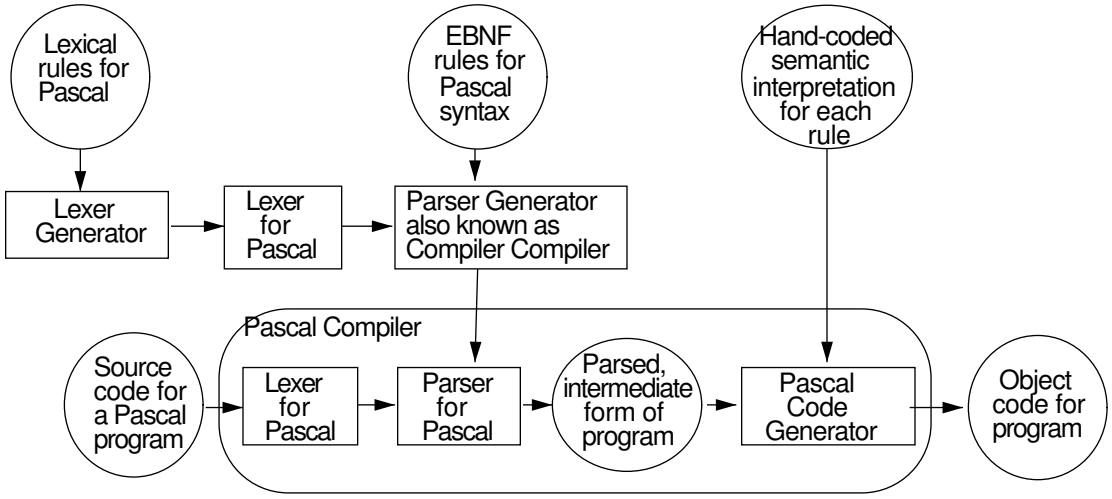
[2]A parser generator can only handle grammars for "context-free" languages. Defining this language class is beyond the scope of this book. Note, though, that the grammars published for most programming languages are context free.

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.

