1.2 GOALS

In this book we are concerned with the basic concepts that appear in modern programming languages, their interaction, and the relationship between programming languages and methods for program development. A recurring theme is the trade-off between language expressiveness and simplicity of implementation. For each programming language feature we consider, we examine the ways that it can be used in programming and the kinds of implementation techniques that may be used to compile and execute it efficiently.

1.2.1 General Goals

In this book we have the following general goals:

- To understand the *design space* of programming languages. This includes concepts and constructs from past programming languages as well as those that may be used more widely in the future. We also try to understand some of the major conflicts and trade-offs between language features, including implementation costs.
- To develop a better understanding of the languages we currently use by comparing them with other languages.
- To understand the programming techniques associated with various language features. The study of programming languages is, in part, the study of conceptual frameworks for problem solving, software construction, and development.

Many of the ideas in this book are common knowledge among professional programmers. The material and ways of thinking presented in this book should be useful to you in future programming and in talking to experienced programmers if you work for a software company or have an interview for a job. By the end of the course, you will be able to evaluate language features, their costs, and how they fit together.

1.2.2 Specific Themes

Here are some specific themes that are addressed repeatedly in the text:

- *Computability:* Some problems cannot be solved by computer. The undecidability of the halting problem implies that programming language compilers and interpreters cannot do everything that we might wish they could do.
- *Static analysis:* There is a difference between compile time and run time. At compile time, the program is known but the input is not. At run time, the program and the input are both available to the run-time system. Although a program designer or implementer would like to find errors at compile time, many will not surface until run time. Methods that detect program errors at compile time are usually conservative, which means that when they say a program does not have a certain kind of error this statement is correct. However, compile-time error-detection methods will usually say that some programs contain errors even if errors may not actually occur when the program is run.
- *Expressiveness versus efficiency:* There are many situations in which it would be convenient to have a programming language implementation do something automatically. An example discussed in Chapter 3 is memory management: The Lisp run-time system uses garbage collection to detect memory locations no longer needed by the program. When something is done automatically, there is a cost. Although an automatic method may save the programmer from thinking about something, the implementation of the language may run more slowly. In some cases, the automatic method may make it easier to write programs and make programming less prone to error. In other cases, the

Chapter 2: Computability

Some mathematical functions are computable and some are not. In all general-purpose programming languages, it is possible to write a program for each function that is computable in principle. However, the limits of computability also limit the kinds of things that programming language implementations can do. This chapter contains a brief overview of computability so that we can discuss limitations that involve computability in other chapters of the book.

2.1 PARTIAL FUNCTIONS AND COMPUTABILITY

From a mathematical point of view, a program defines a function. The output of a program is computed as a function of the program inputs and the state of the machine before the program starts. In practice, there is a lot more to a program than the function it computes. However, as a starting point in the study of programming languages, it is useful to understand some basic facts about computable functions.

The fact that not all functions are computable has important ramifications for programming language tools and implementations. Some kinds of programming constructs, however useful they might be, cannot be added to real programming languages because they cannot be implemented on real computers.

2.1.1 Expressions, Errors, and Nontermination

In mathematics, an expression may have a defined value or it may not. For example, the expression 3 + 2 has a defined value, but the expression 3/0 does not. The reason that 3/0 does not have a value is that division by zero is not defined: division is defined to be the inverse of multiplication, but multiplication by zero cannot be inverted. There is nothing to try to do when we see the expression 3/0; a mathematician would just say that this operation is undefined, and that would be the end of the discussion.

In computation, there are two different reasons why an expression might not have a value:

Alan Turing was a British mathematician. He is known for his early work on computability and his work for British Intelligence on code breaking during the Second World War. Among computer scientists, he is best known for the invention of the Turing machine. This is not a piece of hardware, but an idealized computing device. A Turing machine consists of an infinite tape, a tape read-write head, and a finite-state controller. In each computation step, the machine reads a tape symbol and the finite-state controller decides whether to write a different symbol on the current tape square and then whether to move the read-write head one square left or right. The importance of this idealized computer is that it is both very simple and very powerful.

Turing was a broad-minded individual with interests ranging from relativity theory and mathematical logic to number theory and the engineering design of mechanical computers. There are numerous published biographies of Alan Turing, some emphasizing his wartime work and others calling attention to his sexuality and its impact on his professional career.

The ACM Turing Award is the highest scientific honor in computer science, equivalent to a Nobel Prize in other fields.

- *Error termination:* Evaluation of the expression cannot proceed because of a conflict between operator and operand.
- *Nontermination:* Evaluation of the expression proceeds indefinitely.

An example of the first kind is division by zero. There is nothing to compute in this case, except possibly to stop the computation in a way that indicates that it could not proceed any further. This may halt execution of the entire program, abort one thread of a concurrent program, or raise an exception if the programming language provides exceptions.

The second case is different: There is a specific computation to perform, but the computation may not terminate and therefore may not yield a value. For example, consider the recursive function defined by

```
f(x:int) = if x = 0 then 0 else x + f(x-2)
```

This is a perfectly meaningful definition of a *partial* function, a function that has a value on some arguments but not on all arguments. The expression f(4) calling the function f above has value 4 + 2 + 0 = 6, but the expression f(5) does not have a value because the computation specified by this expression does not terminate.

2.1.2 Partial Functions

A partial function is a function that is defined on some arguments and undefined on others. This is ordinarily what is meant by function in programming, as a function declared in a program may return a result or may not if some loop or sequence of recursive calls does not terminate. However, this is not what a mathematician ordinarily means by the word function.

The distinction can be made clearer by a look at the mathematical definitions. A reasonable definition of the word function is this: A function $f : A \to B$ from set A to set B is a rule associating a unique value $y = f(x)$in B with every x in A. This is almost a mathematical definition, except that the word rule does not have a precise mathematical meaning. The notation $f : A \to B$ means that, given arguments in the set A, the function f produces values from set B. The set A is called the *domain* of *f*, and the set B is called the *range* or the *codomain* of *f*.

The usual mathematical definition of function replaces the idea of rule with a set of argument-result pairs called the graph of a function. This is the mathematical definition:

A *function* $f : A \to B$ is a set of ordered pairs $f \subseteq A \times B$ that satisfies the following conditions:

1. If ?*x*, *y*? ? *f* and ?*x*, *z*?? *f*, then *y* = *z*.

2. For every *x* ? *A*, there exists *y* ? *B* with ?*x*, *y*?? *f*.

When we associate a set of ordered pairs with a function, the ordered pair ?*x*, *y*? is used to indicate that *y* is the value of the function on argument *x*. In words, the preceding two conditions can be stated as (1) a function has at most one value for every argument in its domain, and (2) a function has at least one value for every argument in its domain.

A partial function is similar, except that a partial function may not have a value for every argument in its domain. This is the mathematical definition:

A *partial function* $f : A \to B$ is a set of ordered pairs $f \subseteq A \times B$ satisfying the preceding condition

1. If ?*x*, *y*?? *f* and ?*x*, *z*?? *f*, then *y* = *z*.

In words, a partial function is single valued, but need not be defined on all elements of its domain.

