Chapter 4. **Fundamentals**

This chapter provides some background 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 will compare functional and imperative languages in the last section of this chapter.

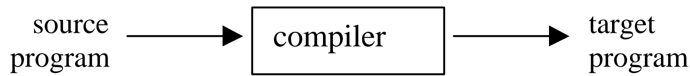
4.1 Compilers and Syntax

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

4.1.1 Structure of a simple compiler

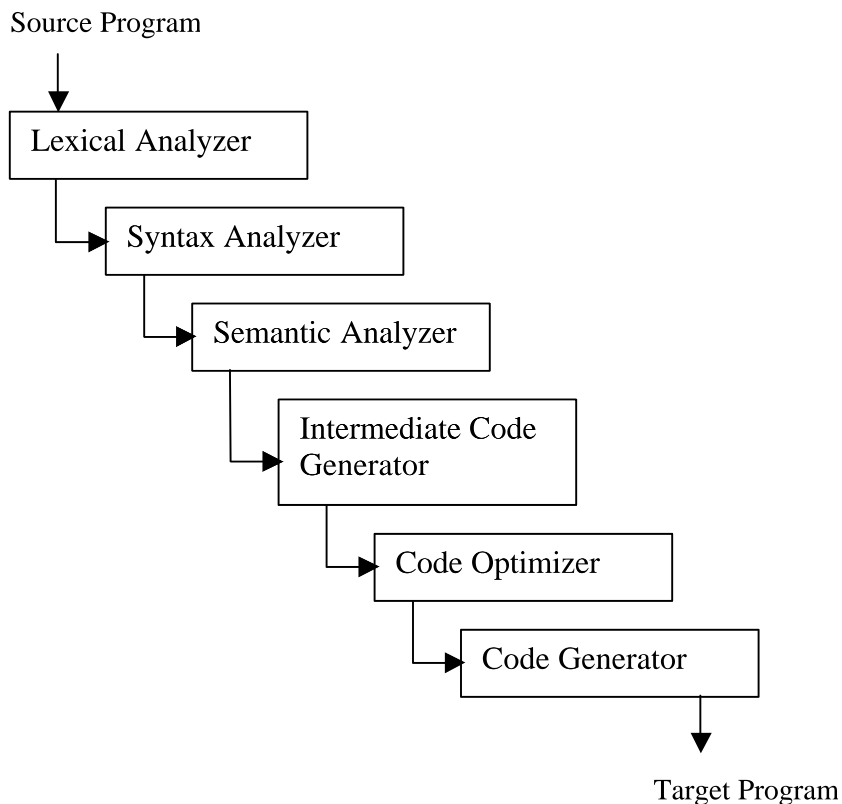
Programming languages that are convenient for people to use are built around concepts and abstractions that may not correspond directly to features of the underlying machine. For this reason, a program must be translated into the basic instruction set of the machine before it can be executed. This can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

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, 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:



We will discuss each of these phases briefly. Our goal in 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 will 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 University 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. But since 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. Since variables and constants are declared differently, variables and constants are distinguished in the semantic analysis phase.

Syntax Analysis

In this phase, tokens are grouped into syntactic units such as expressions, statements, and declarations that must conform to the grammatical rules of the programming language. The action performed during this phase, called *parsing,* is described in Section 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, it is necessary to make sure that the types match. If this assignment occurs in a language where 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, + 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 float and + marked as floating-point addition. Depending on whether temp is float or 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 like 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, we will use the word semantics 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.

Since the last phase of the compiler can translate one set of instructions to another, the intermediate code does not need to be written using 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. By using a single generic intermediate representation, it is possible to use essentially the same compiler to generate target programs for several different machines.

Code Optimization

There are a variety of techniques that may be used to improve the efficiency of a program. These 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.

Some standard optimizations are:

- *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 like x=y, then it may be possible to change subsequent statements to refer to y instead of x and 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, since the instructions associated with calling a function can be eliminated, but increases the size of the program. The most important consequence of in-lining function calls is usually that it allows 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 target machine code. This involves choosing a memory location and/or register 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 since 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 will use grammars to describe various languages in this book. Although we will not usually be too concerned about the pragmatics of parsing, we will take a brief look in this section at the problem of producing a parse tree from a sequence of tokens.

Grammars

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

A *grammar* consists of a start symbol, a set of nonterminals, a set of terminals, and a set of productions. The nonterminals are symbols that are used to write out the grammar, while 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 → tu, with an arrow, meaning that in a string containing the symbol s, we can replace s by the symbols tu. However, we will use a more compact notation commonly referred to as *BNF*.

We will illustrate the main ideas 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, +, - are the terminals. The language defined by this grammar consists of all the sequences of terminals that can be produced by starting with the start symbol, e, and replacing nonterminals according to the productions above. For example, the first production above means that we can replace an occurrence of e by either the symbol n, the three symbols e+e, or the three symbols e-e. The process can be repeated, using any of the three lines above.

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 by terminals in order to produce a well-formed expression of the language.

Derivations

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

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

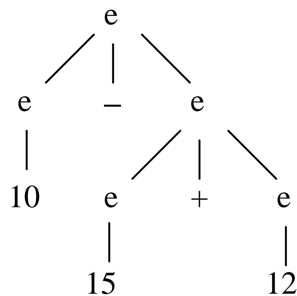
$e \rightarrow n \rightarrow nd \rightarrow dd \rightarrow 2d \rightarrow 25$

$e \rightarrow e - e \rightarrow e - e{+}e \rightarrow \ldots \rightarrow n{-}n{+}n \rightarrow \ldots \rightarrow 10{-}15{+}12$

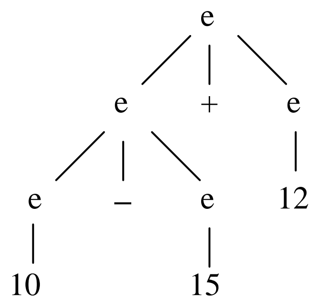
Parse Trees and Ambiguity

It is often convenient to represent a derivation by a tree. This tree, called the *parse tree* of a derivation, or *derivation tree*, is constructed using the start symbol as the root of the tree. If a step in the derivation is to replace *s* by $x_1, \ldots x_n$, 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 above has some useful structure. Specifically, since 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 above corresponds to 10– (15+12) while 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 using the following simple grammar:

s ::= v := e | s;s | if b then s | if b then s else s

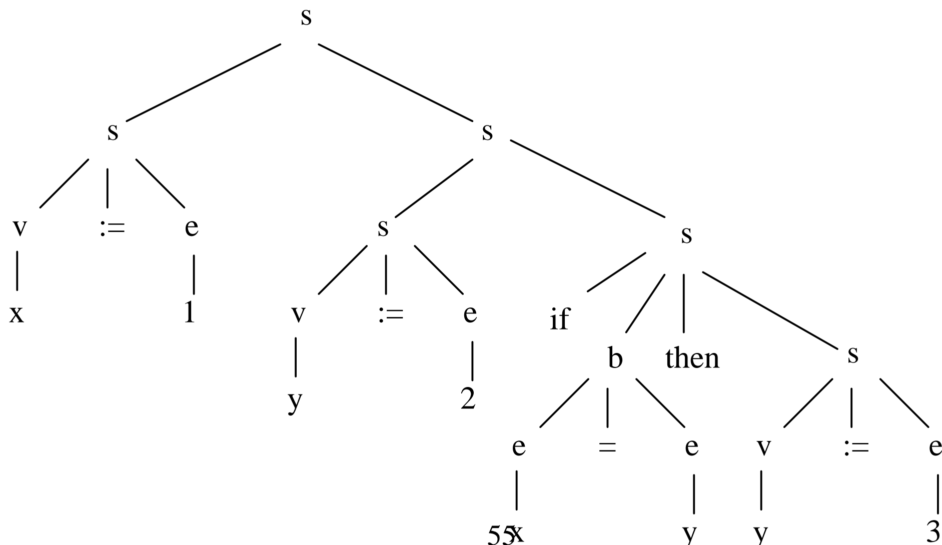
v ::= x | y | z

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”. We will call the expressions of the language generated by this 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, obtained 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 b_1 then if b_2 then s_1 else s_2

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 only for particular forms of grammars. Since parsing is an important 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, since 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 (–). Precedence is incorporated into parsing by treating an unparenthesized expression e op_1 e op_2 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 (5*4) – 3. This coincides with the way that most of us would ordinarily think about the expression 5*4 – 3. Since 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 using an additional convention.

Associativity comes into play when two operators of equal precedence appear next to each other. Under *left-associativity*, an expression e op_1 e op_2 e would be parsed as (e op_1 e) op_2 e, if the two operators have equal precedence. If we adopted a right-associativity convention instead, e op_1 e op_2 e would be parsed as e op_1 (e op_2 e).