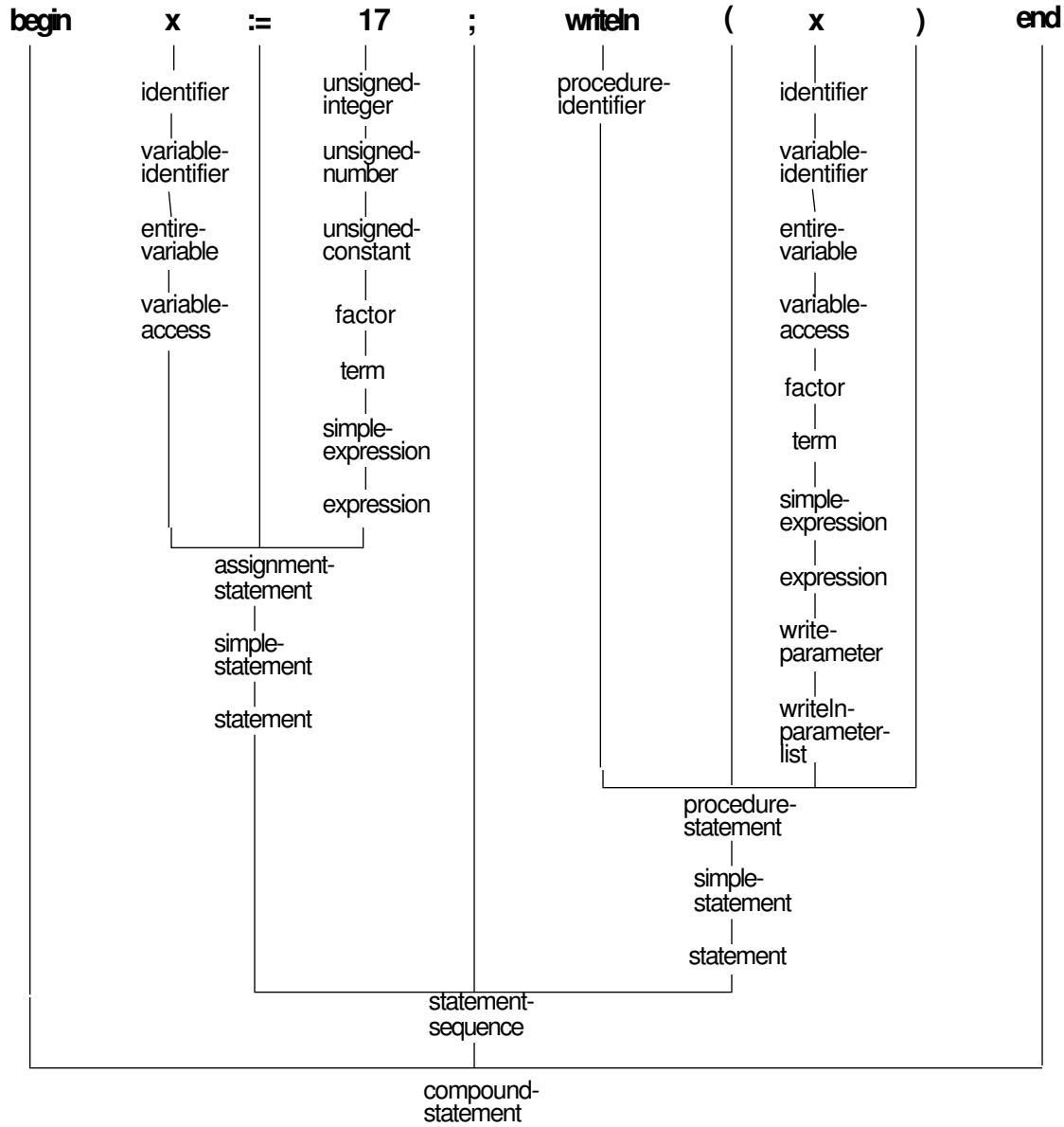


Exhibit 4.6. Syntax diagram for "program".

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

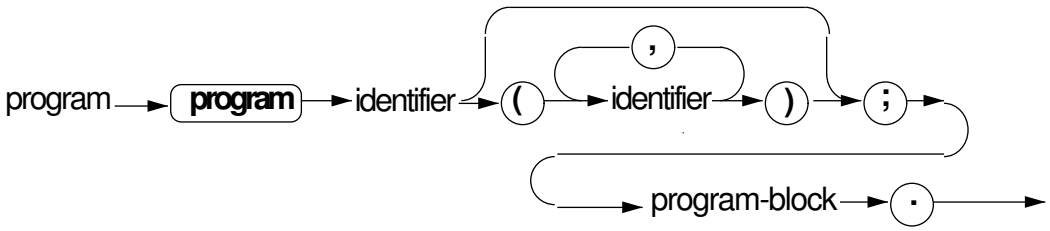


Exhibit 4.7. Syntax diagrams for "statement".

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

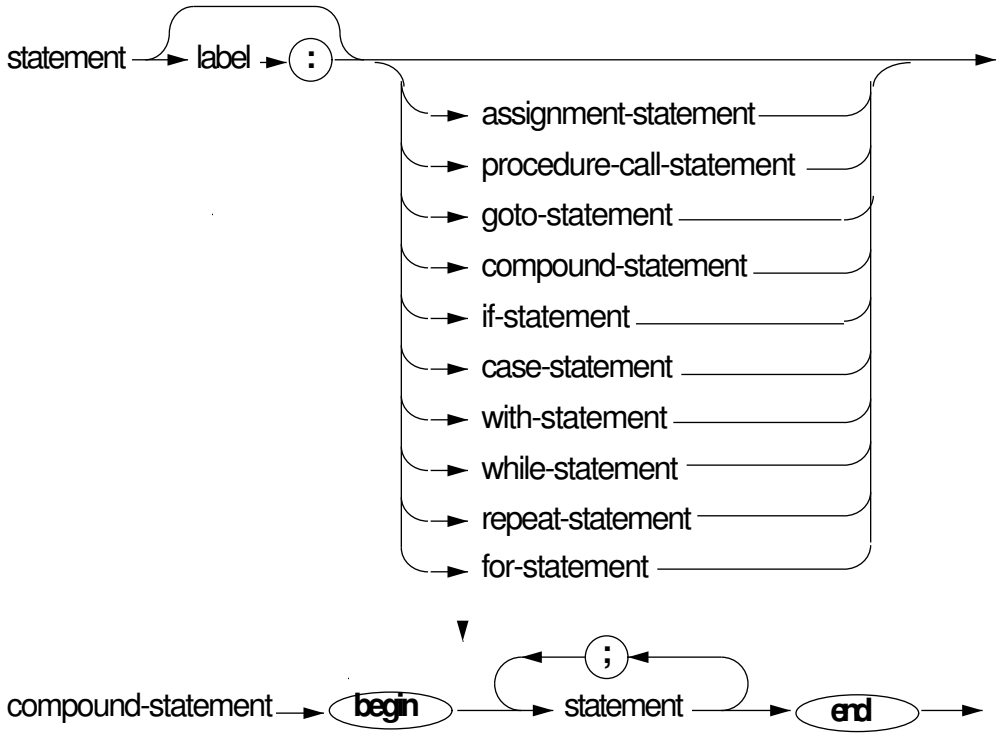


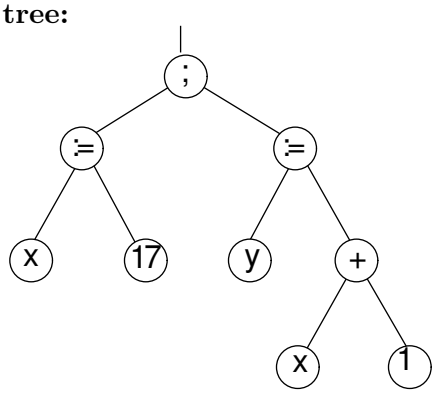
Exhibit 4.8. Translation: from source to machine code.