Programs Define Partial Functions

In most programming languages, it is possible to define functions recursively. For example, here is a function f defined in terms of itself:

```
f(x:int) = ifx=0 then 0 else x + f(x-2);
```

If this were written as a program in some programming language, the declaration would associate the function name f with an algorithm that terminates on every evenx≥0, but diverges (does not halt and return a value) if x is odd or negative. The algorithm for f defines the following mathematical function *f*, expressed here as a set of ordered pairs:

f ={?x, y?| x is positive and even, y = 0 + 2 + 4 +...+ x}.

This is a partial function on the integers. For every integer *x*, there is at most one *y* with *f* (*x*) = *y*. However, if *x* is an odd number, then there is no *y* with *f* (*x*) = *y*. Where the algorithm does not terminate, the value of the function is undefined. Because a function call may not terminate, this program defines a partial function.

2.1.3 Computability

Computability theory gives us a precise characterization of the functions that are computable in principle. The class of functions on the natural numbers that are computable in principle is often called the class of *partial recursive functions*, as recursion is an essential part of computation and computable functions are, in general, partial rather than total. The reason why we say "computable in principle" instead of "computable in practice" is that some computable functions might take an extremely long time to compute. If a function call will not return for an amount of time equal to the length of the entire history of the universe, then in practice we will not be able to wait for the computation to finish. Nonetheless, computability in principle is an important benchmark for programming languages.

Computable Functions

Intuitively, a function is *computable* if there is some program that computes it. More specifically, a function $f : A \rightarrow B$ is computable if there is an algorithm that, given any x ? A as input, halts with $y = f(x)$ as output.

One problem with this intuitive definition of computable is that a program has to be written out in some programming language, and we need to have some implementation to execute the program. It might very well be that, in one programming language, there is a program to compute some mathematical function and in another language there is not.

In the 1930s, Alonzo Church of Princeton University proposed an important principle, called Church's thesis. Church's thesis, which is a widely held belief about the relation between mathematical definitions and the real world of computing, states that the same class of functions on the integers can be computed by any general computing device. This is the class of partial recursive functions, sometimes called the class of *computable functions*. There is a mathematical definition of this class of functions that does not refer to programming languages, a second definition that uses a kind of idealized computing device called a *Turing machine*, and a third (equivalent) definition that uses lambda calculus (see Section 4.2). As mentioned in the biographical sketch on Alan Turing, a Turing machine consists of an infinite tape, a tape read-write head, and a finite-state controller. The tape is divided into contiguous cells, each containing a single symbol. In each computation step, the machine reads a tape symbol and the finite-state controller decides whether to write a different symbol on the current tape square and then whether to move the read-write head one square left or right. Part of the evidence that Church cited in formulating this thesis was the proof that Turing machines and lambda calculus are equivalent. The fact that all standard programming languages express precisely the class of partial recursive functions is often summarized by the statement that *all programming languages are Turing complete*. Although it is comforting to know that all programming languages are universal in a mathematical sense, the fact that all programming languages are Turing complete also means that computability theory does not help us distinguish among the expressive powers of different programming languages.

Noncomputable Functions

It is useful to know that some specific functions are not computable. An important example is commonly referred to as the *halting problem*. To simplify the discussion and focus on the central ideas, the halting problem is stated for programs that require one string input. If P is such a program and x is a string input, then we write $P(x)$for the output of program P on input x.

> *Halting Problem:* Given a program P that requires exactly one string input and a string x, determine whether P halts on input x.

We can associate the halting problem with a function f_{halt} by letting f_{halt} (P, x) = "halts" if P halts on input and $f_{halt}(P, x)$ = "does not halt" otherwise. This function f_{halt} can be considered a function on strings if we write each program out as a sequence of symbols.

The *undecidability of the halting problem* is the fact that the function f_{halt} is not computable. The undecidability of the halting problem is an important fact to keep in mind in designing programming language implementations and optimizations. It implies that many useful operations on programs cannot be implemented, even in principle.

Proof of the Undecidability of the Halting Problem. Although you will not need to know this proof to understand any other topic in the book, some of you may be interested in proof that the halting function is not computable. The proof is surprisingly short, but can be difficult to understand. If you are going to be a serious computer scientist, then you will want to look at this proof several times, over the course of several days, until you understand the idea behind it.

> Step 1: Assume that there is a program Q that solves the halting problem. Specifically, assume that program Q reads two inputs, both strings, and has the following output:

$$Q(P, x) = \begin{cases} \text{halts} & \text{if } P(x) \text{ halts} \end{cases}$$

$$\text{does not halt} \quad \text{if } P(x) \text{ does not}$$

An important part of this specification for *Q* is that *Q*(*P*, *x*) always halts for every *P* and *x*.

Step 2: Using program *Q*, we can build a program *D* that reads one string input and sometimes does not halt. Specifically, let *D* be a program that works as follows:

D(*P*) = if *Q*(*P*, *P*) = halts then *run forever* else *halt*.

Note that *D* has only one input, which it gives twice to *Q*. The program *D* can be written in any reasonable language, as any reasonable language should have some way of programming if-then-else and some way of writing a loop or recursive function call that runs forever. If you think about it a little bit, you can see that *D* has the following behavior:

$$D(P) = \begin{cases} \text{halt} & \text{if } P(P) \text{ runs forever} \\ \text{run forever} & \text{if } P(P) \text{ halts} \end{cases}.$$

In this description, the word halt means that *D*(*P*) comes to a halt, and runs forever means that *D*(*P*) continues to execute steps indefinitely. The program *D*(*P*) halts or does not halt, but does not produce a string output in any case.

Step 3: Derive a contradiction by considering the behavior *D*(*D*) of program *D* on input *D*. (If you are starting to get confused about what it means to run a program with the program itself as input, assume that we have written the program *D* and stored it in a file. Then we can compile *D* and run *D* with the file containing a copy of *D* as input.) Without thinking about how *D* works or what *D* is supposed to do, it is clear that either *D*(*D*) halts or *D*(*D*) does not halt. If *D*(*D*) halts, though, then by the property of *D* given in step 2, this must be because *D*(*D*) runs forever. This does not make any sense, so it must be that *D*(*D*) runs forever. However, by similar reasoning, if *D*(*D*) runs forever, then this must be because *D*(*D*) halts. This is also contradictory. Therefore, we have reached a contradiction.

Step 4: Because the assumption in step 1 that there is a program *Q* solving the halting problem leads to a contradiction in step 3, it must be that the assumption is false. Therefore, there is no program that solves the halting problem.

Applications

Programming language compilers can often detect errors in programs. However, the undecidability of the halting problem implies that some properties of programs cannot be determined in advance. The simplest example is halting itself. Suppose someone writes a program like this:

```
i=0;
while (i != f(i)) i = g(i);
printf( ... i ...);
```

It seems very likely that the programmer wants the while loop to halt. Otherwise, why would the programmer have written a statement to print the value of i after the loop halts? Therefore, it would be helpful for the compiler to print a warning message if the loop will not halt. However useful this might be, though, it is not possible for a compiler to determine whether the loop will halt, as this would involve solving the halting problem.

Previous Next

2.2 CHAPTER SUMMARY

Computability theory establishes some important ground rules for programming language design and implementation. The following main concepts from this short overview should be remembered:

- *Partiality:* Recursively defined functions may be partial functions. They are not always total functions. A function may be partial because a basic operation is not defined on some argument or because a computation does not terminate.
- *Computability:* Some functions are computable and others are not. Programming languages can be used to define computable functions; we cannot write programs for functions that are not computable in principle.
- *Turing completeness:* All standard general-purpose programming languages give us the same class of computable functions.
- *Undecidability:* Many important properties of programs cannot be determined by any computable function. In particular, the halting problem is undecidable.

