4

Fundamentals

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 areas 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

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

JOHN BACKUS

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.

I met John Backus through IFIP WG 2.8, a working group of the International Federation of Information Processing on functional programming. Backus continued to work on functional programming at IBM Almaden through the 1980s, although his group was disbanded after his retirement. A mild-mannered and unpretentious individual, here is a quote that gives some sense of his independent, pioneering spirit:

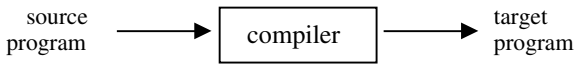
"I really didn't know what the hell I wanted to do with my life. I decided that what I wanted was a good hi fi set because I liked music. In those days, they didn't really exist so I went to a radio technicians' school. I had a very nice teacher – the first good teacher I ever had – and he asked me to cooperate with him and compute the characteristics of some circuits for a magazine."

"I remember doing relatively simple calculations to get a few points on a curve for an amplifier. It was laborious and tedious and horrible, but it got me interested in math. The fact that it had an application – that interested me."

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.

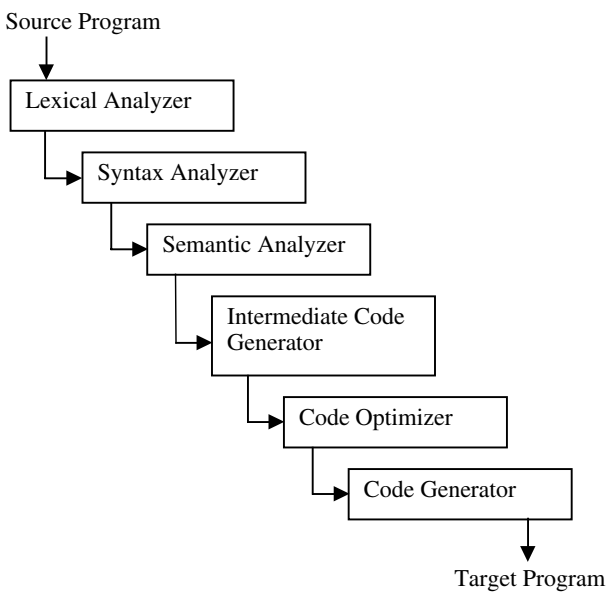
https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

The main function of a compiler is illustrated in this simple diagram:



Given a program in some *source language*, the compiler produces a program in a *target language*, which is usually the instruction set, or machine language, of some machine.

Most compilers are structured as a series of phases, with each phase performing one step in the translation of source program to target program. A typical compiler might consist of the phases shown in the following diagram:



Each of these phases is discussed briefly. Our goal with this book is only to understand the parts of a compiler so that we can discuss how different programming language features might be implemented. We do not discuss how to build a compiler. That is the subject of many books on compiler construction, such as *Compilers: Principles, Techniques and 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.

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

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.

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

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

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

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 $s \to tu$, 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 nonterminals 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!):

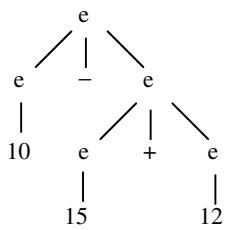
$$\texttt{e} \longrightarrow \texttt{n} \longrightarrow \texttt{nd} \longrightarrow \texttt{dd} \longrightarrow \texttt{2d} \longrightarrow \texttt{25}$$
$$\texttt{e} \longrightarrow \texttt{e-e} \longrightarrow \texttt{e-e+e} \longrightarrow \cdots \longrightarrow \texttt{n-n+n} \longrightarrow \cdots \longrightarrow \texttt{10-15+12}$$

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

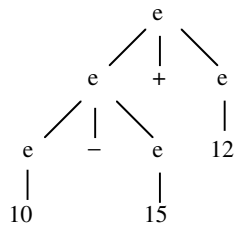
Parse Trees and Ambiguity

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

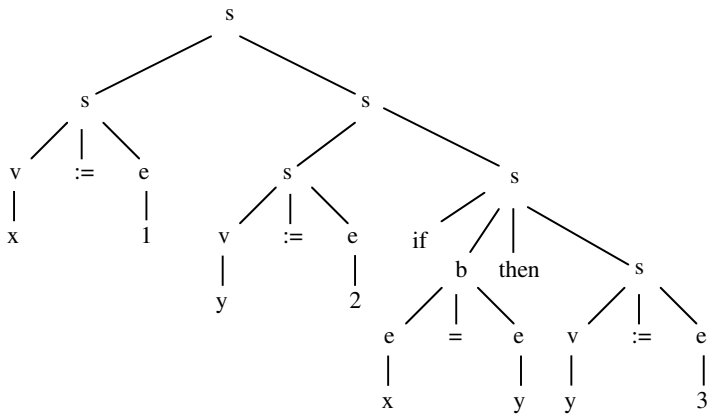
There is an interesting ambiguity involving if-then-else. This can be illustrated by the following simple grammar:

```
s ::= v := e | s;s | if b then s | if b then s else s
v ::= x | y | z
e ::= v | 0 | 1 | 2 | 3 | 4
b ::= e=e
```

where s is the start symbol, s, v, e, and b are nonterminals, and the other symbols are terminals. The letters s, v, e, and b stand for statement, variable, expression, and Boolean test, respectively. We call the expressions of the language generated by this

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

grammar *statements*. Here is an example of a well-formed statement and one of its parse trees:




```
x := 1; y := 2; if x=y then y := 3
```

This statement also has another parse tree, which we obtain by putting two assignments to the left of the root and the if-then statement to the right. However, the difference between these two parse trees will not affect the behavior of code generated by an ordinary compiler. The reason is that $s_1;s_2$ is normally compiled to the code for s_1 followed by the code for s_2. As a result, the same code would be generated whether we consider $s_1;s_2;s_3$ as $(s_1;s_2);s_3$ or $s_1;(s_2;s_3)$.

A more complicated situation arises when if-then is combined with if-then-else in the following way:

```
if b1 then if b2 then s1 else s2
```

What should happen if b_1 is true and b_2 is false? Should s_2 be executed or not? As you can see, this depends on how the statement is parsed. A grammar that allows this combination of conditionals is ambiguous, with two possible 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

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

part of compiling programming languages, parsing is usually covered in courses and textbooks on compilers. For most programming languages you might consider, it is either straightforward 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 $\texttt{e}\ \texttt{op}_1\ \texttt{e}\ \texttt{op}_2\ \texttt{e}$ 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 expression 5*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 expression like this could be disambiguated by use of an additional convention.

Associativity comes into play when two operators of equal precedence appear next to each other. Under left associativity, an expression $\texttt{e}\ \texttt{op}_1\ \texttt{e}\ \texttt{op}_2\ \texttt{e}$ would be parsed as $(\texttt{e}\ \texttt{op}_1\ \texttt{e})\ \texttt{op}_2\ \texttt{e}$, if the two operators have equal precedence. If we adopted a right-associativity convention instead, $\texttt{e}\ \texttt{op}_1\ \texttt{e}\ \texttt{op}_2\ \texttt{e}$ would be parsed as $\texttt{e}\ \texttt{op}_1\ (\texttt{e}\ \texttt{op}_2\ \texttt{e})$.