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 (via function expressions) and binding (via 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 see expressions in your favorite programming language as essentially

a form of lambda expression. For simplicity, we will discuss the untyped lambda calculus; 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 Section 2.1.2 for a discussion of functions in mathematics.) Some examples of functions studied in mathematics are

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

and

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

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 like

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

since 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 and exponentiation to pure lambda calculus. While purists stick to pure lambda calculus without addition and multiplication, we will use these operations in examples since 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, which 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 like this. Another example 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 will 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

Programming language	=	applied λ-calculus
	=	pure λ-calculus + additional data types

This even works for programming languages with side-effects, since the way a program depends on the state of the machine can be represented using 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.

4.2.2 Lambda expressions

Syntax of expressions

The syntax of untyped lambda expressions may be defined using 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_1M_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_1M_2 is the application of function M_1 to argument M_2; and $\lambda x.M$ is the function which, 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*.
$\lambda x.(f\ (g\ x))$	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 confusing to learn. These can generally be avoided 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 $\lambda x.(x\ y)$, 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*, since 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 $\int f(x)\, dx$ 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 only in 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 free otherwise. To be more precise about this, we may define the set $FV(M)$ of *free variables of M* by induction on the structure of expressions, as follows:

$$FV(x) = \{\ x\ \}$$

$$FV(MN) = FV(M) \cup FV(N)$$

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

where − here means set difference, that is, $S-x = \{y \in S \mid y \neq x\}$. For example, $FV(\lambda x.x) = \varnothing$ since there are no free variables in $\lambda x.x$, while $FV(\lambda f.\ \lambda x.\ (f\,(g\,(x)))) = \{g\}$.

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,* since this is where x becomes bound. The other occurrence of x is a bound occurrence. Reading from left to right, the first occurrence of y is a binding occurrence, the second is a bound occurrence, and the third is free since 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

$$\lambda x.\ \textit{function_body}$$

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, while functions are the only data type in pure lambda calculus. Another difference is evaluation order, but we will not go into that topic in detail.

Although the syntax of block-structured languages is farther from lambda calculus than 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;
```

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

$$\text{let } f = (\lambda x.\ x) \text{ in } \textit{block_body}$$

Notice that the C form expression and the lambda expression have the same free and bound variables, and similar structure. One difference between C and λ-calculus is that C has assignment statements and side effects, whereas λ-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 since 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 will only look at 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. Since $\lambda x.\ M$ 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 substitution are discussed below. The value of $\lambda f.\ fx$ applied to $\lambda y.\ y$ may be calculated by substituting the argument $\lambda y.\ y$ 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 macro expansion and in-line substitution of functions.

Renaming Bound Variables

Since λ-bindings in M can conflict with free variables in N, substitution $[N/x]M$ is a little more complicated than one might think at first. However, all of the complications can be avoided by following the variable convention: rename 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 (λf. λx. f (f x)) (λy. y+x). If we first rename bound variables and then perform β-reduction, then we get

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

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

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

However, you should be suspicious of the second reduction because the variable x is free in (λy. y+x), but becomes bound in λx. 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 will define the result $[N/x]M$ of substituting N for x in M by induction on the structure of M.

$[N/x]x = N$

$[N/x]y = y$ *where y is any variable different from x*

$[N/x](M_1M_2) = ([N/x]\ M_1)\ ([N/x]M_2)$

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

$[N/x]\ (\lambda y.M) = \ \lambda y.\ ([N/x]M)$ *where y is not free in N*

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

4.2.3 Programming in lambda calculus

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

Functions of several arguments

Lambda abstraction lets us treat any expression M as a function of any variable x by writing $\lambda x.M$. But 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 will not need to because ordinary λ-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 which, 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 using ordinary lambda abstraction. We define f_{curry} by

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

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

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

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 multi-argument functions in lambda calculus is usually called *Currying*, after the lambda calculus pioneer Haskell Curry.

Declarations

We saw earlier that we can regard simple let declarations

let x = M in N

as lambda terms by adopting the abbreviation

let $x = M$ in $N \quad = \quad (\lambda x.N)\ M$

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

Let $compose \ = \ \lambda f.\ \lambda g.\ \lambda x.\ f\,(gx)$ in $compose\ h\ h$

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

Recursion and Fixed Points

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

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 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 the definition 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

$$f(n) \;=\; \text{if } n=0 \text{ then } 1 \text{ else } n*f(n-1)$$

with the recursive declaration above. 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 equation above by using lambda abstraction to eliminate n from the left-hand side. This gives us

$$f \;=\; \lambda n.\ \text{if } n=0 \text{ then } 1 \text{ else } n*f(n-1)$$

Now consider the function G obtained by moving f to the right-hand-side of the equation.

$$G \;=\; \lambda f.\ \lambda n.\ \text{if } n=0 \text{ then } 1 \text{ else } n*f(n-1)$$

Although it might not be clear what sort of "algebra" is involved in this manipulation of equations, it is possible to check, 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.

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

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

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

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

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 given above, and calculate $fact\ 2 = 2!$ using the calculation rules of lambda calculus. Here are the calculation steps representing the first "call" to factorial.

$$fact\ 2 = (YG)\ 2$$

$$= G\,(YG)\ 2$$

$$= (\lambda f.\ \lambda n.\ \text{if } n=0 \text{ then } 1 \text{ else } n*f(n-1))\ \ (YG)\ \ 2$$

$$= (\lambda n.\ \text{if } n=0 \text{ then } 1 \text{ else } n*((YG)(n-1))\ \ 2$$

$$= \text{if } 2=0 \text{ then } 1 \text{ else } 2*((YG)\ (2-1))$$

= if 2=0 then 1 else 2*((*YG*) 1)

= 2*((*YG*) 1)

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 using 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 → instead of =, to indicate the direction 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 M β-reduces to N, and write M→N, if N is the result of applying one β-reduction step somewhere inside M. Most of the examples of calculation in this section use β-reduction, i.e., β-equivalence is used from left to right rather than right to left. For example,

*(**1**f. **1**z. f (f z)) (**1**y. y+x) ® **1**z.((**1**y. y+x) ((**1**y. y+x) z))) ® **1**z. z+x+x.*

Normal Forms

Intuitively, we think of M→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 above. 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 reduction sequence leading to a normal form:

*(**1**f. **1**x. f (f x)) (**1**y. y+1) 2*

*® (**1**x. (**1**y. y+1) ((**1**y. y+1) x)) 2*

*® (**1**x. (**1**y. y+1) (x+1)) 2*

*® (**1**x. (x+1+1)) 2*

® (2+1+1)

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

2+1+1

→ 3+1

$\rightarrow 4$

This example should give you 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 (λf. λx. f (f x)) (λy. y+1) 2, there were some steps where we had to chose from several possible subexpressions to evaluate. For example, in the second expression,

(λx. (λy. y+1) ((λy. 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, since we also have two choices in evaluating the purely arithmetic expression

2+1+1