When the value of a function or the value of an expression is undefined because a basic operation such as division by zero does not make sense, a compiler or interpreter can cause the program to halt and report the error. However, the undecidability of the halting problem implies that there is no way to detect and report an error whenever a program is not going to halt.

There is a lot more to computability and complexity theory than is summarized in the few pages here. For more information, see one of the many books on computability and complexity theory such as *Introduction to Automata Theory, Languages, and Computation* by Hopcroft, Motwani, and Ullman (Addison Wesley, 2001) or *Introduction to the Theory of Computation* by Sipser (PWS, 1997).

Previous Next

Chapter 4: Fundamentals

OVERVIEW

In this chapter some background is provided on programming language implementation through brief discussions of syntax, parsing, and the steps used in conventional compilers. We also look at two foundational frameworks that are useful in programming language analysis and design: lambda calculus and denotational semantics. Lambda calculus is a good framework for defining syntactic concepts common to many programming languages and for studying symbolic evaluation. Denotational semantics shows that, in principle, programs can be reduced to functions.

A number of other theoretical frameworks are useful in the design and analysis of programming languages. These range from computability theory, which provides some insight into the power and limitations of programs, to type theory, which includes aspects of both syntax and semantics of programming languages. In spite of many years of theoretical research, the current programming language theory still does not provide answers to some important foundational questions. For example, we do not have a good mathematical theory that includes higher-order functions, state transformations, and concurrency. Nonetheless, theoretical frameworks have had an impact on the design of programming languages and can be used to identify problem are as in programming languages. To compare one aspect of theory and practice, we compare functional and imperative languages in Section 4.4.

4.1 COMPILERS AND SYNTAX

A program is a description of a dynamic process. The text of a program itself is called its syntax; the things a program does comprise its semantics. The function of a programming language implementation is to transform program syntax into machine instructions that can be executed to cause the correct sequence of actions to occur.

4.1.1 Structure of a Simple Compiler

Programming languages that are convenient for people to use are built around concepts and abstractions that may not correspond directly to features of the underlying machine. For this reason, a program must be translated into the basic instruction set of the machine before it can be executed. This can be done by a *compiler*, which translates the entire program into machine code before the program is run, or an *interpreter*, which combines translation and program execution. We discuss programming language implementation by using compilers, as this makes it easier to separate the main issues and to discuss them in order.

An early pioneer, John Backus became a computer programmer at IBM in 1950. In the 1950s, Backus developed Fortran, the first high-level computer language, which became commercially available in 1957. The language is still widely used for numerical and scientific programming. In 1959, John Backus invented Backus naur form (BNF), the standard notation for defining the syntax of a programming language. In later years, he became an advocate of pure functional programming, devoting his 1977 ACM Turing Award lecture to this topic.

Tools by Aho, Sethi, and Ullman (Addison-Wesley, 1986),and *Modern Compiler Implementation in Java/ML/C* by Appel (Cambridge Univ. Press, 1998).

Lexical Analysis

The input symbols are scanned and grouped into meaningful units called *tokens*. For example, lexical analysis of the expression temp := x+1, which uses Algol-style notation := for assignment, would divide this sequence of symbols into five tokens: the identifier temp, the assignment "symbol" :=, the variable x, the addition symbol+, and the number 1. Lexical analysis can distinguish numbers from identifiers. However, because lexical analysis is based on a single left-to-right (and top-to-bottom)scan, lexical analysis does not distinguish between identifiers that are names of variables and identifiers that are names of constants. Because variables and constants are declared differently, variables and constants are distinguished in the semantic analysis phase.

Syntax Analysis

In this phase, tokens are grouped into syntactic units such as expressions, statements, and declarations that must conform to the grammatical rules of the programming language. The action performed during this phase, called *parsing*, is described in Subsection 4.1.2. The purpose of parsing is to produce a data structure called a parse tree, which represents the syntactic structure of the program in a way that is convenient for subsequent phases of the compiler. If a program does not meet the syntactic requirements to be a well formed program, then the parsing phase will produce an error message and terminate the compiler.

Semantic Analysis

In this phase of a compiler, rules and procedures that depend on the context surrounding an expression are applied. For example, returning to our sample expression temp := x+1, we find that it is necessary to make sure that the types match. If this assignment occurs in a language in which integers are automatically converted to floats as needed, then there are several ways that types could be associated with parts of this expression. In standard semantic analysis, the types of temp and x would be determined from the declarations of these identifiers. If these are both integers, then the number 1 could be marked as an integer and + marked as integer addition, and the expression would be considered correct. If one of the identifiers, say x, is a float, then the number 1 would be marked as a float and + marked as a floating-point addition. Depending on whether temp is a float or an integer, it might also be necessary to insert a conversion around the subexpression x+1. The output of this phase is an augmented parse tree that represents the syntactic structure of the program and includes additional information such as the types of identifiers and the place in the program where each identifier is declared.

Although the phase following parsing is commonly called *semantic analysis*, this use of the word *semantic* is different from the standard use of the term for *meaning*. Some compiler writers use the word semantic because this phase relies on context information, and the kind of grammar used for syntactic analysis does not capture context information. However, in the rest of this book, the word semantics is used to refer to how a program executes, not the essentially syntactic properties that arise in the third phase of a compiler.

Intermediate Code Generation

Although it might be possible to generate a target program from the results of syntactic and semantic analysis, it is difficult to generate efficient code in one phase. Therefore, many compilers first produce an intermediate form of code and then optimize this code to produce a more efficient target program.

Because the last phase of the compiler can translate one set of instructions to another, the intermediate code does not need to be written with the actual instruction set of the target machine. It is important to use an intermediate representation that is easy to produce and easy to translate into the target language. The intermediate representation can be some form of generic low-level code that has properties common to several computers. When a single generic intermediate representation is used, it is possible to use essentially the same compiler to generate target programs for several different machines.

Code Optimization

There are a variety of techniques that may be used to improve the efficiency of a program. These techniques are usually applied to the intermediate representation. If several optimization techniques are written as transformations of the intermediate representation, then these techniques can be applied over and over until some termination condition is reached.

The following list describes some standard optimizations:

- *Common Subexpression Elimination:* If a program calculates the same value more than once and the compiler can detect this, then it may be possible to transform the program so that the value is calculated only once and stored for subsequent use.
- *Copy Propagation:* If a program contains an assignment such as x=y, then it may be possible to change subsequent statements to refer to y instead of to x and to eliminate the assignment.
- *Dead-Code Elimination:* If some sequence of instructions can never be reached, then it can be eliminated from the program.
- *Loop Optimizations:* There are several techniques that can be applied to remove instructions from loops. For example, if some expression appears inside a loop but has the same value on each pass through the loop, then the expression can be moved outside the loop.
- *In-Lining Function Calls:* If a program calls function f, it is possible to substitute the code for f into the place where f is called. This makes the target program more efficient, as the instructions associated with calling a function can be eliminated, but it also increases the size of the program. The most important consequence of in-lining function calls is usually that they allow other optimizations to be performed by removing jumps from the code.