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

Expression	Precedence	Left Associativity	Right Associativity
5∗4-3	(5∗4)-3	(no change)	(no change)
1+1-1	(no change)	(1+1)-1	1+(1-1)
2+3-4∗5+2	2+3-(4∗5)+2	((2+3)-(4*5))+2	2+(3-((4*5)+2))

4.2 LAMBDA CALCULUS

Lambda calculus is a mathematical system that illustrates some important programming language concepts in a simple, pure form. Traditional lambda calculus has three main parts: a notation for defining functions, a proof system for proving equations between expressions, and a set of calculation rules called reduction. The first word in the name lambda calculus comes from the use of the Greek letter lambda (λ) in function expressions. (There is no significance to the letter λ.) The second word comes from the way that reduction may be used to *calculate* the result of applying a function to one or more arguments. This calculation is a form of symbolic evaluation of expressions. Lambda calculus provides a convenient notation for describing programming languages and may be regarded as the basis for many language constructs. In particular, lambda calculus provides fundamental forms of parameterization (by means of function expressions) and binding (by means of declarations). These are basic concepts that are common to almost all modern programming languages. It is therefore useful to become familiar enough with lambda calculus to regard expressions in your favorite programming language as essentially a form of lambda expression. For simplicity, the untyped lambda calculus is discussed; there are also typed versions of lambda calculus. In typed lambda calculus, there are additional type-checking constraints that rule out certain forms of expressions. However, the basic concepts and calculation rules remain essentially the same.

4.2.1 Functions and Function Expressions

Intuitively, a function is a rule for determining a value from an argument. This view of functions is used informally in most mathematics. (See Subsection 2.1.2 for a discussion of functions in mathematics.) Some examples of functions studied in mathematics are

$$f(x) = x^2 + 3,$$

$$g(x + y) = \sqrt{x^2 + y^2}.$$

In the simplest, pure form of lambda calculus, there are no domain-specific operators such as addition and exponentiation, only function definition and application. This allows us to write functions such as

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

because h is defined solely in terms of function application and other functions that we assume are already defined. It is possible to add operations such as addition

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

and exponentiation to pure lambda calculus. Although purists stick to pure lambda calculus without addition and multiplication, we will use these operations in examples as this makes the functions we define more familiar.

The main constructs of lambda calculus are *lambda abstraction* and *application*. We use lambda abstraction to write functions: If M is some expression, then $\lambda x.M$ is the function we get by treating M as a function of the variable x. For example,

$$\lambda x.x$$

is a lambda abstraction defining the identity function, the function whose value at x is x. A more familiar way of defining the identity function is by writing

$$I(x) = x.$$

However, this way of defining a function forces us to make up a name for every function we want. Lambda calculus lets us write anonymous functions and use them inside larger expressions.

In lambda notation, it is traditional to write function application just by putting a function expression in front of one or more arguments; parentheses are optional. For example, we can apply the identity function to the expression M by writing

$$(\lambda x.x)M.$$

The value of this application is the identity function, applied to M, that just ends up being M. Thus, we have

$$(\lambda x.x)M = M.$$

Part of lambda calculus is a set of rules for deducing equations such as this. Another example of a lambda expression is

$$\lambda f.\lambda g.\lambda x.\, f(g\, x).$$

Given functions f and g, this function produces the composition $\lambda x.\, f(g\, x)$ of f and g.

We can extend pure lambda calculus by adding a variety of other constructs. We call an extension of pure lambda calculus with extra operations an *applied lambda calculus*. A basic idea underlying the relation between lambda calculus and computer science is the slogan

$$\begin{aligned}\text{Programming language} &= \text{applied } \lambda\text{-calculus} \\ &= \text{pure } \lambda\text{-calculus} + \text{additional data types.}\end{aligned}$$

This works even for programming languages with side effects, as the way a program depends on the state of the machine can be represented by explicit data structures for the machine state. This is one of the basic ideas behind Scott–Strachey denotational semantics, as discussed in Section 4.3.

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

4.2.2 Lambda Expressions

Syntax of Expressions

The syntax of untyped lambda expressions may be defined with a BNF grammar. We assume we have some infinite set V of *variables* and use $x, y, z, \ldots$, to stand for arbitrary variables. The grammar for lambda expressions is

$$M ::= x \mid MM \mid \lambda x.M$$

where x may be any variable (element of V). An expression of the form $M_1 M_2$ is called an *application*, and an expression of the form $\lambda x.M$ is called a *lambda abstraction.* Intuitively, a variable x refers to some function, the particular function being determined by context; $M_1 M_2$ is the application of function M_1 to argument M_2; and $\lambda x.M$ is the function that, given argument x, returns the value M. In the lambda calculus literature, it is common to refer to the expressions of lambda calculus as *lambda terms*.

Here are some example lambda terms:

$\lambda x.x$	a lambda abstraction called the *identity function*
$\lambda x.(f(gx))$	another lambda abstraction
$(\lambda x.x)5$	an application

There are a number of syntactic conventions that are generally convenient for lambda calculus experts, but they are confusing to learn. We can generally avoid these by writing enough parentheses. One convention that we will use, however, is that in an expression containing a λ, the scope of λ extends as far to the right as possible. For example, $\lambda x.xy$ should be read as $\lambda x.(xy)$, not $(\lambda x.x)y$.

Variable Binding

An occurrence of a variable in an expression may be either free or bound. If a variable is *free* in some expression, this means that the variable is not declared in the expression. For example, the variable x is free in the expression $x + 3$. We cannot evaluate the expression $x + 3$ as it stands, without putting it inside some larger expression that will associate some value with x. If a variable is not free, then that must be because it is *bound*.

The symbol λ is called a *binding operator*, as it binds a variable within a specific scope (part of an expression). The variable x is bound in $\lambda x.M$. This means that x is just a place holder, like x in the integral $\int f(x)\,\mathrm{dx}$ or the logical formula $\forall x.P(x)$, and the meaning of $\lambda x.M$ does not depend on x. Therefore, just as $\int f(x)\,dx$ and $\int f(y)\,dy$ describe the same integral, we can rename a λ-bound x to y without changing the meaning of the expression. In particular,

$\lambda x.x$ defines the same function as $\lambda y.y$.

Expressions that differ in only the names of bound variables are called α equivalent. When we want to emphasize that two expressions are α equivalent, we write $\lambda x.x =_\alpha \lambda y.y$, for example.

In $\lambda x.M$, the expression M is called the *scope* of the binding λx. A variable x appearing in an expression M is bound if it appears in the scope of some λx and is free otherwise. To be more precise about this, we may define the set $\mathrm{FV}(M)$ of *free*

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

variables of M by induction on the structure of expressions, as follows:

$$\mathrm{FV}(x) = \{x\},$$
$$\mathrm{FV}(MN) = \mathrm{FV}(M) \cup \mathrm{FV}(N),$$
$$\mathrm{FV}(\lambda x.M) = \mathrm{FV}(M) - x,$$

where $-$ means set difference, that is, $S - x = \{y \in S \mid y \neq x\}$. For example, $\mathrm{FV}(\lambda x.x) = \emptyset$ because there are no free variables in $\lambda x.x$, whereas FV $(\lambda f.\lambda x.(f(g(x))) = \{g\}$.

It is sometimes necessary to talk about different occurrences of a variable in an expression and to distinguish free and bound occurrences. In the expression

$$\lambda x.(\lambda y.xy)y,$$

the first occurrence of x is called a *binding occurrence*, as this is where x becomes bound. The other occurrence of x is a bound occurrence. Reading from left to right, we see that the first occurrence of y is a binding occurrence, the second is a bound occurrence, and the third is free as it is outside the scope of the λy in parentheses.