An important property of lambda calculus is called *confluence*. In lambda calculus, as a result of confluence, evaluation order does not affect the final value of an expression. Put another way, if an expression M can be reduced to a normal form, then there is exactly one normal form of M, independent of the order in which we choose to evaluate subexpressions. While 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, the 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 using lambda calculus.

Some important properties of lambda calculus are:

- 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 using *Y*.)
- Evaluation in lambda calculus is order-independent. Due to 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 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*, where a state is a mathematical function representing the values in memory at some point in the execution of a program. Specifically, the meaning of this program will be a function that maps any state in which the value of z is some non-negative 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 non-negative 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 only on the denotational semantics of a program, the mathematical function from input to output associated with the program. For example, the program above was designed to compute the sum of all the non-negative integers up to n. If we verify that the actual denotational semantics of this program is this mathematical function, then we have proved that the program is correct. Some disadvantages of denotational semantics are that standard denotational semantics do not tell us anything about the running time or storage requirements of a program. This is sometimes an advantage in disguise, since 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 an operational semantics might be more useful, or we could use a 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 like 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 using only the denotations of *B*, *P* and *Q*; it should not be defined using 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, since these operations on programs involve replacing parts of programs without changing the overall meaning of the whole program.

4.3.1 Object language and meta-language

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, since this is the object of our attention, while the second language is called the meta-language, because it transcends the object language in some way.

To pick an example, let us consider the mathematical interpretation of a simple algebraic expression like 3 + 6 - 4 which 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, while the number 5 is meant to be in our meta-language. 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 meta-language, while the expression 3 + 6 - 4 is written using 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 Section 4.1.2.

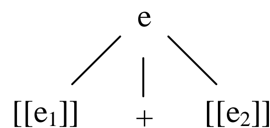
```
e ::= n | e+e | e-e
```

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 defined above as natural numbers 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.

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

$E[[0]] = \mathbb{0}$

$E[[1]] = \mathbb{1}$

$E[[nb]] = E[[n]] * \mathbb{2} + E[[b]]$

$E[[e_1+e_2]] = E[[e_1]] + E[[e_2]]$

$E[[e_1 - e_2]] = E[[e_1]] - E[[e_2]]$

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 since the parse trees $[[e_1]]$ and $[[e_2]]$ are smaller than the parse tree $[[e_1+e_2]]$.

On the right of the equal signs, numbers and arithmetic operations *, + and - are meant to mean 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 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 Section 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.

E[[x]](s) = s(x)

E[[0]](s) = $\mathbb{0}$

E[[1]](s) = $\mathbb{1}$

… = …

E[[9]](s) = $\mathbb{9}$

E[[nd]](s) = E[[n]](s) * $\mathbb{10}$ + 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)

Notice that the state matters in the definition in the base case, the value of a variable. Otherwise, this is essentially the same definition as 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. Since 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 will 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

$$State = Variables \rightarrow Values$$

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

$$Command = State \rightarrow State$$

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

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

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

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 defining 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))

Since 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 obtained by applying the semantics of P_2 the to the state obtained by applying the semantics of P_1 to state *s*.

C[[if e then P_1 else P_2]](s) is the state obtained 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 obtained 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, since 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 will just assume that a definition of this form can be made mathematically rigorous.

Example 1: We can calculate the semantics of various programs 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 $s_0(y)=2$. Since E[[x>y]](s_0) = false, we have

C[[if x>y then x :=y else y := x]](s_0)

= if E[[x>y]](s_0) then C[[x := y]](s_0) else C[[y := x]](s_0)

= C[[y := x]](s_0)

= modify(s_0 , y, E[[x]](s_0))

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 2: 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

s_1 = C[[x:= 0]](s_0)

s_2 = C[[y := 0]](s_1)

Using the semantics of assignment, as above, we have

s_1 = modify(s_0 , x, 0)

s_2 = modify(s_1 , y, 0)

Returning to the program above, we have

$C[[x := 0;\ \ y:=0;\ \ \text{while}\ \ x \le z\ \ \text{do}\ \ (y := y+x; x := x+1)\ ]](s_0)$

$= C[[\ y:=0;\ \ \text{while}\ \ x \le z\ \ \text{do}\ \ (y := y+x; x := x+1)\ ]](\ C\ [[\ x := 0\ ]](s_0)\ )$

$= C[[\ y:=0;\ \ \text{while}\ \ x \le z\ \ \text{do}\ \ (y := y+x; x := x+1)\ ]](s_1\ )$

$= C[[\ \text{while}\ \ x \le z\ \ \text{do}\ \ (y := y+x; x := x+1)\ ]](\ C\ [[\ y := 0\ ]](s_1)\ )$

$= C[[\ \text{while}\ \ x \le z\ \ \text{do}\ \ (y := y+x; x := x+1)\ ]](s_2)$

$= \ \text{if}\ \ \text{not}\ E[[x \le z\ ]](s_2)\ \text{then}\ s_2$

$\quad \text{else}\ C[[\ \text{while}\ \ x \le z\ \ \text{do}\ \ (y := y+x; x := x+1)\ ]](C[[y := y+x; x := x+1\ ]](s_2))$

$= C[[\ \text{while}\ \ x \le z\ \ \text{do}\ \ (y := y+x; x := x+1)\ ]](s_3)$

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

$C[[\ \text{while}\ \ x \le z\ \ \text{do}\ \ (y := y+x; x := x+1)\ ]](s_3)$

$= \text{if}\ \ \text{not}\ E[[x \le z\ ]](s_3)\ \text{then}\ s_3$

$\quad \text{else}\ C[[\ \text{while}\ \ x \le z\ \ \text{do}\ \ (y := y+x; x := x+1)\ ]](C[[y := y+x; x := x+1\ ]](s_3))$

$= C[[\ \text{while}\ \ x \le z\ \ \text{do}\ \ (y := y+x; x := x+1)\ ]](s_4)$

$= \ \text{if}\ \ \text{not}\ E[[x \le z\ ]](s_4)\ \text{then}\ s_4$

$\quad \text{else}\ C[[\ \text{while}\ \ x \le z\ \ \text{do}\ \ (y := y+x; x := x+1)\ ]](C[[y := y+x; x := x+1\ ]](s_4))$

$= C[[\ \text{while}\ \ x \le z\ \ \text{do}\ \ (y := y+x; x := x+1)\ ]](s_5)$

$= s_5$

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 where 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, including *Foundations for Programming Languages* by J.C. Mitchell, *Theories of Programming Language by* J.C. Reynolds, and *The Formal Semantics of Programming Languages: An Introduction* by G. Winskel.

4.3.4 Perspective and nonstandard semantics

There are several ways of viewing the standard methods of denotational semantics. Typically, a denotational semantics is given by associating a function with each program. As many researchers in denotational semantics have observed, a mapping from programs

to functions must be written in some meta-language. Since lambda calculus is a useful notation for functions, it is common to use some form of lambda calculus as a meta-language. 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.

While 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 set above, 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 using denotational semantics, where the meaning of an expression is an element of a finite set.

Since the halting problem is unsolvable, program analysis algorithms are usually designed to be *conservative.* "Conservative" means that ther are no false positivies: an algorithm will only output *correct* 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 using an abstract representation of machine states that only keep track 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

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

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

C[[x :=e]](s) = if E[[e]](s)=*Ok* then modify(s, x ,*init*) 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

C[[x := 0]](s_0) = modify(s_0,x, *init*)

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

The clauses for sequences, P1; P2, is essentially straightforward. For s ≠ *error*,

C[[P_1;P_2]](s) = if C[[P_1]](*s*) = *error* then *error* else C[[P_2]](C[[P_1]](*s*))

The clause for conditional is more complicated since our analysis tool is not going to try to evaluate a boolean expression. Instead, we will treat conditional as if it is possible to execute either branch. (This is the conservative part of the analysis.) Therefore, we only change a variable to initialized 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

$s_1 + s_2 = \lambda v \in Variables.$ If $s_1(v) = s_2(v) = init$ then *init* else *uninit*

We define the meaning of a conditional statement by

C[[if e then P_1 else P_2]](s)

= if E[[e]](s)= *error* or C[[P_1]](s)= *error* or C[[P_2]](s) = *error*

then *error*

else C[[P_1]](s) + C[[P_2]](s)

For example, using s_0 as above, we have

C[[if 0=1 then x := 0 else x := 1; y := 2]](s_0) = modify(s_0, x, *init*)

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

For simplicity, we will 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* since 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 only of 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;}
```