Code Generation

The final phase of a standard compiler is to convert the intermediate code into a target machine code. This involves choosing a memory location, a register, or both, for each variable that appears in the program. There are a variety of register allocation algorithms that try to reuse registers efficiently. This is important because many machines have a fixed number of registers, and operations on registers are more efficient than transferring data into and out of memory.

4.1.2 Grammars and Parse Trees

We use grammars to describe various languages in this book. Although we usually are not too concerned about the pragmatics of parsing, we take a brief look in this subsection at the problem of producing a parse tree from a sequence of tokens.

Grammars

Grammars provide a convenient method for defining infinite sets of expressions. In addition, the structure imposed by a grammar gives us a systematic way of processing expressions.

A *grammar* consists of a start symbol, a set of nonterminals, a set of terminals, and a set of productions. The nonterminals are symbols that are used to write out the grammar, and the terminals are symbols that appear in the language generated by the grammar. In books on automata theory and related subjects, the productions of a grammar are written in the form $s \rightarrow tu$, with an arrow, meaning that in a string containing the symbol *s*, we can replace *s* with the symbols *tu*. However, here we use a more compact notation, commonly referred to as BNF.

The main ideas are illustrated by example. A simple language of numeric expressions is defined by the following grammar:

```
e ::= n | e+e | e-e
n ::= d | nd
d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

where e is the *start symbol*, symbols e, n, and d are nonterminals, and 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, and - are the terminals. The language defined by this grammar consists of all the sequences of terminals that we can produce by starting with the start symbol e and by replacing nonterminals according to the preceding productions. For example, the first preceding production means that we can replace an occurrence of e with the symbol n, the three symbols e+e, or the three symbols e-e. The process can be repeated with any of the preceding three lines.

Some expressions in the language given by this grammar are

0, 1+3+5, 2+4 - 6 - 8

Sequences of symbols that contain nonterminals, such as

e, e+e, e+6 - e

are not expressions in the language given by the grammar. The purpose of non-terminals is to keep track of the form of an expression as it is being formed. All nonterminals must be replaced with terminals to produce a well-formed expression of the language.

Derivations

A sequence of replacement steps resulting in a string of terminals is called a *derivation*.

Here are two derivations in this grammar, the first given in full and the second with a few missing steps that can be filled in by the reader (be sure you understand how!):

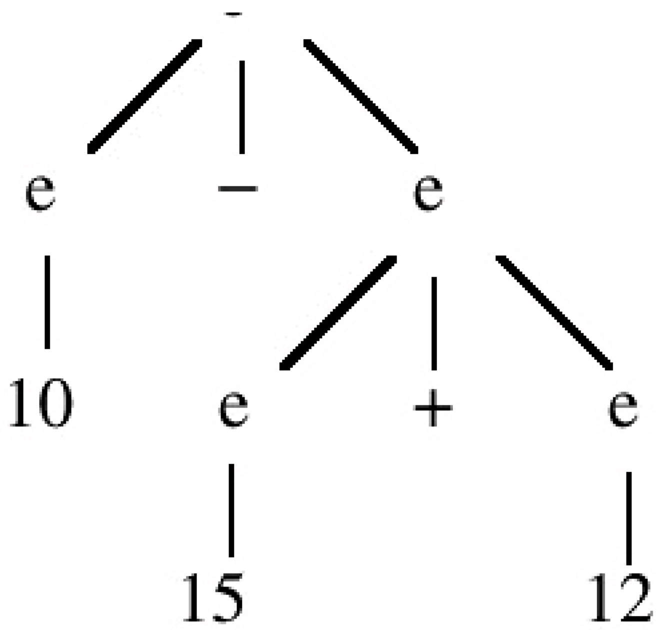
e ⟶ n ⟶ nd ⟶ dd ⟶ 2d ⟶ 25

e ⟶ e - e ⟶ e - e+e ⟶ … ⟶ n-n+n ⟶ … … 10-15+12

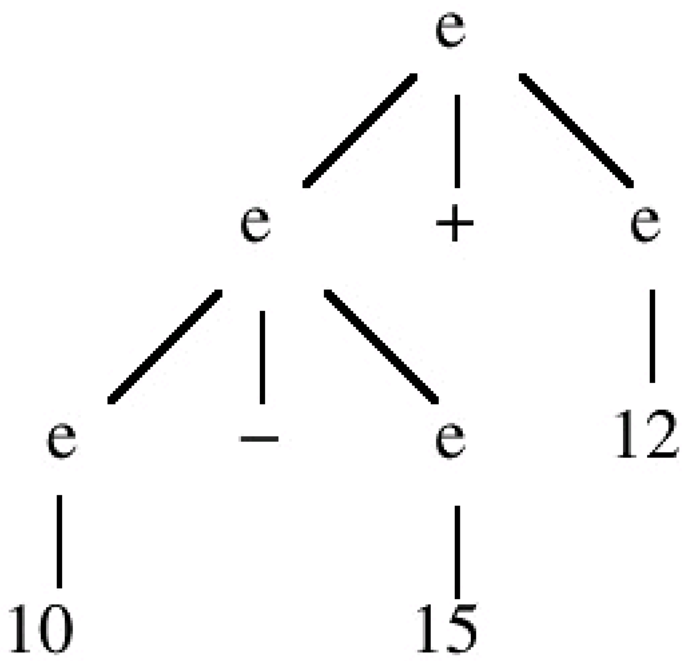
Parse Trees and Ambiguity

It is often convenient to represent a derivation by a tree. This tree, called the *parse tree* of a derivation, or *derivation tree*, is constructed with the start symbol as the root of the tree. If a step in the derivation is to replace *s* with $x_1,\ldots,x_n$, then the children of *s* in the tree will be nodes labeled $x_1,\ldots,x_n$.

The parse tree for the derivation of 10–15+12 in the preceding subsection has some useful structure. Specifically, because the first step yields e-e, the parse tree has the form



where we have contracted the subtrees for each two-digit number to a single node. This tree is different from



which is another parse tree for the same expression. An important fact about parse trees is that each corresponds to a unique parenthesization of the expression. Specifically, the first tree corresponds to 10-(15+12) whereas the second corresponds to (10-15)+12. As this example illustrates, the value of an expression may depend on how it is parsed or parenthesized.

A grammar is *ambiguous* if some expression has more than one parse tree. If every expression has at most one parse tree, the grammar is *unambiguous*.

Example 4.1

There is an interesting ambiguity involving if-then-else. This can be illustrated by the following simple grammar:

s ::= v := e | s;s | if b then s | if b then s else s meanings for statements of this form.

4.1.3 Parsing and Precedence

Parsing is the process of constructing parse trees for sequences of symbols. Suppose we define a language *L* by writing out a grammar *G*. Then, given a sequence of symbols *s*, we would like to determine if *s* is in the language *L*. If so, then we would like to compile or interpret *s*, and for this purpose we would like to find a parse tree for *s*. An algorithm that decides whether *s* is in *L*, and constructs a parse tree if it is, is called a *parsing algorithm* for *G*.

There are many methods for building parsing algorithms from grammars. Many of these work for only particular forms of grammars. Because parsing is an important part of compiling programming languages, parsing is usually covered in courses and text books on compilers. For most programming languages you might consider, it is either straight forward to parse the language or there are some changes in syntax that do not change the structure of the language very much but make it possible to parse the language efficiently.

Two issues we consider briefly are the syntactic conventions of precedence and right or left associativity. These are illustrated briefly in the following example.

Example 4.2

A programming language designer might decide that expressions should include addition, subtraction, and multiplication and write the following grammar:

```
e ::= 0 | 1 | e+e | e-e | e*e
```

This grammar is ambiguous, as many expressions have more than one parse tree. For expressions such as 1–1+1, the value of the expression will depend on the way it is parsed. One solution to this problem is to require complete parenthesization. In other words, we could change the grammar to

```
e ::= 0 | 1 | (e+e) | (e-e) | (e*e)
```

so that it is no longer ambiguous. However, as you know, it can be awkward to write a lot of parentheses. In addition, for many expressions, such as 1+2+3+4, the value of the expression does not depend on the way it is parsed. Therefore, it is unnecessarily cumbersome to require parentheses for every operation.

The standard solution to this problem is to adopt parsing conventions that specify a single parse tree for every expression. These are called *precedence* and *associativity*. For this specific grammar, a natural precedence convention is that multiplication(*) has a higher precedence than addition (+) and subtraction (–). We incorporate precedence into parsing by treating an unparenthesized expression e op1 e op2 e as if parentheses are inserted around the operator of higher precedence. With this rule in effect, the expression 5*4-3 will be parsed as if it were written as (5*4)-3. This coincides with the way that most of us would ordinarily think about the expression5*4-3. Because there is no standard way that most readers would parse 1+1-1, we might give addition and subtraction equal precedence. In this case, a compiler could issue an error message requiring the programmer to parenthesize 1+1-1. Alternatively, an variable f:

$$(\lambda\, f\,.f\,x)\ (\lambda\, y\,.y) = (\lambda y,y)x$$

Of course, $(\lambda\, y.y)x$ may be simplified by an additional substitution, and so we have

$$(\lambda\, f\,.f\,x)(\lambda\, y\,.y) = (\lambda\, y\,.y)x = x$$

Any readers old enough to be familiar with the original documentation of Algol 60 will recognize β-equivalence as the *copy rule* for evaluating function calls. There are also parallels between (β) and macroexpansion and in-line substitution of functions.

Renaming Bound Variables

Because λ bindings in M can conflict with free variables in N, substitution $[N/x]M$ is a little more complicated than we might think at first. However, we can avoid all of the complications by following the variable convention: renaming bound variables in $(\lambda\, x.M)N$ so that all of the bound variables are different from each other and different from all of the free variables. For example, let us reduce the term$(\lambda\, f\,.\lambda\, x.\ f\,(\,fx))(\lambda\, y.y + x)$. If we first rename bound variables and then perform β-reduction, we get

$$(\lambda\, f\,.\lambda\, z\,.f(f\,z))\ (\lambda\, y\,.y + x) = \lambda\, z.((\lambda\, y\,.y + x)((\lambda\, y\,.y + x)z)) = \lambda\, z\,.z + x + x$$

If we were to forget to rename bound variables and substitute blindly, we might simplify as follows:

$$(\lambda\, f\,.\ \lambda\, x\,.\ f(f\,x))(\lambda\, y\,.\ y + x) = \lambda\, x.((\lambda\, y\,.\ y + x)((\lambda\, y\,.\ y + x)x)) = \lambda\, x\,.x + x + x$$

However, we should be suspicious of the second reduction because the variable x is free in $(\lambda\, y.y + x)$ but becomes bound in $\lambda\, x.x + x + x$. *Remember:* In working out$[N/x]M$, we must rename any bound variables in M that might be the same as free variables in N. To be precise about renaming bound variables in substitution, we define the result $[N/x]M$ of substituting N for x in M by induction on the structure of M:

- $[N/x]x = N$,
- $[N/x]y = y$, where y is any variable different from x,
- $[N/x](M_1\,M_2) = ([N/x]M^1)([N/x]M_2)$,
- $[N/x](\lambda\, x.M) = \lambda\, x.M$,
- $[N/x](\lambda\, y.M) = \lambda\, y.([N/x]M)$, where y is not free in N.

Because we are free to rename the bound variable y in $\lambda\, y.M$, the final clause $\lambda\, y.([N/x]M)$ always makes sense. With this precise definition of substitution, we now have a precise definition of β-equivalence.

4.2.3 Programming in Lambda Calculus

Lambda calculus may be viewed as a simple functional programming language. We will see that, even though the language is very simple, we can program fairly naturally if we adopt a few abbreviations. Before declarations and recursion are discussed, it is worth mentioning the problem of multiargument functions.

Functions of Several Arguments

Lambda abstraction lets us treat any expression M as a function of any variable x by writing $\lambda\, x.M$. However, what if we want to treat M as function of two variables, x and y? Do we need a second kind of lambda abstraction $\lambda_2 x,\ y.M$ to treat M as function of the pair of arguments x, y? Although we could add lambda operators for sequences of formal parameters, we do not need to because ordinary lambda abstraction will suffice for most purposes.

We may represent a function f of two arguments by a function $\lambda\, x.(\lambda\, y.M)$of a single argument that, when applied, returns a second function that accepts a second argument and then computes a result in the same way as f. For example, the function

$$f(g,\ x) = g(x)$$

has two arguments, but can be represented in lambda calculus by ordinary lambda abstraction. We define f_{curry} by

$$f_{\text{curry}} = \lambda g.\lambda x.gx$$

The difference between f and f_{curry} is that f takes a pair (g, x) as an argument, whereas f_{curry} takes a single argument g. However, f_{curry} can be used in place of f because

$$f_{\text{curry}}g\ x = gx = f(g,\ x)$$

Thus, in the end, f_{curry} does the same thing as f . This simple idea was discovered by Schönfinkel, who investigated functionality in the 1920s. However, this technique for representing multiargument functions in lambda calculus is usually called *Currying*, after the lambda calculus pioneer Haskell Curry.

Declarations

We saw in Subsection 4.2.2 that we can regard simple let declarations,

$$\text{let } x = M \text{ in } N$$

as lambda terms by adopting the abbreviation

$$\text{let } x = M \text{ in } N = (\lambda x.N)M$$

The let construct may be used to define a composition function, as in the expression

$$\text{let } compose = \lambda f.\lambda g.\lambda x.f(gx) \text{ in } compose\ h\ h$$

Using β-equivalence, we can simplify this let expression to $\lambda\, x.\ h(hx)$, the composition of h with itself. In programming language parlance, the let construct provides local declarations.

Recursion and Fixed Points

An amazing fact about pure lambda calculus is that it is possible to write recursive functions by use of a self-application "trick." This does not have a lot to do with comparisons between modern programming languages, but it may interest readers with a technical bent. (Some readers and some instructors may wish to skip this subsection.)

Many programming languages allow recursive function definitions. The characteristic property of a recursive definition of a function f is that the body of the function contains one or more calls to f. To choose a specific example, let us suppose we define the factorial function in some programming language by writing a declaration like

```
function f(n) {if n=0 then 1 else n*f(n-1)};
```

where the body of the function is the expression inside the braces. This definition has a straightforward computational interpretation: when f is called with argument a, the function parameter n is set to a and the function body is evaluated. If evaluation of the body reaches the recursive call, then this process is repeated. As definitions of a computational procedure, recursive definitions are clearly meaningful and useful.

One way to understand the lambda calculus approach to recursion is to associate an equation with a recursive definition. For example, we can associate the equation