It can be confusing to work with expressions that use the same variable in more than one way. A useful convention is to rename bound variables so that all bound variables are different from each other and different from all of the free variables. Following this convention, we will write $(\lambda y.(\lambda z.z)y)x$ instead of $(\lambda x.(\lambda x.x)x)x$. The variable convention will be particularly useful when we come to equational reasoning and reduction.

Lambda Abstraction in Lisp and Algol

Lambda calculus has an obvious syntactic similarity to Lisp: the Lisp lambda expression

```
(lambda (x) function_body)
```

looks like the lambda calculus expression

$$\lambda x.\, function\_body,$$

and both expressions define functions. However, there are some differences between Lisp and lambda calculus. For example, lists are the basic data type of Lisp, whereas functions are the only data type in pure lambda calculus. Another difference is evaluation order, but that topic will not be discussed in detail.

Although the syntax of block-structured languages is farther from lambda calculus than from Lisp, the basic concepts of declarations and function parameterizations in Algol-like languages are fundamentally the same as in lambda calculus. For example, the C program fragment

```
int f (int x) {return x};
block_body;
```

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

with a function declaration at the beginning of a block is easily translated into lambda calculus. The translation is easier to understand if we first add declarations to lambda calculus.

The simple let declaration,

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

which declares that x has value M in the body N, may be regarded as syntactic sugar for a combination of lambda abstraction and application:

$$\text{let } x = M \text{ in } N \quad \textit{is sugar for} \quad (\lambda x.N)M.$$

For those not familiar with the concept of *syntactic sugar*, this means that let $x = M$ in N is "sweeter" to write in some cases, but we can just think of the syntax let $x = M$ in N as standing for $(\lambda x.N)M$.

Using let declarations, we can write the C program from above as

```
let f = (λx.x) in block_body
```

Note that the C form expression and the lambda expression have the same free and bound variables and a similar structure. One difference between C and lambda calculus is that C has assignment statements and side effects, whereas lambda calculus is purely functional. However, by introducing *stores*, mappings from variable locations to values, we can translate C programs with side effects into lambda terms as well. The translation preserves the overall structure of programs, but makes programs look a little more complicated as the dependencies on the state of the machine are explicit.

Equivalence and Substitution

We have already discussed α equivalence of terms. This is one of the basic axioms of lambda calculus, meaning that it is one of the properties of lambda terms that defines the system and that is used in proving other properties of lambda terms. Stated more carefully than before, the equational proof system of lambda calculus has the equational axiom

$$\lambda x.M = \lambda y.[y/x]M, \qquad (\alpha)$$

where $[y/x]M$ is the result of substituting y for free occurrences of x in M and y cannot already appear in M. There are three other equational axioms and four inference rules for proving equations between terms. However, we look at only one other equational axiom.

The central equational axiom of lambda calculus is used to calculate the value of a function application $(\lambda x.M)N$ by substitution. Because $\lambda x.M$ is the function we get by treating M as a function of x, we can write the value of this function at N by substituting N for x. Again writing $[N/x]M$ for the result of substituting N for free occurrences of x in M, we have

$$(\lambda x.M)N = [N/x]M, \qquad (\beta)$$

which is called the axiom of β-equivalence. Some important warnings about

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

substitution are discussed in the next subsection. The value of $\lambda f.fx$ applied to $\lambda y.y$ may be calculated if the argument $\lambda y.y$ is substituted in for the bound variable f:

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

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

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

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

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

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

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

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:

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

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

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

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

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

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

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

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)
```

with the preceding recursive declaration. This equation states a property of factorial.

Specifically, the value of the expression f(n) is equal to the value of the expression if n=0 then 1 else n*f(n-1) when f is the factorial function. The lambda calculus approach may be viewed as a method of finding solutions to equations in which an identifier (the name of the recursive function) appears on both sides of the equation.

We can simplify the preceding equation by using lambda abstraction to eliminate n from the left-hand side. This gives us

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

Now consider the function G that we obtain by moving f to the right-hand-side of the equation:

```
G = λf. λn. if n=0 then 1 else n*f(n-1)
```

Although it might not be clear what sort of "algebra" is involved in this manipulation of equations, we can check, by using lambda calculus reasoning and basic understanding of function equality, that the factorial function f satisfies the equation

```
f = G(f)
```

This shows that recursive declarations involve finding fixed points.

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

A *fixed point* of a function G is a value f such that $f = G(f)$. In lambda calculus, fixed points may be defined with the *fixed-point operator*:

$$Y = \lambda f.(\lambda x.\, f(xx))(\lambda x.\, f(xx)).$$

The surprising fact about this perplexing lambda expression is that, for any f, the application Yf is a fixed point of f. We can see this by calculation. By β-equivalence, we have

$$Yf = (\lambda x.\, f(xx))(\lambda x.\, f(xx)).$$

Using β-equivalence again on the right-hand term, we get

$$Yf = (\lambda x.\, f(xx))(\lambda x.\, f(xx)) = f(\lambda x.\, f(xx))(\lambda x.\, f(xx)) = f(Yf).$$

Thus Yf is a fixed point of f.

Example 4.3
We can define factorial by $fact = YG$, where lambda terms Y and G are as given above, and calculate $fact\ 2 = 2!$ by using the calculation rules of lambda calculus. Here are the calculation steps representing the first "call" to factorial:

$$\begin{aligned}
&fact\, 2 = (YG)\, 2\\
&= G\,(YG)\, 2\\
&= (\lambda f.\lambda n.\, \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1))(YG)\, 2\\
&= (\lambda n.\, \text{if } n = 0 \text{ then } 1 \text{ else } n * ((YG)(n-1))\, 2\\
&= \text{if } 2 = 0 \text{ then } 1 \text{ else } 2 * ((YG)(2-1))\\
&= \text{if } 2 = 0 \text{ then } 1 \text{ else } 2 * ((YG)\, 1)\\
&= 2 * ((YG)\, 1).
\end{aligned}$$

Using similar steps, we can calculate $(YG)\ 1 = 1! = 1$ to complete the calculation.

It is worth mentioning that Y is not the only lambda term that finds fixed points of functions. There are other expressions that would work as well. However, this particular fixed-point operator played an important role in the history of lambda calculus.

4.2.4 Reduction, Confluence, and Normal Forms

The computational properties of lambda calculus are usually described with a form of symbolic evaluation called reduction. In simple terms, reduction is equational reasoning, but in a certain direction. Specifically, although β-equivalence was written as an equation, we have generally used it in one direction to "evaluate" function calls. If we write $\rightarrow$ instead of $=$ to indicate the direction in which we intend to use the equation, then we obtain the basic computation step called β-reduction:

$$(\lambda x.M)N \rightarrow [N/x]M, \qquad (\beta)$$

We say that M β-reduces to N, and write $M \rightarrow N$, if N is the result of applying one β-reduction step somewhere inside M. Most of the examples of calculation in this

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

section use β-reduction, i.e., β-equivalence is used from left to right rather than from right to left. For example,

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

Normal Forms

Intuitively, we think of $M \to N$ as meaning that, in one computation step, the expression M can be evaluated to the expression N. Generally, this process can be repeated, as illustrated in the preceding subsection. However, for many expressions, the process eventually reaches a stopping point. A stopping point, or expression that cannot be further evaluated, is called a *normal form*. Here is an example of a reduction sequence that leads to a normal form:

$$\begin{aligned}
&(\lambda f.\lambda x.\, f(fx))(\lambda y.y + 1)\,2\\
&\to (\lambda x.(\lambda y.y + 1)((\lambda y.y + 1)x))\,2\\
&\to (\lambda x.(\lambda y.y + 1)(x + 1))\,2\\
&\to (\lambda x.(x + 1 + 1))\,2\\
&\to (2 + 1 + 1).
\end{aligned}$$

This last expression is a normal form if our only computation rule is β-reduction, but not a normal form if we have a computation rule for addition. Assuming the usual evaluation rule for expressions with addition, we can continue with

$$\begin{aligned}
&2 + 1 + 1\\
&\to 3 + 1\\
&\to 4.
\end{aligned}$$

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

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

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 imortant 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

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

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 nonnegative 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.

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

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.

To pick an example, let us consider the mathematical interpretation of a simple algebraic expression such as $3 + 6 - 4$ that might appear in a program written in C, Java, or ML. The ordinary "mathematical" meaning of this expression is the number obtained by doing the addition and subtraction, namely 5. Here, the symbols in the expression $3 + 6 - 4$ are in our object language, whereas the number 5 is meant to be in our metalanguage. One way of making this clearer is to use an outlined number, such as $\mathbb{1}$, to mean "the mathematical entity called the natural number 1." Then we can say that the meaning of the object language expression $3 + 6 - 4$ is the natural number $\mathbb{5}$. In this sentence, the symbol $\mathbb{5}$ is a symbol of the metalanguage, whereas the expression $3 + 6 - 4$ is written with symbols of the object language.

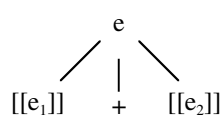
4.3.2 Denotational Semantics of Binary Numbers

The following grammar for binary expressions is similar to the grammar for decimal expressions discussed in Subsection 4.1.2:

```
e ::= n | e+e | e-e
n ::= b | nb
b ::= 0 | 1
```

We can give a mathematical interpretation of these expressions in the style of denotational semantics. In denotational semantics and other studies of programming languages, it is common to forget about how expressions are converted into parse trees and just give the meaning of an expression as a function of its parse tree.

We may interpret the expressions previously defined as natural numbers by using induction on the structure of parse trees. More specifically, we define a function from parse trees to natural numbers, defining the function on a compound expression by referring to its value on simpler expressions. A historical convention is to write $[[e]]$ for any parse tree of the expression e. When we write $[[e_1+e_2]]$, for example, we mean a parse tree of the form



with $[[e_1]]$ and $[[e_2]]$ as immediate subtrees.

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

Using this notation, we may define the meaning E[[e]] of an expression e, according to its parse tree [[e]], as follows:

$$
\begin{aligned}
&\mathtt{E[[0]]} = \mathbb{0}\\
&\mathtt{E[[1]]} = \mathbb{1}\\
&\mathtt{E[[nb]]} = \mathtt{E[[n]]} * \mathbb{2} + \mathtt{E[[b]]}\\
&\mathtt{E[[e_1{+}e_2]]} = \mathtt{E[[e_1]]} + \mathtt{E[[e_2]]}\\
&\mathtt{E[[e_1{-}e_2]]} = \mathtt{E[[e_1]]} - \mathtt{E[[e_2]]}
\end{aligned}
$$

In words, the value associated with a parse tree of the form $[[e_1{+}e_2]]$, for example, is the sum of the values given to the subtrees $[[e_1]]$ and $[[e_2]]$. This is not a circular definition because the parse trees $[[e_1]]$ and $[[e_2]]$ are smaller than the parse tree $[[e_1{+}e_2]]$.

On the right-hand side of the equal signs, numbers and arithmetic operations $*$, $+$, and $-$ are meant to indicate the actual natural numbers and the standard integer operations of multiplication, addition, and subtraction. In contrast, the symbols $+$ and $-$ in expressions surrounded by double square brackets on the left-hand side of the equal signs are symbols of the object language, the language of binary expressions.

4.3.3 Denotational Semantics of While Programs

Without going into detail about the kinds of mathematical functions that are used, let us take a quick look at the form of semantics used for a simplified programming language with assignment and loops.

Expressions with Variables

Program statements contain expressions with variables. Here is a grammar for arithmetic expressions with variables. This is the same as the grammar in Subsection 4.1.2, except that expressions can contain variables in addition to numbers:

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

In the simplified programming language we consider, the value of a variable depends on the state of the machine. We model the state of the machine by a function from variables to numbers and write E[[e]](s) for the value of expression e in state s. The value of an expression in a state is defined as follows.

$$
\begin{aligned}
&\mathtt{E[[x]]}(s) = s(x)\\
&\mathtt{E[[0]]}(s) = \mathbb{0}\\
&\mathtt{E[[1]]}(s) = \mathbb{1}\\
&\qquad \ldots = \ldots
\end{aligned}
$$

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

$$\begin{aligned}
&E[[9]](s) = 9\\
&E[[nd]](s) = E[[n]](s) * 10 + E[[d]](s)\\
&E[[e_1{+}e_2]](s) = E[[e_1]](s) + E[[e_2]](s)\\
&E[[e_1{-}\ e_2]](s) = E[[e_1]](s) - E[[e_2]](s)
\end{aligned}$$

Note that the state matters in the definition in the base case, the value of a variable. Otherwise, this is essentially the same definition as that in the semantics of variable-free expressions in the preceding subsection.

The syntax and semantics of Boolean expressions can be defined similarly.

While Programs

The language of *while* programs may be defined over any class of value expressions and Boolean expressions. Without specifying any particular basic expressions, we may summarize the structure of while programs by the grammar

```
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.

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

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

$$modify(s,x,a) = \lambda v \in Variables.\ \text{if } v{=}x \text{ then } a \text{ else } s(v)$$

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))
C[[if e then P_1 else P_2]](s) = if E[[e]](s) then C[[P_1]](s) else C[[P_2]](s)
C[[while e do P]](s) = if not E[[e]](s) then s
else C[[while e do P]](C[[P]](s))

Because e is an expression, not a statement, we apply the semantic function E to obtain the value of e in a given state.

In words, we can describe the semantics of programs as follows:

C[[x := e]](s) is the state similar to s, but with x having the value of e.
C[[P_1;P_2]](s) is the state we obtain by applying the semantics of P_2 the to the state we obtain by applying the semantics of P_1 to state s.
C[[if e then P_1 else P_2]](s) is the state we obtain by applying the semantics of P_1 to s if e is true in s and P_2 to s otherwise.
C[[while e do P]](s) is a recursively defined function f, from states to states. In words, $f(s)$ is either s if e is false or the state we obtain by applying f to the state resulting from executing P once in s.

The recursive function definition in the while clause is relatively subtle. It also raises some interesting mathematical problems, as it is not always mathematically reasonable to define functions by arbitrary recursive conditions. However, in the interest of keeping our discussion simple and straightforward, we just assume that a definition of this form can be made mathematically rigorous.

Example 4.4

We can calculate the semantics of various programs by using this definition. To begin with, let us consider a simple loop-free program,

```
if x>y then x :=y else y :=x
```

which sets both x and y to the minimum of their initial values. For concreteness, let us calculate the semantics of this program in the state s_0, where $s_0(x) = 1$ and $s_0(y)=2$.