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.

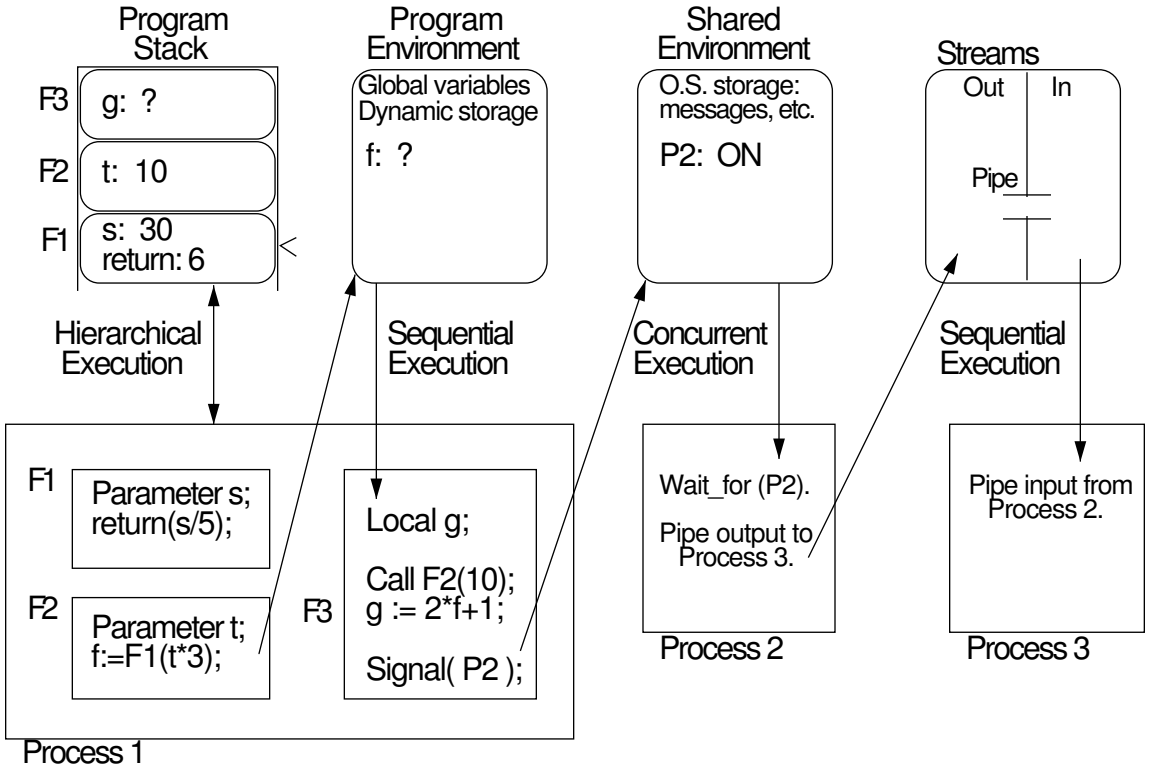
Control. The control section of the abstract model implements the semantic rules of the language that define the order in which the pieces of the abstract computation tree will be evaluated. It defines how execution of statements and functions is to begin, proceed, and end, including the details of sequencing, 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.

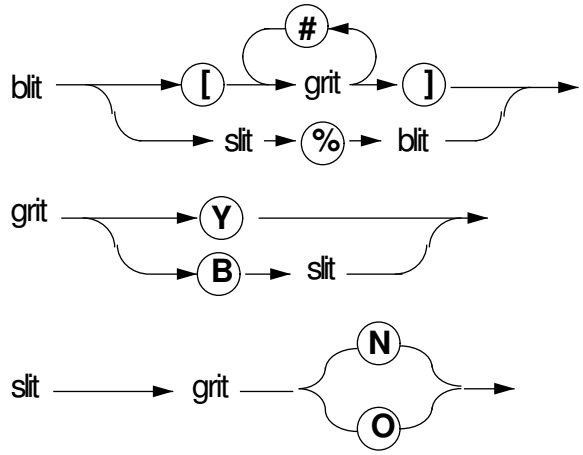
```
sneech ::=   '*' |
             ( '(' ⟨sneech⟩ ')' ) |
             [  ⟨bander⟩ ] '*' ⟨sneech⟩

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?