```
f(n) = if n=0 then 1 else n*f(n-1)
```

$2 + 1 + 1$

$\rightarrow 3 + 1$

$\rightarrow 4.$

This example should give a good idea of how reduction in lambda calculus corresponds to computation. Since the 1930s, lambda calculus has been a simple mathematical model of expression evaluation.

Confluence

In our example starting with $(\lambda f.\lambda x.\ f\ (fx))\ (\lambda y.y + 1)$ 2, there were some steps in which we had to chose from several possible subexpressions to evaluate. For example, in the second expression,

$$(\lambda x.(\lambda y.y + 1)((\lambda y.y + 1)x))2,$$

we could have evaluated either of the two function calls beginning with λy. This is not an artifact of lambda calculus itself, as we also have two choices in evaluating the purely arithmetic expression

$$2 + 1 + 1.$$

An important property of lambda calculus is called *confluence*. In lambda calculus, as a result of confluence, evaluation order does not affect the final value of an expression. Put another way, if an expression M can be reduced to a normal form, then there is exactly one normal form of M, independent of the order in which we choose to evaluate subexpressions. Although the full mathematical statement of confluence is a bit more complicated than this, the important thing to remember is that, in lambda calculus, expressions can be evaluated in any order.

4.2.5 Important Properties of Lambda Calculus

In summary, lambda calculus is a mathematical system with some syntactic and computational properties of a programming language. There is a general notation for functions that includes a way of treating an expression as a function of some variable that it contains. There is an equational proof system that leads to calculation rules, and these calculation rules are a simple form of symbolic evaluation. In programming language terminology, these calculation rules are a form of macro expansion(with renaming of bound variables!) or function in lining. Because of the relation to in lining, some common compiler optimizations may be defined and proved correct by use of lambda calculus.

Lambda calculus has the following important properties:

- Every computable function can be represented in pure lambda calculus. In the terminology of Chapter 2, lambda calculus is Turing complete. (Numbers can be represented by functions and recursion can be expressed by Y.)
- Evaluation in lambda calculus is order independent. Because of confluence, we can evaluate an expression by choosing any subexpression. Evaluation in pure functional programming languages (see Section 4.4) is also confluent, but evaluation in languages whose expressions may have side effects is not confluent.

Macro expansion is another setting in which a form of evaluation is confluent. If we start with a program containing macros and expand all macro calls with the macro bodies, then the final fully expanded program we obtain will not depend on the order in which macros are expanded.

4.3 DENOTATIONAL SEMANTICS

In computer science, the phrase denotational semantics refers to a specific style of mathematical semantics for imperative programs. This approach was developed in the late 1960s and early 1970s, following the pioneering work of Christopher Strachey and Dana Scott at Oxford University. The term denotational semantics suggests that a meaning or *denotation* is associated with each program or program phrase (expression, statement, declaration, etc.). The denotation of a program is a mathematical object, typically a function, as opposed to an algorithm or a sequence of instructions to execute.

In denotational semantics, the meaning of a simple program like

```
x := 0; y:=0; while x ≤ z do y:=y+x;x:=x+1
```

is a mathematical function from *states* to *states*, in which a state is a mathematical function representing the values in memory at some point in the execution of a program. Specifically, the meaning of this program will be a function that maps any state in which the value of *z* is some nonnegative integer *n* to the state in which *x*=*n*, *y* is the sum of all numbers up to *n*, and all other locations in memory are left unchanged. The function would not be defined on machine states in which the value of *z* is not a nonnegative integer.

Associating mathematical functions with programs is good for some purposes and not so good for others. In many situations, we consider a program correct if we get the correct output for any possible input. This form of correctness depends on only the denotational semantics of a program, the mathematical function from input to output associated with the program. For example, the preceding program was designed to compute the sum of all the non negative integers up to *n*. If we verify that the actual denotational semantics of this program is this mathematical function, then we have proved that the program is correct. Some disadvantages of denotational semantics are that standard denotational semantics do not tell us anything about the running time or storage requirements of a program. This is sometimes an advantage in disguise because, by ignoring these issues, we can sometimes reason more effectively about the correctness of programs.

Forms of denotational semantics are commonly used for reasoning about program optimization and static analysis methods. If we are interested in analyzing running time, then operational semantics might be more useful, or we could use more detailed denotational semantics that also involves functions representing time bounds. An alternative to denotational semantics is called operational semantics, which involves modeling machine states and (generally) the step-by-step state transitions associated with a program. Lambda calculus reduction is an example of operational semantics.

Compositionality

An important principle of denotational semantics is that the meaning of a program is determined from its text *compositionally*. This means that the meaning of a program must be defined from the meanings of its parts, not something else, such as the text of its parts or the meanings of related programs obtained by syntactic operations. For example, the denotation of a program such as *if B then P else Q* must be explained with only the denotations of *B*, *P*, and *Q*; it should not be defined with programs constructed from *B*, *P*, and *Q* by syntactic operations such as substitution.

The importance of compositionality, which may seem rather subtle at first, is that if two program pieces have the same denotation, then either may safely be substituted for the other in any program. More specifically, if *B*, *P*, and *Q* have the same denotations as *B* ', *P* ', and *Q* ', respectively, then *if B then P else Q* must have the same denotation as *if B' then P' else Q'*. Compositionality means that the denotation of an expression or program statement must be detailed enough to capture everything that is relevant to its behavior in larger programs. This makes denotational semantics useful for understanding and reasoning about such pragmatic issues as program transformation and optimization, as these operations on programs involve replacing parts of programs without changing the overall meaning of the whole program.

4.3.1 Object Language and Metalanguage

One source of confusion in talking (or writing) about the interpretation of syntactic expressions is that everything we write is actually syntactic. When we study a programming language, we need to distinguish the programming language we study from the language we use to describe this language and its meaning. The language we study is traditionally called the object language, as this is the object of our attention, whereas the second language is called the metalanguage, because it transcends the object language in some way.

```
P ::=x:=e | P; P | if e then P else P | while e do P
```

where we assume that *x* has the appropriate type to be assigned the value of e in the assignment x :=e and that the test *e* has type *bool* in if-then-else and while statements. Because this language does not have explicit input or output, the effect of a program will be to change the values of variables. Here is a simple example:

```
x := 0; y:=0; while x ≤ z do (y := y+x; x := x+1)
```

We may think of this program as having input *z* and output *y*. This program uses an additional variable *x* to set *y* to the sum of all natural numbers up to *z*.

States and Commands

The meaning of a program is a function from states to states. In a more realistic programming language, with procedures or pointers, it is necessary to model the fact that two variable names may refer to the same location. However, in the simple language of while programs, we assume that each variable is given a different location. With this simplification in mind, we let the set *State* of mathematical representations of machine states be

$$State = Variables \to Values$$

In words, a state is a function from variables to values. This is an idealized view of machine states in two ways: We do not explicitly model the locations associated with variables, and we use infinite states that give a value to every possible variable.

The meaning of a program is an element of the mathematical set *Command* of commands, defined by

$$Command = State \to State$$

In words, a command is a function from states to states. Unlike states themselves, which are total functions, a command may be a *partial* function. The reason we need partial functions is that a program might not terminate on an initial state.

A basic function on states that is used in the semantics of assignment is *modify*,which is defined as follows:

modify(s,x,a) = λ *v* ? *Variables*. if *v*=*x* then *a* else *s*(*v*)

In words, *modify*(*s,x,a*) is the state (function from variables to values) that is just like state *s*, except that the value of *x* is *a*.

Denotational Semantics

The denotational semantics of while programs is given by the definition of a function C from parsed programs to commands. As with expressions, we write [[P]] for a parse tree of the program P. The semantics of programs are defined by the following clauses, one for each syntactic form of program:

C[[x :=e]](s) =modify(s,x, E[[e]](s))

C[[P_1;P_2]](s) =C[[P_2]](C[[P_1]](s))

Similarly, C[[while x=y do x := y]](s) is s if s(x) ≠ s(y) and undefined otherwise. If you are interested in more information, there are many books that cover denotational semantics in detail.

4.3.4 Perspective and Nonstandard Semantics

There are several ways of viewing the standard methods of denotational semantics. Typically, denotational semantics is given by the association of a function with each program. As many researchers in denotational semantics have observed, a mapping from programs to functions must be written in some metalanguage. Because lambda calculus is a useful notation for functions, it is common to use some form of lambda calculus as a metalanguage. Thus, most denotational semantics actually have two parts: a translation of programs into a lambda calculus (with some extra operations corresponding to basic operations in programs) and a semantic interpretation of the lambda calculus expressions as mathematical objects. For this reason, denotational semantics actually provides a general technique for translating imperative programs into functional programs.

Although the original goal of denotational semantics was to define the meanings of programs in a mathematical way, the techniques of denotational semantics can also be used to define useful "nonstandard" semantics of programs.

One useful kind of nonstandard semantics is called *abstract interpretation*. In abstract interpretation, programs are assigned meaning in some simplified domain. For example, instead of interpreting integer expressions as integers, integer expressions could be interpreted elements of the finite set {0, 1, 2, 3, 4, 5, …, 100, >100}, where>100 is a value used for expressions whose value might be greater than 100. This might be useful if we want to see if two array expressions A[e_1] and A[e_2] refer to the same array location. More specifically, if we assign values to e_1 from the preceding set, and similarly for e_2, we might be able to determine that e_1=e_2. If both are assigned the value >100, then we would not know that they are the same, but if they are assigned the same ordinary integer between 0 and 100, then we would know that these expressions have the same value. The importance of using a finite set of values is that an algorithm could iterate over all possible states. This is important for calculating properties of programs that hold in all states and also for calculating the semantics of loops.

Example. Suppose we want to build a program analysis tool that checks programs to make sure that every variable is initialized before it is used. The basis for this kind of program analysis can be described by denotational semantics, in which the meaning of an expression is an element of a finite set.

Because the halting problem is unsolvable, program analysis algorithms are usually designed to be *conservative.* Conservative means that there are no false positivies: An algorithm will output *correct* only if the program is correct, but may sometimes output *error* even if there is no error in the program. We cannot expect a computable analysis to decide correctly whether a program will ever access a variable before it is initialized. For example, we cannot decide whether

```
(complictated error-free program); x := y
```

executes the assignment x:=y without deciding whether complictated error-free program halts. However, it is often possible to develop efficient algorithms for conservative analysis. If you think about it, you will realize that most compiler warnings are conservative: Some warnings could be a problem in general, but are not a problem in a specific program because of program properties that the compiler is not "smart" enough to understand.

We describe initialize-before-use analysis by using an abstract representation of machine states that keep track only of whether a variable has been initialized or not. More precisely, a state will either be a special error state called *error* or a function from variable names to the set { *init*, *uninit* } with two values, one representing any value of initialized variable and the other an uninitialized one. The set *State* of mathematical abstractions of machine states is therefore

State ={error} ? (*Variables* → {*init, uninit*})

4.4 FUNCTIONAL AND IMPERATIVE LANGUAGES

4.4.1 Imperative and Declarative Sentences

The languages that humans speak and write are called *natural languages* as they developed naturally, without concern for machine readability. In natural languages, there are four main kinds of sentences: imperative, declarative, interrogative, and exclamatory.

In an imperative sentence, the subject of the sentence is implicit. For example, the subject of the sentence

Pick up that fish

is (implicitly) the person to whom the command is addressed. A declarative sentence expresses a fact and may consist of a subject and a verb or subject, verb, and object. For example,

Claude likes bananas

is a declarative sentence.

Interrogatives are questions. An exclamatory sentence may consist of only an interjection, such as *Ugh!* or *Wow!*

In many programming languages, the basic constructs are imperative statements. For example, an assignment statement such as

```
x:=5
```

is a command to the computer (the implied subject of the utterance) to store the value5 in a certain location. Programming languages also contain declarative constructs such as the function declaration

```
function f(int x) {return x+1;}
```

that states a fact. One reading of this as a declarative sentence is that the subject is the name f and the sentence about f is "f is a function whose return value is 1 greater than its argument."

In programming, the distinction between imperative and declarative constructs rests on the distinction between changing an existing value and declaring a new value. The first is imperative, the latter declarative. For example, consider the following program fragment:

```
{ int x=1;              /* declares new x */
    x = x+1;             /* assignment to existing x */
    { int y = x+1;       /* declares new y */
        { int x = y+1;    /* declares new x */
 }}}
```

Here, only the second line is an imperative statement. This is an imperative command that changes the state of the machine by storing the value 2 in the location associated with variable x. The other lines contain declarations of new variables.

A subtle point is that the last line in the preceding code declares a new variable with the same name as that of a previously declared variable. The simplest way to understand the distinction between declaring a new variable and changing the value of an old one is by variable renaming. As we saw in lambda calculus, a general principle of binding is that bound variables can be renamed without changing the meaning of an expression or program. In particular, we can rename bound variables in the preceding program fragment to get