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

Because E[[x>y]](s_0) = false, we have

$$\begin{aligned}&C[[\text{if x>y then x :=y else y := x }]](s_0)\\ &\quad = \text{if } E[[x>y]](s_0) \text{ then } C[[\text{ x := y }]](s_0) \text{ else } C[[\text{ y := x }]](s_0)\\ &\quad = C[[\text{ y := x }]](s_0)\\ &\quad = \text{modify}(s_0, y, E[[\text{ x }]](s_0))\end{aligned}$$

In words, if the program if x>y then x :=y else y := x is executed in the state s_0, then the result will be the state that is the same as s_0, but with variable y given the value that x has in state s_0.

Example 4.5

Although it takes a few more steps than the previous example, it is not too difficult to work out the semantics of the program

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

in the state s_0, where s_0(z)=2. A few preliminary definitions will make the calculation easier. Let s_1 and s_2 be the states

$$\begin{aligned}s_1 &= C[[\text{ x:= 0 }]](s_0)\\ s_2 &= C[[\text{ y := 0 }]](s_1)\end{aligned}$$

Using the semantics of assignment, as above, we have

$$\begin{aligned}s_1 &= \text{modify}(s_0 , x, 0)\\ s_2 &= \text{modify}(s_1 , y, 0)\end{aligned}$$

Returning to the preceding program, we have

$$\begin{aligned}&C[[\text{x := 0; y:=0; while x ≤ z do (y := y+x; x := x+1) }]](s_0)\\ &= C[[\text{ y:=0; while x ≤ z do (y := y+x; x := x+1) }]](\text{ C [[x := 0 }]](s_0))\\ &= C[[\text{ y:=0; while x ≤ z do (y := y+x; x := x+1) }]](s_1)\\ &= C[[\text{ while x ≤ z do (y := y+x; x := x+1) }]](\text{ C [[y := 0 }]](s_1))\\ &= C[[\text{ while x ≤ z do (y := y+x; x := x+1) }]](s_2)\\ &= \text{if not } E[[\text{ x ≤ z }]](s_2) \text{ then } s_2\\ &\quad \text{else } C[[\text{ while x ≤ z do (y := y+x; x := x+1) }]](C[[\text{ y := y+x; x := x+1 }]](s_2))\\ &= C[[\text{ while x ≤ z do (y := y+x; x := x+1) }]](s_3)\end{aligned}$$

where s_3 has y set to 0 and x set to 1. Continuing in the same manner, we have

$$\begin{aligned}&C[[\text{ while x ≤ z do (y := y+x; x := x+1) }]](s_3)\\ &= \text{if not } E[[\text{x ≤ z }]](s_3) \text{ then } s_3\\ &\quad \text{else } C[[\text{ while x ≤ z do (y := y+x; x := x+1) }]](C[[\text{y := y+x; x := x+1 }]](s_3))\\ &= C[[\text{ while x ≤ z do (y := y+x; x := x+1) }]](s_4)\\ &= \text{if not } E[[\text{x ≤ z }]](s_4) \text{ then } s_4\\ &\quad \text{else } C[[\text{ while x ≤ z do (y := y+x; x := x+1) }]](C[[\text{y := y+x; x := x+1 }]](s_4))\\ &= C[[\text{ while x ≤ z do (y := y+x; x := x+1) }]](s_5)\\ &= s_5\end{aligned}$$

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

where s_4 has y set to 1 and x to 2 and s_5 has y set to 3 and x to 3. The steps are tedious to write out, but you can probably see without doing so that if $s_0(z) = 5$, for example, then this program will yield a state in which the value of x is 0+1+2+3+4+5.

As these examples illustrate, the denotational semantics of while programs unambiguously associates a partial function from states to states with each program. One important issue we have not discussed in detail is what happens when a loop does not terminate. The meaning C[[while x=x do x := x]] of a loop that does not terminate in any state is a partial function that is not defined on any state. In other words, for any state s, C[[while x=x do x := x]](s) is not defined. Similarly, C[[while x=y do x := y]](s) is s if $s(x) \neq s(y)$ 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.

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

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

$$State = \{\text{error}\} \cup (Variables \to \{init, uninit\})$$

As usual, the meaning of a program will be a function from states to states. Let us assume that `E[[e]](s)` = *error* if `e` contains any variable `y` with `s(y)` = *uninit* and `E[[e]](s)` = *Ok* otherwise.

The semantics of programs is given by a set of clauses, one for each program form, as usual. For any program `P`, we let `C[[P]](error)` = *error*. The semantic clause for assignment in state `s` ≠ *error* can be written as

$$\text{C[[x :=e]](s)} = \text{if E[[e]](s)}{=}Ok \text{ then modify(s, x, } init\text{) else } error$$

In words, if we execute an assignment `x:= e` in a state `s` different from *error*, then the result is either a state that has `x` initialized or *error* if there is some variable in `e` that was not initialized in `s`. For example, let

$$s_0 = \lambda v \in Variables.\ uninit$$

be the state with every variable uninitialized. Then

$$\text{C[[x := 0]]}(s_0) = \text{modify}(s_0, \text{x}, init)$$

is the state with variable `x` initialized and all other variables uninitialized.

The clauses for sequences `P1; P2` are essentially straightforward. For $s \neq error$,

$$\text{C[[}P_1;P_2\text{]](s)} = \text{if C[[}P_1\text{]](}s\text{)} = error \text{ then } error \text{ else C[[} P_2 \text{]](C[[}P_1\text{]](}s\text{))}$$

The clause for conditional is more complicated because our analysis tool is not going to try to evaluate a Boolean expression. Instead, we treat conditional as if it were

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

possible to execute either branch. (This is the conservative part of the analysis.) Therefore, we change a variable to initialized only if it is initialized in both branches. If s_1 and s_2 are states different from *error*, then let $s_1 + s_2$ be the state

$$s_1 + s_2 = \lambda v \in \mathit{Variables}.\ \text{If } s_1(v) = s_2(v) = \mathit{init} \text{ then } \mathit{init} \text{ else } \mathit{uninit}$$

We define the meaning of a conditional statement by

$$\begin{aligned} &C[[\texttt{if e then } P_1 \texttt{ else } P_2]](s) \\ &= \text{if } E[[\texttt{ e }]](s) = \mathit{error} \text{ or } C[[\texttt{ } P_1]](s) = \mathit{error} \text{ or } C[[\texttt{ } P_2\texttt{ }]](s) = \mathit{error} \\ &\qquad \text{then } \mathit{error} \\ &\qquad \text{else } C[[\texttt{ } P_1]](s) + C[[\texttt{ } P_2\texttt{ }]](s) \end{aligned}$$

For example, using s_0 as above, we have

$$C[[\texttt{ if 0=1 then x := 0 else x := 1; y := 2 }]](s_0) = \text{modify}(s_0, \texttt{x}, \mathit{init})$$

as only x is initialized in both branches of the conditional.

For simplicity, we do not consider the clause for while e do P.

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 value 5 in a certain location. Programming languages also contain declarative constructs such as the function declaration

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

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

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.

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

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 counterexample 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

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

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 Cumputing 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 run-time efficiency. Because of his belief in the importance of correctness, readability, and reliability, Backus thought that pure functional languages would be appreciated as superior to languages with side effects.

To advance his argument, Backus proposed a pure functional programming language called FP, an acronym for functional programming. FP contains a number of basic functions and a rich set of combining forms with which to build new functions from old ones. An example from Backus' paper is a simple program to compute the inner product of two vectors. In C, the inner product of vectors stored in arrays a and b could be written as

```
int i, prod;
prod=0;
for (i=0; i < n; i++) prod = prod + a[i] * b[i];
```

In contrast, the inner product function would be defined in FP by combining functions + and × (multiplication) with vector operations. Specifically, the inner product would be expressed as

```
Inner_product = (Insert +) ∘ (ApplyToAll x) ∘ Transpose
```

where ∘ is function composition and Insert, ApplyToAll, and Transpose are vector

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

operations. Specifically, the Transpose of a pair of lists produces a list of pairs and ApplyToAll applies the given operation to every element in a list, like the maplist function in Subsection 3.4.7. Given a binary operation and a list, Insert has the effect of inserting the operation between every pair of adjacent list elements and calculating the result. For example, (Insert +) <1, 2, 3, 4> = 1+2+3+4=10, where <1, 2, 3, 4> is the FP notation for the list with elements 1, 2, 3, 4.

Although the C syntax may seem clearer to most readers, it is worth trying to imagine how these would compare if you had not seen either before. One facet of the FP expression is that all of its parts are functions that we can understand without thinking about how vectors are represented in memory. In contrast, the C program has extra variables i and prod that are not part of the function we are trying to compute. In this sense, the FP program is higher level, or more abstract, than the C code.

A more general point about FP programs is that if one functional expression is equivalent to another, then we can replace one with the other in any program. This leads to a set of algebraic laws for FP programs. In addition to algebraic laws, an advantage of functional programming languages is the possibility of parallelism in implementations. This is subsequently discussed.

In retrospect, Backus' argument seems more plausible than his solution. The importance of program correctness, readability, and reliability has increased compared with that of run-time efficiency. The reason is that these affect the amount of time that people must spend in developing, debugging, and maintaining code. When computers become faster, it is acceptable to run less efficient programs – hardware improvements can compensate for software. However, increases in computer speed do not make humans more efficient. In this regard, Backus was absolutely right about the aspects of programming languages that would be important in the future.

Backus' language FP, on the other hand, was not a success. In the years since his lecture, there was an effort to develop a FP implementation at IBM, but the language was not widely used. One problem is that the language has a difficult syntax. Perhaps more importantly, there are severe limitations in the kind of data structures and control 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 $f(e_1,\ldots,e_n)$, where function arguments $e_1,\ldots,e_n$ are expressions that may need to be evaluated.

Functional Programming: We can evaluate $f(e_1,\ldots,e_n)$ by evaluating $e_1,\ldots,e_n$ 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

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

> languages must be evaluated in a fixed, sequential order. This ordering 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 definitive answer. From one theoretical point of view, FP languages are as good as imperative programming languages. This can be demonstrated when a translation is made of C programs into FP programs, lambda calculus expressions, or pure Lisp. Denotational semantics provides one method for doing this.

To answer the question in practice, however, we would need to carry out large projects in a functional language and see whether it is possible to produce usable software in a reasonable amount of time. Some work toward answering this question was done at IBM on the FP project (which was canceled soon after Backus retired). Additional efforts with other languages such as Haskell and Lazy ML are still being carried out at other research laboratories and universities. Although most programming is done in imperative languages, it is certainly possible that, at some

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

future time, pure or mostly pure FP languages will become more popular. Whether or not that happens, FP projects have generated many interesting language design ideas and implementation techniques that have been influential beyond pure functional programming.

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

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

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.

EXERCISES

4.1 Parse Tree

Draw the parse tree for the derivation of the expression 25 given in Subsection 4.1.2. Is there another derivation for 25? Is there another parse tree?

4.2 Parsing and Precedence

Draw parse tress for the following expressions, assuming the grammar and precedence described in Example 4.2:

(a) `1 - 1 * 1`.

(b) `1 - 1 + 1`.

(c) `1 - 1 + 1 - 1 + 1`, if we give + higher precedence than −.

4.3 Lambda Calculus Reduction

Use lambda calculus reduction to find a shorter expression for $(\lambda x.\lambda y.xy)(\lambda x.xy)$. Begin by renaming bound variables. You should do all possible reductions to get the shortest possible expression. What goes wrong if you do not rename bound variables?

4.4 Symbolic Evaluation

The Algol-like program fragment

```
function f(x)
    return x+4
end;
function g(y)
    return 3-y
end;
f(g(1));
```

can be written as the following lambda expression:

$$\left(\underbrace{(\lambda f.\lambda g.\, f(g\ 1))}_{\text{main}}\ \underbrace{(\lambda x.x+4)}_{f} \right)\ \underbrace{(\lambda y.3-y)}_{g}.$$

Reduce the expression to a normal form in two different ways, as described below.

(a) Reduce the expression by choosing, at each step, the reduction that eliminates a λ as far to the *left* as possible.

(b) Reduce the expression by choosing, at each step, the reduction that eliminates a λ as far to the *right* as possible.

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

4.5 Lambda Reduction with Sugar

Here is a "sugared" lambda expression that uses let declarations:

$$\begin{array}{l}\text{let } \mathit{compose} = \lambda f.\, \lambda g.\, \lambda x.\ f(g\, x) \text{ in}\\ \quad \text{let } h = \lambda x.\, x + x \text{ in}\\ \quad\quad \mathit{compose}\, h\, h\, 3\end{array}$$

The "desugared" lambda expression, obtained when each let $z = U$ in V is replaced with $(\lambda z.\, V)\, U$ is

$$\begin{array}{l}(\lambda \mathit{compose}.\\ \quad (\lambda h.\ \mathit{compose}\ h\ h\ 3)\ \ \lambda x.\, x + x)\\ \quad \lambda f.\, \lambda g.\, \lambda x.\, f(g\ x).\end{array}$$

This is written with the same variable names as those of the let form to make it easier to read the expression.

Simplify the desugared lambda expression by using reduction. Write one or two sentences explaining why the simplified expression is the answer you expected.

4.6 Translation into Lambda Calculus

A programmer is having difficulty debugging the following C program. In theory, on an "ideal" machine with infinite memory, this program would run forever. (In practice, this program crashes because it runs out of memory, as extra space is required every time a function call is made.)

```
int f(int (*g)(...)){ /* g points to a function that returns an int */
   return g(g);
}
int main(){
  int x;
  x = f(f);
  printf("Value of x = %d\n", x);
  return 0;
}
```

Explain the behavior of the program by translating the definition of f into lambda calculus and then reducing the application f(f). This program assumes that the type checker does not check the types of arguments to functions.

4.7 Order of Evaluation

In pure lambda calculus, the order of evaluation of subexpressions does not effect the value of an expression. The same is true for pure Lisp: if a pure Lisp expression has a value under the ordinary Lisp interpreter, then changing the order of evaluation of subterms cannot produce a different value.

To give a concrete example, consider the following section of Lisp code:

```
(define a ( . . . ))
(define b ( . . . ))
. . .
(define f (lambda (x y z) (cons (car x) (cons (car y) (cdr z)))))
. . .
(f e1 e2 e3)
```

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

The ordinary evaluation order for the function call

```
(f e1 e2 e3)
```

is to evaluate the arguments `e1 e2 e3` from left to right and then pass this list of values to the function f.

Give an example of Lisp expressions `e1 e2 e3`, possibly by using functions `rplaca` or `rplacd` with side effects, so that evaluating expressions from left to right gives a different result from that obtained by evaluating them from right to left. You may fill in specific declarations for `a` or `b` if you like and refer to `a` or `b` in your expressions. Explain briefly, in one or two sentences, why one order of evaluation is different from the other.