```
{ int x=1;              /* declares new x */
    x = x+1;             /* assignment to existing x */
    { int y = x+1;       /* declares new y */
        { int z = y+1;    /* declares new z */
}}}
```

(If there were additional occurrences of x inside the inner block, we would rename them to z also.) After rewriting the program to this equivalent form, we easily see that the declaration of a new variable z does not change the value of any previously existing variable.

4.4.2 Functional versus Imperative Programs

The phrase *functional language* is used to refer to programming languages in which most computation is done by evaluation of expressions that contain functions. Two examples are Lisp and ML. Both of these languages contain declarative and imperative constructs. However, it is possible to write a substantial program in either language without using any imperative constructs.

Some people use the phrase functional language to refer to languages that do not have expressions with side effects or any other form of imperative construct. However, we will use the more emphatic phrase *pure functional language* for declarative languages that are designed around flexible constructs for defining and using functions. We learned in Subsection 3.4.9 that pure Lisp, based on atom, eq, car, cdr, cons, lambda, define, is a pure functional language. If rplaca, which changes the car of a cell, and rplacd, which changes the cdr of a cell, are added, then the resulting Lisp is not a pure functional language.

Pure functional languages pass the following test:

Declarative Language Test: Within the scope of specific declarations of $x_1, \ldots, x_n$, all occurrences of an expression e containing only variables $x_1, \ldots, x_n$ have the same value.

As a consequence, pure functional languages have a useful optimization property: If expression e occurs several places within a specific scope, this expression needs to be evaluated only once. For example, suppose a program written in pure Lisp contains two occurrences of (cons a b). An optimizing Lisp compiler could compute (cons a b)once and use the same value both places. This not only saves time, but also space, as evaluating cons would ordinarily involve a new cell.

Referential Transparency

In some of the academic literature on programming languages, including some textbooks on programming language semantics, the concept that is used to distinguish declarative from imperative languages is called *referential transparency*. Although it is easy to define this phrase, it is a bit tricky to use it correctly to distinguish one programming language from another.

In linguistics, a name or noun phrase is considered *referentially transparent* if it may be replaced with another noun phrase with the same referent (i.e., referring to the same thing) without changing the meaning of the sentence that contains it. For example, consider the sentence

I saw Walter get into *his car.*

If Walter owns a Maserati Biturbo, say, and no other car, then the sentence

I saw Walter get into *his Maserati Biturbo*

has the same meaning because the noun phrases (in italics) have the same meaning. A traditional counter example to referential transparency, attributed to the language philosopher Willard van Orman Quine, occurs in the sentence

He was called *William Rufus* because of his red beard.

The sentence refers to William IV of England and rufus means reddish or orange in color. If we replace William Rufus with William IV, we get a sentence that makes no sense:

He was called *William IV* because of his red beard.

Obviously, the king was called William IV because he was the fourth William, not because of the color of his beard.

Returning to programming languages, it is traditional to say that a language is referentially transparent if we may replace one expression with another of equal value anywhere in a program without changing the meaning of the program. This is a property of pure functional languages.

The reason referential transparency is subtle is that it depends on the value we associate with expressions. In imperative programming languages, we can say that a variable x refers to its value or to its location. If we say that a variable refers to its location in memory, then imperative languages *are* referentially transparent, as replacing one variable with another that names the same memory location will not change the meaning of the program. On the other hand, if we say that a variable refers to the value stored in that location, then imperative languages are not referentially transparent, as the value of a variable may change as the result of assignment.

Historical Debate

John Backus received the 1977 ACM Turing Award for the development of Fortran. In his lecture associated with this award, Backus argued that pure functional programming languages are better than imperative ones. The lecture and the accompanying paper, published by the Association for Computing Machinery, helped inspire a number of research projects aimed at developing practical pure functional programming languages. The main premise of Backus' argument is that pure functional programs are easier to reason about because we can understand expressions independently of the context in which they occur.

Backus asserts that, in the long run, program correctness, readability, and reliability are more important than other factors such as efficiency. This was a controversial position in 1977, when programs were a lot smaller than they are today and computers were much slower. In the 1990s, computers finally reached the stage at which commercial organizations began to choose software development methods that value programmer development time over structures that can be defined. FP languages that allow variable names and binding (as in Lisp and lambda calculus) have been more successful, as have all programming languages that support modularity and reuse of library components. However, Backus raised an important issue that led to useful reflection and debate.

Functional Programming and Concurrency

An appealing aspect of pure functional languages and of programs written in the pure functional subset of larger languages is that programs can be executed concurrently. This is a consequence of the declarative language test mentioned at the beginning of this subsection. We can see how parallelism arises in pure functional languages by using the example of a function call $f(e_1,\dots,e_n)$, where function arguments $e_1,\dots,e_n$ are expressions that may need to be evaluated.

> *Functional Programming:* We can evaluate $f(e_1,\dots,e_n)$ by evaluating $e_1,\dots,e_n$ in parallel because values of these expressions are independent.
>
> *Imperative Programming:* For an expression such as f(g(x), h(x)), the function g might change the value of x. Hence the arguments of functions in imperative
>
> languages must be evaluated in a fixed, sequential order. This order ing restricts the use of concurrency.

Backus used the term *von Neumann bottleneck* for the fact that in executing an imperative program, computation must proceed one step at a time. Because each step in a program may depend on the previous one, we have to pass values one at a time from memory to the CPU and back. This sequential channel between the CPU and memory is what he called the von Neumann bottleneck.

Although functional programs provide the opportunity for parallelism, and parallelism is often an effective way of increasing the speed of computation, effectively taking advantage of inherent parallelism is difficult. One problem that is fairly easy to understand is that functional programs sometimes provide too much parallelism. If all possible computations are performed in parallel, many more computation steps will be executed than necessary. For example, full parallel evaluation of a conditional

```
if e1 then e2 else e3
```

will involve evaluating all three expressions. Eventually, when the value of e_1 is found, one of the other computations will turn out to be irrelevant. In this case, the irrelevant computation can be terminated, but in the meantime, resources will have been devoted to calculation that does not matter in the end.

In a large program, it is easy to generate so many parallel tasks that the time setting up and switching between parallel processes will detract from the efficiency of the computation. In general, parallel programming languages need to provide some way for a programmer to specify where parallelism may be beneficial. Parallel implementations of functional languages often have the drawback that the programmer has little control over the amount of parallelism used in execution.

Practical Functional Programming

Backus' Turing Award lecture raises a fundamental question:

> Do pure functional programming languages have significant practical advantages over imperative languages?

Although we have considered many of the potential advantages of pure FP languages in this section, we do not have a

4.5 CHAPTER SUMMARY

In this chapter, we studied the following topics:

- the outline of a simple compiler and parsing issues,
- lambda calculus,
- denotational semantics,
- the difference between functional and imperative languages.

A standard compiler transforms an input program, written in a source language, into an output program, written in a target language. This process is organized into a series of six phases, each involving more complex properties of programs. The first three phases, lexical analysis, syntax analysis, and semantic analysis, organize the input symbols into meaningful tokens, construct a parse tree, and determine context-dependent properties of the program such as type agreement of operators and operands. (The name semantic analysis is commonly used in compiler books, but is somewhat misleading as it is still analysis of the parse tree for context-sensitive syntactic conditions.) The last three phases, intermediate code generation, optimization, and target code generation, are aimed at producing efficient target code through language transformations and optimizations.

Lambda calculus provides a notation and symbolic evaluation mechanism that is useful for studying some properties of programming languages. In the section on lambda calculus, we discussed binding and α conversion. Binding operators arise in many programming languages in the form of declarations and in parameter lists of functions, modules, and templates. Lambda expressions are symbolically evaluated by use of β-reduction, with the function argument substituted in place of the formal parameter. This process resembles macro expansion and function in lining, two transformations that are commonly done by compilers. Although lambda calculus is a very simple system, it is theoretically possible to write every computable function in the lambda calculus. Untyped lambda calculus, which we discussed, can be extended with type systems to produce various forms of typed lambda calculus.

Denotational semantics is a way of defining the meanings of programs by specifying the mathematical value, function, or function on functions that each construct denotes. Denotational semantics is an abstract way of analyzing programs because it does not consider issues such as running time and memory requirements. However, denotational semantics is useful for reasoning about correctness and has been used to develop and study program analysis methods that are used in compilers and programming environments. Some of the exercises at the end of the chapter present applications for type checking, initialization-before-use analysis, and simplified security analysis. From a theoretical point of view, denotational semantics shows that every imperative program can be transformed into an equivalent functional program.

In pure functional programs, syntactically identical expressions within the same scope have identical values. This property allows certain optimizations and makes it possible to execute independent subprograms in parallel. Because functional languages are theoretically as powerful as imperative languages, we discussed some of the pragmatic differences between functional and imperative languages. Although functional languages may be simpler to reason about in certain ways, imperative languages often make it easier to write efficient programs. Although Backus argues that functional programs can eliminate the von Neumann bottleneck, practical parallel execution of functional programs has not proven as successful as he anticipated in his Turing Award lecture.