4.8 Denotational Semantics

The text describes a denotational semantics for the simple imperative language given by the grammar

$$P ::= x := e \mid P_1; P_2 \mid \text{if } e \text{ then } P_1 \text{ else } P_2 \mid \text{while } e \text{ do } P.$$

Each program denotes a function from *states* to *states*, in which a *state* is a function from *variables* to *values*.

(a) Calculate the meaning $\mathcal{C}[\![x := 1; x := x + 1;]\!](s_0)$ in approximately the same detail as that of the examples given in the text, where $s_0 = \lambda v \in \mathit{variables}.\, 0$, giving every variable the value 0.

(b) Denotational semantics is sometimes used to justify ways of reasoning about programs. Write a few sentences, referring to your calculation in part (a), explaining why

$$\mathcal{C}[\![x := 1; x := x + 1;]\!](s) = \mathcal{C}[\![x := 2;]\!](s)$$

for every state s.

4.9 Semantics of Initialize-Before-Use

A nonstandard denotational semantics describing initialize-before-use analysis is presented in the text.

(a) What is the meaning of

$$\mathcal{C}[\![x := 0; y := 0; \text{if } x = y \text{ then } z := 0 \text{ else } w := 1\ ]\!](s_0)$$

in the state $s_0 = \lambda y \in \mathit{variables}.\mathit{uninit}$? Show how to calculate your answer.

(b) Calculate the meaning

$$\mathcal{C}[\![\text{if } x = y \text{ then } z := y \text{ else } z := w]\!](s)$$

in state s with $s(x) = \mathit{init}$, $s(y) = \mathit{init}$, and $s(v) = \mathit{uninit}$ or every other variable v.

4.10 Semantics of Type Checking

This problem asks about a nonstandard semantics that characterizes a form of type analysis for expressions given by the following grammar:

$$e ::= 0 \mid 1 \mid \text{true} \mid \text{false} \mid x \mid e + e \mid \text{if } e \text{ then } e \text{ else } e \mid \text{let } x{:}\tau{=}e \text{ in } e$$

In the let expresion, which is a form of local declaration, the type τ may be either *int* or *bool*. The meaning $\mathcal{V}[\![e]\!](\eta)$ of an expression e depends on an environment.

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

An environment η is a mapping from variables to values. In type analysis, we use three values,

$$Values = \{integer, boolean, type\_error\}.$$

Intuitively, $\mathcal{V}[\![e]\!](\eta) = integer$ means that the value of expression e is an integer (in environment η) and $\mathcal{V}[\![e]\!](\eta) = type\_error$ means that evaluation of e *may* involve a type error.

Here are the semantic clauses for the first few expression forms.

$$\begin{aligned}\mathcal{V}[\![0]\!]\eta &= integer,\\ \mathcal{V}[\![1]\!]\eta &= integer,\\ \mathcal{V}[\![\text{true}]\!]\eta &= boolean,\\ \mathcal{V}[\![\text{false}]\!]\eta &= boolean.\end{aligned}$$

The value of a variable in some environment is simply the value the environment gives to that variable:

$$\mathcal{V}[\![x]\!]\eta = \eta(x)$$

For addition, the value will be *type_error* unless both operands are integers:

$$\mathcal{V}[\![e_1 + e_2]\!]\eta = \begin{cases} integer & \text{if } \mathcal{V}[\![e_1]\!]\eta = integer \text{ and } \mathcal{V}[\![e_2]\!]\eta = integer \\ type\_error & \text{otherwise} \end{cases}.$$

Because declarations involve setting the types of variables, and the types of variables are recorded in an environment, the semantics of declarations involves changing the environment. Before giving the clause for declarations, we define the notation $\eta[x \mapsto \sigma]$ for an environment that is similar to η, except that it must map the variable x to type σ. More precisely,

$$\eta[x \mapsto \sigma] = \lambda y \in Variables.\ \text{if } y = x \text{ then } \sigma \text{ else } \eta(y).$$

Using this notation, we can define the semantics of declarations by

$$\begin{aligned}&\mathcal{V}[\![\text{let } x{:}\tau = e_1 \text{ in } e_2]\!]\eta\\ &\quad= \begin{cases} \mathcal{V}[\![e_2]\!](\eta[x \mapsto integer]) & \text{if } \mathcal{V}[\![e_1]\!]\eta = integer \text{ and } \tau = int \\ \mathcal{V}[\![e_2]\!](\eta[x \mapsto boolean]) & \text{if } \mathcal{V}[\![e_1]\!]\eta = boolean \text{ and } \tau = bool. \\ type\_error & \text{otherwise} \end{cases}\end{aligned}$$

The clause for conditional is

$$\begin{aligned}&\mathcal{V}[\![if\ e_1\ then\ e_2\ else\ e_3]\!]\eta\\ &\quad= \begin{cases} boolean & \text{if } \mathcal{V}[\![e_1]\!]\eta = \mathcal{V}[\![e_2]\!]\eta = \mathcal{V}[\![e_3]\!]\eta = boolean \\ integer & \text{if } \mathcal{V}[\![e_1]\!]\eta = boolean \text{ and } \mathcal{V}[\![e_2]\!]\eta = \mathcal{V}[\![e_3]\!]\eta = integer. \\ type\_error & \text{otherwise} \end{cases}\end{aligned}$$

For example, with $\eta_0 = \lambda y \in Var.\ type\_error$,

$$\mathcal{V}[\![\text{if true then } 0+1 \text{ else } x+1]\!]\eta_0 = type\_error$$

as the expression $x + 1$ produces a type error if evaluating x produces a type error. On the other hand,

$$\mathcal{V}[\![\text{let } x{:}int = (1{+}1) \text{ in (if true then 0 else } x)]\!]\eta_0 = integer$$

as the let expression declares that x is an integer variable.

Questions:

(a) Show how to calculate the meaning of the expression if false then 0 else 1 in the environment $\eta_0 = \lambda y \in Var.\ type\_error$.

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

(b) Suppose e_1 and e_2 are expressions with $\mathcal{V}[\![e_1]\!]\eta = integer$ and $\mathcal{V}[\![e_2]\!]\eta = boolean$ in every environment. Show how to calculate the meaning of the expression let x:*int*=e_1 in (if e_2 then e_1 else x) in environment $\eta_0 = \lambda y \in Var.\, type\_error$.

(c) In declaration let $x{:}\tau{=}e$ in e, the type of x is given explicitly. It is also possible to leave the type out of the declaration and infer it from context. Write a semantic clause for the alternative form, let $x{=}e_1$ in e_2 by using the following partial solution (i.e., fill in the missing parts of this definition):

$$\mathcal{V}[\![let\ x{=}e_1\ in\ e_2]\!]\eta = \begin{cases} \mathcal{V}[\![e_2]\!](\eta[x \mapsto \sigma]) & \text{if} \\ type\_error & \text{otherwise} \end{cases}.$$

4.11 Lazy Evaluation and Parallelism

In a "lazy" language, we evaluate a function call f(e) by passing the *unevaluated* argument to the function body. If the value of the argument is needed, then it is evaluated as part of the evaluation of the body of f. For example, consider the function g defined by

```
fun g(x,y) = if = x=0
                then 1
                else if x+y=0
                        then 2
                        else 3;
```

In a lazy language, we evaluate the call g(3,4+2) by passing some representation of the expressions 3 and 4+2 to g. We evaluate the test x=0 by using the argument 3. If it were true, the function would return 1 without ever computing 4+2. Because the test is false, the function must evaluate x+y, which now causes the actual parameter 4+2 to be evaluated. Some examples of lazy functional languages are Miranda, Haskell, and Lazy ML; these languages do not have assignment or other imperative features with side effects.

If we are working in a pure functional language without side effects, then for any function call f(e_1, e_2), we can evaluate e_1 before e_2 or e_2 before e_1. Because neither can have side effects, neither can affect the value of the other. However, if the language is lazy, we might not need to evaluate both of these expression. Therefore, something can go wrong if we evaluate both expressions and one of them does not terminate.

As Backus argues in his Turing Award lecture, an advantage of pure functional languages is the possibility of parallel evaluation. For example, in evaluating a function call f(e_1, e_2) we can evaluate both e_1 and e_2 in parallel. In fact, we could even start evaluating the body of f in parallel as well.

(a) Assume we evaluate g(e_1, e_2) by starting to evaluate g, e_1, and e_2 in parallel, where g is the function defined above. Is it possible that one process will have to wait for another to complete? How can this happen?

(b) Now, suppose the value of e_1 is zero and evaluation of e_2 terminates with an error. In the normal (i.e., eager) evaluation order that is used in C and other common languages, evaluation of the expression g(e_1, e_2) will terminate in error. What will happen with lazy evaluation? Parallel evaluation?

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

(c) Suppose you want the same value, for every expression, as lazy evaluation, but you want to evaluate expressions in parallel to take advantage of your new pocket-sized multiprocessor. What actions should happen, if you evaluate $g(e_1, e_2)$ by starting g, e_1, and e_2 in parallel, if the value of e_1 is zero and evaluation of e_2 terminates in an error?

(d) Suppose now that the language contains side effects. What if e_1 is z and e_2 contains an assignment to z; can you still evaluate the arguments of $g(e_1, e_2)$ in parallel? How? Or why not?

4.12 Single-Assignment Languages

A number of so-called *single-assignment languages* have been developed over the years, many designed for parallel scientific computing. Single-assignment conditions are also used in program optimization and in hardware description languages. Single-assignment conditions arise in hardware as only one assignment to each variable may occur per clock cycle.

One example of a single-assignment language is *SISAL,* which stands for streams and iteration in a single-assignment language. Another is *SAC,* or single-assignment C. Programs in single-assignment languages must satisfy the following condition:

> *Single-Assignment Condition:* During any run of the program, each variable may be assigned a value only once, within the scope of the variable.

The following program fragment satisfies this condition,

```
if (y>3) then x = 42+29/3 else x = 13.39;
```

because only one branch of the if-then-else will be executed on any run of the program. The program `x=2; loop_forever; x=3` also satisfies the condition because no execution will complete both assignments.

Single-assignment languages often have specialized loop constructs, as otherwise it would be impossible to execute an assignment inside a loop body that gets executed more than once. Here is one form, from SISAL:

```
for ⟨range⟩
   ⟨body⟩
returns ⟨returns clause⟩
end for
```

An example illustrating this form is the following loop, which computes the dot (or inner) product of two vectors:

```
for i in 1, size
   elt_prod := x[i]*y[i]
returns value of sum elt_prod
end for
```

This loop is parallelizable because different products `x[i]*y[i]` can be computed in parallel. A typical SISAL program has a sequential outer loop containing a set of parallel loops.

Suppose you have the job of building a parallelizing compiler for a single-assignment language. Assume that the programs you compile satisfy the single-assignment condition and do not contain any explicit process fork or other parallelizing instructions. Your implementation must find parts of programs that can

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

be safely executed in parallel, producing the same output values as if the program were executed sequentially on a single processor.

Assume for simplicity that every variable is assigned a value before the value of the variable is used an in expression. Also assume that there is no potential source of side effects in the language other than assignment.

(a) Explain how you might execute parts of the sample program

```
x = 5;
y = f(g(x),h(x));
if y==5 then z=g(x) else z=h(x);
```

in parallel. More specifically, assume that your implementation will schedule the following processes in some way:

```
process 1 - set x to 5
process 2 - call g(x)
process 3 - call h(x)
process 4 - call f(g(x),h(x)) and set y to this value
process 5 - test y==5
process 6 - call g(x) and then set z=g(x)
process 7 - call h(x) and then set z=h(x)
```

For each process, list the processes that this process must wait for and list the processes that can be executed in parallel with it. For simplicity, assume that a call cannot be executed until the parameters have been evaluated and assume that processes 6 and 7 are *not* divided into smaller processes that execute the calls but do not assign to z. Assume that parameter passing in the example code is by value.

(b) If you further divide process 6 into two processes, one that calls g(x) and one that assigns to z, and similarly divide process 7 into two processes, can you execute the calls g(x) and h(x) in parallel? Could your compiler correctly eliminate these calls from processes 6 and 7? Explain briefly.

(c) Would the parallel execution of processes you describe in parts (a) and (b), if any, be correct if the program does not satisfy the single-assignment condition? Explain briefly.

(d) Is the single-assignment condition decidable? Specifically, given an program written in a subset of C, for concreteness, is it possible for a compiler to decide whether this program satisfies the single-assignment condition? Explain why or why not. If not, can you think of a decidable condition that implies the single-assignment condition and allows many useful single-assignment programs to be recognized?

(e) Suppose a single-assignment language has no side-effect operations other than assignment. Does this language pass the declarative language test? Explain why or why not.

4.13 Functional and Imperative Programs

Many more lines of code are written in imperative languages than in functional ones. This question asks you to speculate about reasons for this. First, however, an explanation of what the reasons *are not* is given:

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press

- It is not because imperative languages can express programs that are not possible in functional languages. Both classes can be made Turing complete, and indeed a denotational sematics of an imperative language can be used to translate it into a functional language.
- It is not because of syntax. There is no reason why a functional language could not have a syntax similar to that of C, for example.
- It is not because imperative languages are always compiled whereas functional languages are always interpreted. Basic is imperative, but is usually interpreted, whereas Haskell is functional and usually compiled.

For this problem, consider general properties of imperative and functional languages. Assume that a functional language supports higher-order functions and garbage collection, but not assignment. For the purpose of this question, an imperative language is a language that supports assignment, but not higher-order functions or garbage collection. Use only these assumptions about imperative and functional languages in your answer.

(a) Are there inherent reasons why imperative languages are superior to functional ones for the majority of programming tasks? Why?

(b) Which variety (imperative or functional) is easier to implement on machines with limited disk and memory sizes? Why?

(c) Which variety (imperative or functional) would require bigger executables when compiled? Why?

(d) What consequence might these facts have had in the early days of computing?

(e) Are these concerns still valid today?

4.14 Functional Languages and Concurrency

It can be difficult to write programs that run on several processors concurrently because a task must be decomposed into independent subtasks and the results of subtasks often must be combined at certain points in the computation. Over the past 20 years, many researchers have tried to develop programming languages that would make it easier to write concurrent programs. In his Turing Lecture, Backus advocated functional programming because he believed functional programs would be easier to reason about and because he believed that functional programs could be executed efficiently in parallel.

Explain why functional programming languages do not provide a complete solution to the problem of writing programs that can be executed efficiently in parallel. Include two specific reasons in your answer.

https://doi.org/10.1017/CBO9780511804175.005 Published online by Cambridge University Press