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 code above declares a new variable with the same name as 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 program fragment above 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, it is easy to see that the declaration of a new variable z does not change the value of any previously existing variable.

4.4.2 Functional vs. imperative programs

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

Some people use the phrase "functional language" to refer to languages that do not have expressions with side effects or any other form of imperative construct. However, we will use the more emphatic phrase *pure functional language* for declarative languages that are designed around flexible constructs for defining and using functions. We learned in Section 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 $x_1, ..., x_n$, all occurrences of an expression e containing only variables $x_1, ..., x_n$ 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 only needs to be evaluated 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, since 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*. While 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 by 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 have the same meaning. A traditional counter-example to referential transparency, attributed to the philosopher of language 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 by 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 by another of equal value anywhere in a program, without changing the meaning of the program. This is a property of pure functional languages.

The reason referential transparency is subtle is that it depends on the value we associate with expressions. In imperative programming languages, we can say that a variable x refers to its value or to its location. If we say that a variable refers to its location in memory, then imperative languages *are* referentially transparent, since replacing one variable by 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, since 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 argues that pure functional programming languages are better than imperative ones. The lecture and the accompanying paper, published by the ACM, 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 independent 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 where 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.

In order to advance his argument, Backus proposed a pure functional programming language called FP, an abbreviation of *F*unctional *P*rogramming. FP contains a number of basic functions and a rich set of combining forms 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

```
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, inner product can be expressed as

Inner_product = (Insert +) ° (ApplyToAll ×) ° Transpose

where ° is function composition and Insert, ApplyToAll and Transpose are vector operations. Specifically, 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 Section 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 can be understood 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 by 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 discussed below.

In retrospect, Backus' argument seems more plausible than his solution. The importance of program correctness, readability, and reliability have increased in comparison with 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 an FP implementation at IBM, but the language was not widely used. One problem is that the language has an difficult syntax. Perhaps more importantly, there are severe limitations in the kind of data structures and control structures that can be defined. Functional programming 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 above. We can see how parallelism arises in pure functional languages using the example of a function call $f(e_1,...,e_n)$, where function arguments $e_1,...,e_n$ are expressions that may need to be evaluated.

Functional programming: we can evaluate $f(e_1,...,e_n)$ by evaluating $e_1,...,e_n$ in parallel, since values of these expressions are independent.

Imperative programming: For an expression such as f(g(x), h(x)), the function g might change the value of x. Hence the arguments of functions in imperative languages must be evaluated in a fixed, sequential order. This 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. Since 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.

While 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 e_1 then e_2 else e_3

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?

While we have considered many of the potential advantages of pure functional programming languages in this section, we do not have a definitive answer. From one theoretical point of view, functional programming languages are as good as imperative programming languages. This can be demonstrated by giving a translation of C programs into either 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 towards answering this question was done at IBM on the FP project (which was cancelled soon after Backus retired). Additional efforts using 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 future time, pure or mostly-pure functional programming languages will become more popular. Whether or not that happens, functional programming 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 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 since 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 using 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 since it does not consider issues such as running time and memory requirements. However, denotational semantics is useful for reasoning about correctness and has been used to develop and study program-analysis methods that are used in compilers and programming environments. Some of the exercises at the end of the chapter present applications for type checking, initialization-before-use analysis, and simplified security analysis. From a theoretical point of view, denotational semantics shows that every imperative program can be transformed into an equivalent functional program.

In pure functional programs, syntactically identical expressions within the same scope have identical values. This property allows certain optimizations and makes it possible to execute independent subprograms in parallel. Since 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.

4.6 **Exercises**

(* Insert exercises from LaTeX files *)

Robin Milner

A thoughtful, engaging, and optimistic person, Robin Milner has had a profound effect on several areas of computer science and on computing in the U.K in general. Always looking for new insight and open to new ideas, Robin is an unassuming but forceful presence at any discussion over coffee after a conference talk or workshop presentation.

Milner was awarded the 1991 ACM Turing Award for “several distinct and complete achievements: LCF, probably the first theoretically based yet practical tool for machine-assisted proof construction; ML, the first language to include polymorphic type inference and a type-safe exception-handling mechanism; CCS, a general theory of concurrency; and full abstraction, the study of the relationship between operational and denotational semantics."

After approximately 20 years in Edinburgh, Robin returned to Cambridge University, where he held a Chair in Computer Science and was Head of the Computer Laboratory until his retirement in 1999.

Chapter 5. The Algol Family and ML

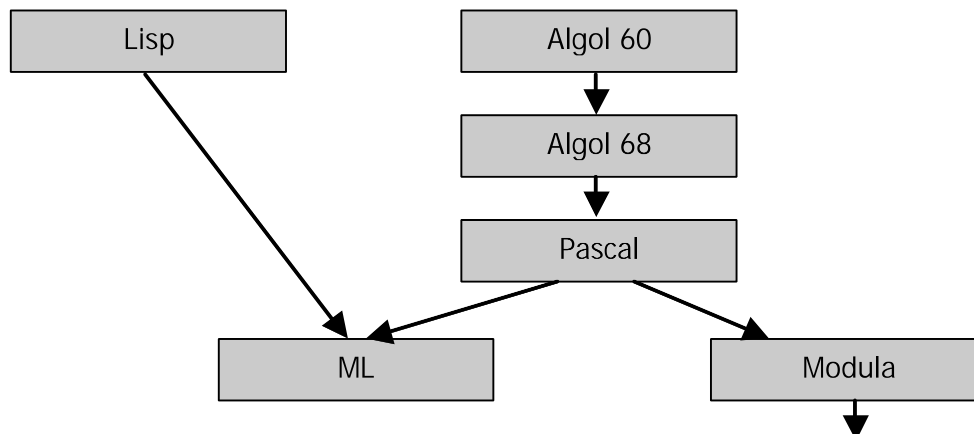
The Algol-like programming languages evolved in parallel with the Lisp family of languages, beginning with Algol 58 and Algol 60 in the late 1950s. The most prominent Algol-like programming languages are Pascal and C, although C differs from most of the Algol-like languages in some significant ways.

In this chapter, we look at some of the historically important languages from the Algol family, including Algol 60, Pascal, and C. Since many of the central features of the Algol family are used in ML, we then use the ML programming language to discuss some important concepts in more detail. The ML section of this chapter is also a useful reference for later chapters that use ML examples to illustrate concepts that are not found in C.

There are many Algol-related languages that we will not have time to cover, such as Algol 58, Algol W, Euclid, EL1, Mesa, Modula-2, Oberon, and Modula-3. We will discuss Modula and modules in Chapter 9.

5.1 The Algol family of programming languages

A number of important language ideas were developed in the Algol family, which began with work on Algol 58 and Algol 60 in the late 1950s. The Algol family developed in parallel with Lisp languages and led to the late development of ML and Modula.



The main characteristics of the Algol family are the familiar colon-separated sequence of statements used in most languages today, block structure, functions and procedures, and static typing.

5.1.1 **Algol 60**

Algol 60 was designed between 1958 and 1963, by a committee that included many important computer pioneers, such as John Backus (designer of Fortran), John McCarthy (designer of Lisp), and Alan Perlis. Algol 60 was intended to be a general-purpose language, which at the time meant there was emphasis on scientific and numerical applications. In comparison with Fortran, Algol 60 provided better ways to represent data structures and, like LISP, allowed functions to be called recursively. Until the development of Pascal, Algol 60 was the academic standard for describing complex algorithms in scientific and engineering publications.

Some important features of Algol 60 are:

- Simple statement-oriented syntax, involving colon-separated sequences of statements
- Blocks, indicated by begin ... end (corresponding to curly braces { … } in C)
- Recursive functions and stack storage allocation
- Fewer ad hoc restrictions than previous languages. For example,
  - General expressions inside array indices
  - Procedures could be called with procedure parameters
- A primitive static type system that was later improved upon on Algol 68 and Pascal.

Here is an example program (explained below) that will give you some feel for Algol 60 syntax:

```
real procedure average(A,n);
    real array A; integer n;
    begin
        real sum; sum := 0;
        for i = 1 step 1 until n do
            sum := sum + A[i];
        average := sum/n
    end;
```

In Algol, the two-character sequence := is used for assignment, and the single character = for test for equality. In this program, notice that the types of the parameters to the procedure (function) are declared in the line following the procedure name. While A is declared to be a real array, no array bounds are given as part of the declaration. The return value of the procedure is given by assigning a value to the name of the procedure. (This is the assignment to average in the last line.) An irritating syntactic peculiarity of Algol 60 is the way that semicolons must be (and must not be) used between statements. In particular, it would be a syntactic error to place a semicolon after the last assignment and before the keyword end. There is a systematic explanation for why semicolons are needed some places and not others in Algol60, but the systematic reason is hard for programmers to learn, remember and apply when programs are edited.

There were a number of trouble spots in Algol 60 that motivated computer scientists to develop better programming languages. For example,

- *The Algol 60 type discipline had some shortcomings. Two examples illustrated in more detail in the exercises are:*
  - The type of a procedure parameter to a procedure does not include the types of parameters,
  - An array parameter to a procedure is given type array, without array bounds.
- *Algol 60 was designed around two parameter-passing mechanisms, pass-by-value and pass-by-name.*
  - Pass-by-name interacts badly with side effects
  - Pass-by-value is expensive for arrays
- *There are some awkward issues related to control flow, such as memory management when a program jumps out of a nested block.*

Pass-by-name. Perhaps the strangest feature of Algol 60, in retrospect, is the use of pass-by-name. In pass-by-name, the result of a procedure call is the same as if the formal parameter were substituted into the body of the procedure. This rule for defining the result of a procedure call by copying the procedure and substituting for the formal parameters is called the Algol 60 *copy rule*. While the copy rule works well for pure functional programs, as illustrated by β-reduction in lambda calculus, the interaction with side effects to the formal parameter are a bit strange. Here is an example program showing a technique referred to as "Jensen's device": passing an expression and a variable it contains to a procedure so that the procedure can use one parameter to change the location referred to by the other:

```
begin integer i;
    integer procedure sum(i, j);
        integer i, j;
            comment parameters passed by name;
        begin integer sm; sm := 0;
            for i := 1 step 1 until 100 do  sm := sm + j;
            sum := sm
        end;
    print( sum(i, i*10 ))
end
```

In this program, the procedure sum(i,j) adds up the values of j as i goes from 1 to 100. If you look at the code, you will realize that the procedure makes no sense unless changes to i cause some change in the value of j; otherwise, the procedure just computes 100*j. In the call sum(i, i*10)) shown here, the for loop in the body of procedure sum adds up the value of i*10 as i goes from 1 to 100.

BNF. An important byproduct of the Algol 60 design effort was the invention of Backus-Normal Form (or BNF), which was used in the Algol 60 report to define the well-formed programs of the language. BNF, summarized in Section 4.1.2, remains the standard notation for describing the syntax of programming languages. Although Algol 60 was very influential and commonly used in academic circles and in Europe, it was not a commercial success in the United States.

5.1.2 Algol 68

Algol 68 was intended to remove some of the difficulties found in Algol 60 and improve the expressiveness of the language. However, in the end, the Algol 68 committee produced a design that was more problematic than Algol 60. One main problem is that while programming in Algol 68 appears no more difficult than Algol 60, some features of Algol 68 made it difficult to compile efficiently. One source of difficulty was the combination of procedure parameters and procedure return values, which was not well understood at the time. (We will discuss the implementation consequences of higher-order functions in Section 7.4.) Another reason that Algol 68 was not entirely successful was that the authors chose to define entirely new terminology for the language and its documentation. This made it difficult for programmers to move from Algol 60 to Algol 68.

One contribution of Algol 68 was its regular, systematic type system. For some reason, the Algol 68 designers chose to call types "modes". The modes of Algol 68 are either primitive or compound modes. The primitive modes included

> int, real, char, bool, string, complex, bits, bytes, semaphore, format (for input and output), and file.

The compound modes include modes formed using the forms:

> array, structure, procedure, set, and pointer.

These type constructions can be combined without restriction, so that a programmer can build an array of pointers to procedures, for example. The decision to allow unrestricted combinations of the mode constructors made the type system seem more systematic than previous languages.

Some other advances in Algol 68 were in the areas of memory management and parameter passing. (We will cover the general concepts of memory management, parameter passing, and related issues in Chapter 7.) Algol 68 memory management involves a stack for local variables and heap storage for data that is intended to live beyond the current function call. Like C, Algol 68 data on the heap is explicitly allocated, but unlike C heap data is reclaimed by garbage collection. This combination of explicit allocation and garbage collection carried over into Pascal. Algol 68 parameter passing is pass-by-value, with pass-by-reference accomplished by pointer types. This is essentially the same design as adopted in C some years later. The decision to allow independent constructs to be combined without restriction also led to some complex features, such as assignable procedure variables.

5.1.3 Pascal

Pascal was design in the 1970s by Niklaus Wirth, using data structuring ideas advanced by C.A.R. (Tony) Hoare. Wirth first designed and implemented a language called Algol W and then refined the design of Algol W to produce Pascal. Wirth designed Pascal around a series of teaching exercises, enumerated in his book, *Algorithms + Data Structures = Programs*. He also designed the language so that it could have a simple one-pass compiler.

Pascal was significantly simpler than Algol 68 and achieved more widespread acceptance than either Algol 60 or Algol 68. Although the use of Pascal declined in the 1990s, Pascal was one of the most widely used programming languages over approximately a twenty year period. Pascal was very successful as a programming language for teaching, in part because it was designed explicitly for this purpose. Pascal was also used for a significant number of production programming projects, including operating systems and applications for the Apple Macintosh.

The Pascal type system is more expressive than the Algol 60 type system. The type system also repairs some of the Algol 60 type loopholes, such as Algol 60 procedure types that do not include the types of parameters. The Pascal type system is also simpler and more limited than the Algol 68 type system, eliminating some of the compilation difficulties of Algol 68.

An important contribution of the Pascal type system is the rich set of data structuring concepts. These include records (similar to C structs), variant records (a form of union type), and subranges. For example, the subrange [1 .. 10] of integers between 1 and 10 became a type for the first time in Pascal. A restriction that made Pascal simpler than Algol 68 was that procedure parameters could not be procedures with procedure parameters. More specifically, in Pascal syntax, procedures of the form

```
procedure DoSomething( j, k : integer );
procedure DoSomething( procedure P(i:integer); j,k, : integer);
```

are allowed. The first is a procedure with integer parameters, and the second is a procedure whose parameter is a procedure with integer parameters. However, a procedure of the form

```
procedure NotAllowed( procedure MyProc( procedure P(i:integer)));
```

with a procedure parameter that has a procedure parameter, is *not* allowed.

One place where Wirth's focus on teaching and on systematic language design caused a small problem was in the typing of array parameters. In Pascal, the type of an array has the form

```
array ⟨indexType⟩ of ⟨entryType⟩
```

where ⟨indexType⟩ is often a subrange type. The important detail here is that the index type is part of the type of an array, and two array types are equal only if they have the same index type and entry type. This definition of array type allows a procedure declaration such as

procedure p(a : array [1..10] of integer)

where the argument to the procedure is an array of some array type, but does not allow

procedure p(n: integer, a : array [1..n] of integer)since array [1..n] of integer is not considered a legal type in Pascal. A procedure of the form

procedure p(a : array [1..10] of integer)

may only be called with an array of length 10 as an actual parameter. This is an unfortunate limitation. If we want to write a sort procedure, for example, then we will have to write the procedure out for sorting arrays of some fixed length. If we want to sort arrays of several different lengths, then Pascal (as originally designed) would make it necessary to copy over the procedure several times, changing the array length in each copy.

Not only is this awkward, it is also unnecessary because the memory associated with an array is already allocated before it is passed to any procedure. Therefore, there is no reason, other than type checking, for a procedure to only allow array parameters of a fixed length. Since this aspect of the Pascal design was awkward and unnecessary, later versions of Pascal have this limitation removed.

5.1.4 Modula

The Modula programming language is a descendent of Pascal, developed by Pascal designer Niklaus Wirth in Switzerland in the late 70s. The main innovation of Modula over Pascal is a module system, used for grouping sets of related declarations into program units. Some examples of Modula-2, a successful version of the language, appear in Section 9.3.1, as part of the discussion of program modules.

5.2 The development of C

While Pascal was a successful academic and teaching language, C eventually eclipsed Pascal as a production programming language. There are several reasons for the success of C. One reason that is unrelated to the design of the language itself is the popularity of the Unix operating system, which was written in C. When writing programs to run under Unix, all of the basic system calls are immediately available in C. Therefore, it is easier to write many Unix applications in C than in other programming languages. Another reason for the popularity of C is that it has a distinctive memory model that is close to the underlying hardware. Although C has many of the same concepts as Pascal, C is less rigid in its enforcement of basic principles and restrictions. Many C programmers like the resulting flexibility of C.

C was originally designed and implemented from 1969 to1973, as part of the Unix operating system project at Bell Laboratories. C was designed by Dennis Ritchie, one of the original designers of Unix, so that he and Ken Thompson could build Unix in a language that they liked. The design evolved from Ritchie and Thompson's B language (hence the name C, the next letter in the alphabet), which was in turn based on a language called BCPL. Significant changes in C occurred in 1977-1979, as part of a push to

achieve portability of the Unix system, and in the mid-1980s when an ANSI committee standardized the language. BCPL was a systems programming language designed in the 1960s and used by Bell Laboratories members of the Multics operating system project. B was a pared-down version of BCPL, designed to run on the small PDP computer used by the Unix project. The main difference between B and C is that B was untyped while the C language has types and type-checking rules.

One characteristic that distinguishes C from other popular languages is the treatment of memory locations, arrays, and pointers. This part of C is inherited from BCPL and B. The only data type in B is the "word," or "cell," a fixed-length bit pattern. Memory in BCPL and B is presented to the programmer as a linear array of words. Pointers are treated as integer indices into this array and programmer-declared arrays are treated as contiguous words drawn from the larger array of all memory words. This view has several consequences. One is that arrays and pointers are largely equivalent, as described below for C. Another consequence is that since pointers are integer indices in the memory array, pointer arithmetic is considered meaningful: if p is the address of a memory location, then p+1 is the address of the next location.

C arrays and pointers

In C, pointers and arrays are declared differently, as if pointers and arrays are different types of values. For example, the following code declares a pointer p to an integer location and an array A of integers:

```
int * p;
int A[5];
```

In most languages, there is some operation for dereferencing a pointer. Dereferencing is the operation that returns the location pointed to by the pointer. In most languages with arrays, there is an indexing operation that can be used to find one of the locations within the array. There are some similarities between these two operations, but most typed languages (including others in the Algol family) would consider dereferencing an array name or indexing a pointer illegal. In C, however, arrays are effectively treated as pointers. To quote Dennis Ritchie's 1975 C Reference Manual,

> Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. … By definition, the subscript operator [ ] is interpreted in such a way that "E1[E2]"is identical to "*((E1)+(E2))." Because of the conversion rules which apply to +, if E1 is an array and E2 is an integer, then E1[E2] refers to the E2-th member of E1.

There is no other programming language in widespread use that allows pointer arithmetic in this way.

Critique

An important feature of C has been the tolerance of C compilers to type errors. This is partly because C evolved from typeless languages. Ritchie had to adapt existing programs as the language developed, and make allowance for existing code in a typeless language. As C evolved further, and was later standardized by an ANSI committee, backward

compatibility with then-existing C code also prevented strong typing restrictions. Although some C programmers have liked the ability to write and compile programs with type errors, most C programmers have eventually come to consider the weak type checking of many C compilers to be a disadvantage. In fact, one of the most commonly cited advantages of C++ over C is the fact that C++ provides better type checking.

As Dennis Ritchie says in The Development of the C Language (ACM Second History of Programming Languages conference, 1993), “C is quirky, flawed, and an enormous success.” While accidents of history surely helped, C evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments.

An interesting discussion may be found in Kernighan’s “Why Pascal is not my favorite programming language”, Bell Labs CSTR 100, July 1981. If you are interested in doing extra reading, you can find this document on the web.

5.3 The LCF system and ML

ML might be called a mostly-functional language with imperative features, or perhaps a “function-oriented imperative language”. ML has very flexible function features, similar to Lisp, allowing functions to be created in-line as parts of expressions, passed as arguments to functions, and returned as function results. At the same time, it is possible to write imperative Algol-like programs in a syntax that resembles the Algol family, with approximately the same degree of ease as modern descendants of Algol. ML also has concurrent extensions, making it suitable for developing concurrent systems, and an object-oriented extension. However, our main use of ML in this part of the book will be to examine concepts common to Algol-like languages, Lisp-like languages, and concurrent and object-oriented extensions of these languages. Therefore, we will focus primarily on the core fragment of ML.

The main reasons for looking at ML in some detail are:

- ML illustrates most of the important concepts of the Lisp/Algol families of languages.
- Type systems have been an important part of programming language design from 1960 to the present day, and the ML type system is often considered the cleanest and most expressive type system to date.
- Since most readers are familiar with C, and many have not written many programs in significantly different languages, it is useful to have a language other than C to use for examples in the following chapters.
- ML allows higher-order functions and other constructions that are discussed in the following chapters.

One distinguishing feature of ML is its type system, which extends the successful Pascal type system in a number of ways. Unlike C, which has numerous loopholes, the ML type

system is sound in a precise mathematical sense. Specifically, if the type checker determines that an expression has a certain type, then any terminating evaluation of that expression is guaranteed to produce a legitimate value of that type. For example, if an expression has a type like "pointer to string", then the value of that expression is guaranteed to be a pointer to allocated memory that contains a string. It cannot be a dangling pointer to a location that has been deallocated or used to store some value other than a string.

Before ML, programming languages with sound type systems were generally considered unpleasantly restrictive. Many C programmers have considered it important to "break" the type system in various ways (confusing integers and pointers, for example), and Lisp fans have valued their freedom from static typing. However, the ML type system is unobtrusive, since many type declarations are automatically deduced by the compiler, and flexible, since the type system allows an expression to have many possible types. We will explore these aspects of the ML type system in more detail in Chapter 6.

5.3.1 History and Intended Applications of ML

Most successful programming languages were originally designed for a single application or a set of closely related programming tasks. The ML programming language was designed by Robin Milner and his associates as part of the LCF project. The LCF project, aimed at developing a *Logic for Computable Functions*, drew inspiration from a set of logical principles outlined by Dana Scott. Robin Milner's goal was to build a system that would make it practical to prove interesting properties of functional programs in an automated or semi-automated manner. His LCF project started at Stanford in 1970 and continued at Edinburgh through the 1980s, making substantial progress towards this goal and stimulating a number of related efforts in the process.

ML was designed as the *M*eta-*L*anguage (hence its name) of the LCF System. Its original purpose was for writing programs that would attempt to construct mathematical proofs. As any reader who has developed mathematical proofs will know, this can be a very difficult task. In many cases, it is necessary to try a number of methods for finding proofs. A fundamental concept in the LCF system is that of *proof tactic*. A proof tactic is a function that, given a formula making some assertion, tries to find a proof of the formula. Since a tactic may search indefinitely, or reach some situation where it is clear that no further search is likely to produce a proof, there are three possible results of applying a tactic to a formula:

$$\text{tactic(formula)} \quad = \quad \begin{cases} \text{succeed and return proof} \\ \text{search forever} \\ \text{fail} \end{cases}$$

The concept of tactic may be used to understand some basic properties of the ML programming language. Since the goal of LCF is to find correct proofs, a programming language mechanism that ensures correctness, in whole or in part, might improve the LCF system. An idea that was adopted in LCF was to try to use a type system to

distinguish successful proofs from unsuccessful ones. In particular, there was a type proof, with the intent that values of type proof are correct proofs, not incorrect ones.

Once we have a type of correct proofs, a problem arises. If a tactic fails to find a proof, what should the function do? More specifically, since a tactic is a function from formulas to proofs, the type of a tactic would be a function type,

tactic : formula $\rightarrow$ proof

However, this type seems to say that if a tactic is applied to a function, the result will be a proof. But what if the formula is not a correct statement and therefore has no proof? The solution was to develop an exception mechanism and allow a tactic to raise an exception if the computation determines that no proof will be found. From this inspiration, Milner developed the first type-safe exception mechanism, one of the accomplishments that led to his Turing Award in 1991. Allowing for the possibility of exceptions, a function f maps A to B, written

$f : A \rightarrow B$

in ML, means

For all x in A,

if f(x) terminates normally without raising an exception, then f(x) is in B.

Thus type correctness and exceptions, two basic concepts in ML, arose naturally as the result of the intended application of ML.

Another emphasis of ML is the use of higher-order functions. This can also be attributed to the interest in defining complex proof-search tactics: Since a tactic is a function, a method for combining tactics into a proof-search strategy is a function from functions to functions. For example, here is the outline of a function that combines two tactics according to an if-then-else strategy:

f($tactic_1$, $tactic_2$) =

λformula. try $tactic_1$(formula)

else $tactic_2$ (formula)

$tactic_1 tactic_2$, the function f returns a tactic that, given a formula, first tries to prove the formula using $tactic_1$ and then uses $tactic_2$ if $tactic_1$ fails.

5.4 The ML programming language

Since we will use ML in the next few chapters of the book to illustrate properties of programming languages, we will study ML in a little more detail than some other languages. The version of ML that we will use is called Standard ML 97. Compilers for SML97 are available on the Internet without charge. Several books and manuals covering the language are available. In addition to online sources easily located by web search, Ullman's *Elements of ML Programming* (Prentice Hall, 1994) is a good reference.

5.4.1 Interactive Sessions and the Run-time System

Most ML compilers are based on the same kind of read-eval-print loop as many Lisp implementations. The standard way of interacting with the ML system is to enter expressions and declarations one at a time. As each is entered, the source code is type-checked, compiled, and executed. Once an identifier has been given a value by a declaration, that identifier can be used in subsequent expressions.

Expressions

For expressions, user interaction with the ML compiler has the form

```
— <expression>;
val it = <print_value> : <type>
```

where "—"is the prompt for user input and the line below is output from the ML compiler and run-time system. The lines above show that if you type in an expression, the compiler will compile the expression and evaluate it. The output is a bit cryptic: "it" is a special identifier bound to the value of the last expression entered, so "it = <print_value> : <type>" means that the value of the expression is <print_value> and this is a value of type <type>. It is probably easier to understand the idea from a few examples. Here are four lines of input and the resulting compiler output:

```
— (5+3) -2;
val it = 6 : int
— it + 3;
val it = 9 : int
— if true then 1 else 5;
val it = 1 : int
— (5 = 4);
val it = false : bool
```

In words, the value of the first expression is the integer 6. The second expression adds 3 to the value it of the previous expression, giving integer value 9. The third expression is an if-then-else, which evaluates to the integer 1, and the fourth expression is a Boolean-valued expression (comparison for equality) with value false.

Each expression is parsed, type-checked, compiled and executed before the next input is read. If an expression does not parse correctly or does not pass the type checking phase of the compiler, no code is generated and no code is executed. The ill-typed expression

```
if true then 3 else false;
```

for example, parses correctly since this has the correct form for an if-then-else. However, the type checker rejects this expression since the ML type checker requires the "then" and "else" parts of an if-then-else expression to have the same types, as described below. The compiler output for this expression includes the error message

```
stdIn:1.1-29.10 Error: types of rules don't agree [literal]
```

indicating a type mismatch. A full discussion of ML type checking appears in Chapter 6.

Declarations

User input can be an expression or a declaration. The standard form for ML declarations, followed by compiler output, is:

```
— val <identifier> = <expression>;
val <identifier> = <print_value> : <type>
```

The keyword val stands for "value." When a declaration is given to the compiler, the value associated with the identifier is computed and bound to that identifier. Since the value of the expression used in a declaration has a name, the compiler output uses this name instead of "it" for the value of the expression. Here are some examples:

```
— val x=7+2;
val x = 9 : int
— val y = x+3;
val y = 12 : int
— val z = x*y - (x div y);
val z = 108 : int
```

In words, the first declaration binds the integer value 9 to the identifier x. The second declaration refers to the value of x from the first declaration and binds the value 12 to the identifier y. The third declaration refers to both of the previous declarations and binds the integer value 108 to the identifier z.

You might notice that integer division in ML is written "div" instead of "/". There are a few syntactic peculiarities of ML like this, especially when it comes to integer and real number arithmetic. As you will see from the discussion of ML type inference in Chapter 6, it is useful for the compiler to distinguish operations of different types. In ML, / is used for real number (floating point) division, and there is no automatic conversion from integer to real. Therefore, x/y would not have been syntactically well-formed if we had written this instead of x div y in the declaration of z.

Functions can be declared using the keyword fun (which stands for "function") instead of val. The general form of user input and compiler output is

```
— fun <identifier> <arguments> = <expression>;
val <identifier> = fn  <arg_type> → <result_type>
```

This declares a function whose name is <identifier>. The argument type is determined by the form of <arguments> and the result type is determined by the form of <expression>.

Here is an example:

```
— fun f(x) = x + 5;
val f = fn : int → int
```

This declaration binds a function value to the identifier f. The value of f is a function from integers to integers. The same function can be declared using a val declaration, by writing

```
— val f = fn x => x+5;
val f = fn : int → int
```

In this declaration, the identifier f is given the value of expression fn x => x+5, which is a function expression like (lambda (x) (+ x 5)) in Lisp or λx. x+5 in lambda calculus. We will discuss function declarations and function expressions further in Section 5.4.3.

The system prints fn for the value of f because the value of a function is not printable. One reason why function values are not printed is that most functions are infinite values in principle – an integer function has infinitely many possible results, one for each possible integer argument. After a function is declared, the compiler stores compiled code for the function. It would be possible to print the compiled code, but this is not in ML. It is in some other target low-level language, and printing it would not usually be useful to a programmer.

Identifiers vs variables: An important aspect of ML is that the value of an identifier cannot be changed by assignment. More specifically, if an identifier x is declared by val x = 3, for example, then the value of x will always be 3. It is not possible to change the value of x by assignment. In other words, ML declarations introduce constants, not variables. The way to declare an assignable variable in ML is to define a reference cell, which is similar to a cons cell in Lisp, except that reference cells do not come in pairs. References and assignment are explained in Section 5.4.5.

Although most readers will initially think otherwise, the ML treatment of identifiers and variables is more uniform than the treatment of identifiers and variables in languages such as Pascal and C. If an integer identifier is declared in C or Pascal, it is treated as an assignable variable. On the other hand, if a function is declared and given a name in either of these languages, the name of the function is a constant, not a variable. It is not possible to assign to the function name and change it to a different function. Thus Pascal and C choose between variables and constants according to the type of the value given to the identifier. In ML, a val declaration works the same way for all types of values.

5.4.2 Basic types and type constructors

The core expression, declaration, and statement part of ML is best summarized by listing the basic types along with the expression forms associated with each type.

Unit

The type unit has only one element, written as empty parentheses:

```
( ) : unit
```

Like void in C, unit is used as the result type for functions that are executed only for side-effects. Unit is also used as the type of argument for functions that have no arguments. C programmers may be confused by the fact that "unit" suggests one element, while "void" seems to mean no elements. From a mathematical point of view, "one element" is correct.

In particular, if a function is supposed to return an element of an empty type, then that function cannot return since the empty set (or empty type) has no elements. On the other hand, a function that returns an element of a one-element type can return. However, we do not need to keep track of what value such a function returns, since there is only one thing that it could possibly return. The ML type system is based on years of theoretical study of types; most of the typing concepts in ML have been considered with great care.

Bool

There are two values of type bool, true and false.

```
true : bool
false : bool
```

The most common expression form associated with Booleans is conditional, with

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

having the same type as e2 and e3 if these have the same type and e1 has type bool. There is no if-then without else, since a conditional expression must have a value whether the test is true or false. For example, an expression

```
— val nonsense = if a then 3;
```

is not legal ML since there is no value for nonsense if the expression a is false. More specifically, the input "if a then 3" does not even parse correctly; there is no parse tree for this string in the syntax of ML.

There are also ML boolean operations for and, or, not, and so on. These are similar to AND, OR, NOT in Pascal or &&, || and ! in C, with some minor differences. Negation is written not, conjunction (and) is written andalso and disjunction (or) is written orelse. For example, here is a function that determines whether its two arguments have the same Boolean value, followed by an expression that calls this function:

```
— fun equiv(x,y) = (x andalso y) orelse ((not x) andalso (not y));
val equiv = fn : bool * bool -> bool
— equiv(true,false);
val it = false : bool
```

In words, Boolean arguments x and y are the same Boolean value if they are either both true or both false. The first subexpression, (x andalso y), is true if x and y are both true and the second subexpression, ((not x) andalso (not y)), is true if they are both false.

The reason for the long names andalso and orelse is to emphasize evaluation order. In an expression (a andalso b), where a and b are both expressions, a is evaluated first. If a is true, then b is evaluated. Otherwise, the value of expression (a andalso b) is determined to be false without evaluating b. Similarly, b in (a orelse b) is evaluated only if the value of a is false.

Integers

Many ML integer expressions are written in the usual way, with number constants and standard arithmetic operations:

```
0,1,2,...,-1,-2,... : int
+, -, *, div : int * int → int
```

The operator div is a binary infix operator on integers, used as follows:

```
— fun quotient(x,y) = x div y;
val quotient = fn : int * int → int
```

The identifier div by itself is not an expression, though. Similarly, +, -, and * are infix binary operators.

Strings

Strings are written as sequence of symbols between double quotes:

```
"William Jefferson Clinton" : string
"Boris Yeltsin" : string
```

String concatenation is written ^, so we have

```
— "Chelsey" ^ " " ^ "Clinton";
val it = "Chelsey Clinton" : string
```

Real

The ML type for floating point numbers is real. For reasons that will be easier to understand when we come to type inference, ML requires a decimal point in real constants:

```
1.0, 2.0, 3.14159, 4.44444, ... : real
```

The arithmetic operators +, -, and * may be applied to either integers or real numbers. Here are some example expressions and the resulting compiler output:

```
— 3+4;
val it = 7 : int
— 4.0 + 5.1;
val it = 9.1 : real
```

Notice that when + has two integer arguments, the result is an integer and when + has two real arguments, the result is a real. However, it is a type error to combine integer and real arguments. Here is part of the compiler output for an expression adding an integer to a real:

```
— 4+5.1;
stdIn:1.1-1.6 Error: operator and operand don't agree [literal]
  operator domain: int * int
```

```
operand:       int * real
```

This error message is telling us that since the first argument is an integer, the + symbol is considered to be integer addition. Therefore, the operator + has domain int * int, which is ML notation for the type of pairs of integers. However, the operand is applied to a pair of type int * real, which is ML notation for the type of pairs with one integer and one real.

Conversion from integer to real is done by explicit conversion function real. For example, the value of the expression real(3) is 3.0. Conversion from real to integer can be done using functions floor (round down), ceil (round up), round, and trunc.

Although arithmetic expressions in ML are a little more cumbersome than some other languages, the language is generally usable for most purposes. In part, explicit typing of numeric constants and explicit conversion is the price to pay for automatic type inference, a useful feature of ML described in Chapter 6.

Tuples

A tuple may be a pair, triple, quadruple, and so on. In ML, tuples may be formed of any types of values. Tuple values are written using parentheses and tuple types are written using *. For example, here is the compiler output for a pair, a triple, and a quadruple:

```
— (3,4);
val it = (3,4) : int * int
— (4,5,true);
val it = (4,5,true) : int * int * bool
— ("Bob", "Carol", "Ted", "Alice");
val it = ("Bob","Carol","Ted","Alice") : string * string * string * string
```

For all types τ_1 and τ_2, the type $\tau_1 * \tau_2$ is the type of pairs whose first component has type τ_1 and whose second component type τ_2. The type $\tau_1 * \tau_2 * \tau_3$ is a type of triples, $\tau_1 * \tau_2 * \tau_3 * \tau_4$ a type of quadruples, and so on.

Components of a tuple are accessed by functions that name the position of the desired component. For example, #1, selects the first component of any tuple, #2, the second component of any tuple with at least two components, and so on. Here are some examples:

```
— #2(3,4);
val it = 4 : int
— #3("John", "Paul", "George", "Ringo");
val it = "George" : string
```

Records

Like Pascal records and C structs, ML records are similar to tuples, but with named components. Record values and record types are written with curly braces, as follows:

```
- { First_name = "Donald", Last_name = "Knuth" };
```

```
val it = {First_name="Donald",Last_name="Knuth"}
  : {First_name:string, Last_name:string}
```

The expression here has two components, one called First_name and the other called Last_name. The type of this record tells us the type of each component. Record components can be accessed using # functions like tuples, but named according to the component names instead of position. Here is one example:

```
— #First_name( {First_name="Donald", Last_name="Knuth"} );
val it = "Donald" : string
```

Another way of selecting components of tuples and records is by pattern matching, described in Section 5.4.3.

Lists

ML lists can have any length, but all elements of a list must have the same type. Lists can be written by listing their elements, separated by commas, between square brackets. Here are some example lists of different types:

```
— [1,2,3,4];
val it = [1,2,3,4] : int list
— [true, false];
val it = [true,false] : bool list
— ["red", "yellow", "blue"];
val it = ["red","yellow","blue"] : string list
— [ fn x => x+1, fn x => x+2];
val it = [fn,fn] : (int -> int) list
```

For short lists, the compiler prints the elements of the list when showing that a list expression has been evaluated. For longer lists, the last elements are replaced by an ellipsis (three dots: …). As the last list example above shows, it is possible to write a list of functions.

In general, τ list is the type of all lists whose elements have type τ.

Like Lisp, the empty list is written nil in ML. List cons is an infix operator written as a pair of colons:

```
—  3 :: nil;
val it = [3] : int list
— 4 :: 5 :: it;
val it = [4,5,3] : int list
```

In the first list expression, 3 is “consed” onto the front of the empty list. The result is a list containing the single element 3. In the second expression, 4 and 5 are “consed” onto this list. In both cases, the result is an int list.

5.4.3 Patterns, declarations and function expressions

The declarations we have seen so far bind a value to a single identifier. One very convenient syntactic feature of ML is that declarations can also bind values to a set of identifiers, using patterns.

Value declarations

The general form of value declaration associates a value with a pattern. A pattern is an expression containing variables (such as x, y, z, …) and constants (such as true, false, 1, 2, 3, …), combined using certain forms such as tupling, record expressions, and a form of operation called a constructor. The general form of value declaration is

```
val <pattern> = <exp> ;
```

Where the common forms of patterns are summarized by the following grammar

```
<pattern> ::= <id> | <tuple> | <cons> | <record>  | <constr>
 <tuple>  ::= (<pattern>, ..., <pattern>)
<cons>  ::=  <pattern>::pattern
 <record>  ::= {<id>=<pattern>, ..., <id>=<pattern>}
 <constr>  ::= <id>(<pattern>, ..., <pattern>)
```

In words, a pattern can be an identifier, a tuple pattern, a list cons pattern, a record pattern, or a declared datatype constructor pattern. A tuple pattern is a sequence of patterns between parentheses, a list cons pattern is two patterns separated by double colons, a record pattern is a record-like expression with each field in the form of a pattern, and a constructor pattern is an identifier (a declared constructor) applied to the right number of pattern arguments. This BNF does not define the set of patterns exactly, since some conditions on patterns are not context free and therefore cannot be expressed using BNF. For example, the conditions that in a constructor pattern the identifier must be a declared constructor and that the constructor must be applied to the right number of pattern arguments are not context-free conditions. An additional condition on patterns, discussed below in connection with function declarations, is that no variable can occur twice in any pattern.

Since a variable is a pattern, a value declaration can simply associate a value with a variable. For example, here is a declaration binding a tuple to one identifier, followed by a declaration that uses a tuple pattern to bind components of the tuple:

```
— val t = (1,2,3);
val t = (1,2,3) : int * int * int
— val (x,y,z) = t;
val x = 1 : int
val y = 2 : int
val z = 3 : int
```

Notice that there are two lines of input in this example and four lines of compiler output. In the first declaration, the identifier t is bound to a tuple. In the second declaration, the tuple pattern (x,y,z) is given the value of t. Matching the pattern (x,y,z) against the triple t, identifier x gets value 1, identifier y gets value 2, and identifier z gets value 3.

Function declarations

The general form of a function declaration uses patterns. A single-clause definition has the form

```
fun f( <pattern> ) = <exp>
```

and a multiple-clause definition has the form

```
fun f( <pattern1> ) = <exp1> | ... |  f( <patternn> ) = <expn>
```

For example, a function adding its arguments can be written

```
fun f(x,y) = x + y;
```

Technically, the formal parameter of this function is a pattern (x, y) which must match the actual parameter on a call to f. The formal parameter to f is a tuple, which is broken down by pattern matching into its first and second components. You may think you are calling a function of two arguments. In reality, you are calling a function of one argument. That argument happens to be a pair of values. Pattern matching takes the tuple apart, binding x to what you might think is the first parameter, and y to the second.

An example using more than one clause is the following function computing the length of a list:

```
— fun length(nil) = 0
|  length( x :: xs ) = 1 + length(xs);
 val length = fn : 'a list → int
```

This code is explained below. The first two lines here are input (the declaration of function length) and the last line is the compiler output giving the type of this function. Here is an example application of length and the resulting value:

```
— length [“a”, “b”, “c”, “d”];
val it = 4 : int
```

When the function length is applied to an argument, the clauses are matched in the order they are written. If the argument matches the constant nil (i.e., the argument is the empty list), then the function returns the value 0, as specified by the first clause. Otherwise, the argument is matched against the pattern given in the second clause, (x::xs), and then the code for the second branch is executed. Because type checking guarantees that length will only be applied to a list, these two clauses cover all values that could possibly be passed to this function. The type of length, 'a list → int, will be explained in the next chapter.

In addition to declarations, ML has syntax for anonymous functions. We have already seen some simple examples. The general form allows the argument to be given by a pattern:

```
fn <pattern> => <exp>,
```

As mentioned briefly in passing in an earlier example, fn <pattern> => <exp> is like (lambda (<parameters>) (<exp>)) in Lisp. Here is an example, with compiler output.

```
— fn (x,y) => x+y;
val it = fn : int * int → int
```

The function expressed here takes a pair and adds its two components. The type of this function is int * int → int, meaning a function that maps a pair of integers to a single integer.

Here are some more examples illustrating other forms of patterns, each shown with associated compiler output:

```
— fun f(x, (y,z)) = y;
val f = fn : 'a * ('b * 'c) -> 'b
— fun g(x::y::z) = x::z;
val g = fn : 'a list -> 'a list
— fun h {a=x, b=y, c=z} = {d=y, e=z};
val h = fn : {a:'a, b:'b, c:'c} -> {d:'b, e:'c}
```

The first is a function on nested tuples, the second a function on lists that have at least two elements, and the third a function on records. The second declaration produces a compiler warning, since the function g is not defined for lists that have fewer than 2 elements.

Pattern-matching is applied in order. For example, when the function

```
fun f (x,0) = x
|   f (0,y) = y
|   f (x,y) = x+y;
```

is applied to an argument (a,b), the first clause is used if b=0, the second clause if b≠0 and a=0, and the third clause if b≠0 and a≠0. The ML type system will keep f from being applied to any argument that is not a pair (a,b).

An important condition on patterns is that no variable can occur twice in any pattern. For example, the following function declaration is not syntactically correct since the identifier x occurs twice in the pattern:

```
— fun eq(x,x) = true
    | eq(x,y) = false;
stdIn:24.5-25.20 Error: duplicate variable in pattern(s):
```

This function is not allowed since multiple occurrences of variables express equality and equality must be written explicitly into the body of a function.

5.4.4 ML datatype declaration

The ML datatype declaration is a special form of type declaration, declaring a type name and operations for building and making use of elements of the type. The ML datatype declaration has the syntactic form

```
datatype <type_name> = <constructor_clause> | ... | <constructor_clause>
```

where a constructor clause has the form

```
<constructor_clause> ::= <constructor> | <constructor> of <arg_types>
```

The idea is that each constructor clause tells one way to construct elements of the type. Elements of the type may be "deconstructed" into their constituent parts by pattern matching. This is illustrated by three examples that show some common ways of using datatype declarations in ML programs.

Example: An enumerated datatype

Types consisting of a finite set of tokens can be declared as ML datatypes. Here is a type consisting of three tokens, named to indicate three specific colors:

```
— datatype color = Red | Blue | Green;
datatype color = Blue | Green | Red
```

The compiler output, which looks just like the ML input code, indicates that the three elements of type color are Blue, Green, and Red. Technically, values Blue, Green, and Red are called *constructors*. They are called constructors because they are they the ways of constructing values with type color.

Example: A tagged union datatype

ML constructors can be declared so that they must be applied to arguments when constructing elements of the datatype. Constructors do not actually do anything to their arguments, other than to “tag” their arguments so that values constructed in different ways can be distinguished by pattern matching.

Suppose we are keeping student records, with names of BS students, names and undergraduate institutions of MS students, and names and faculty supervisors of PhD students. Then we could define a type student allowing these three forms of tuples as follows:

```
— datatype student = BS of name | MS of name*school | PhD of name*faculty;
```

In this datatype declaration, BS, MS and PhD are each constructors. However, unlike the color example, each student constructor must be applied to arguments to construct a value of type student. We must apply BS to a name, MS to a pair consisting of a name and a school, and PhD to a pair consisting of a name and a faculty name in order to produce a value of type student.

In effect, the type student is the union of three types

```
student ≈ union {name, name*school, name*faculty }
```

except that in ML "unions" (which are defined using datatype), each value of the union is tagged by a constructor that tells which of the constituent types the value comes from. This is illustrated in the following function, which returns the name of a student:

```
— fun name(BS(n)) = n
    | name(MS(n,s)) = n
    | name(PhD(n,f)) = n;
val name = fn : student → name
```

The first three lines are the declaration of the function name and the last line is the compiler output indicating that name is a function from students to names. The function has three clauses, one for each form of student.

Example: A recursive type

Datatype declaration may be recursive in that the type name may appear in one or more of the constructor argument types. Because of the way type recursion is implemented, ML datatype provides a convenient, high-level language construct that hides a common form of routine pointer manipulation.

The set of trees with integer labels at the leaves may be defined mathematically as follows:

> A *tree* is either
> a leaf, with an associated integer label, or
> a compound tree, consisting of a left subtree and a right subtree

This definition can be expressed as an ML datatype declaration, with each part of the definition corresponding to a clause of the datatype declaration:

```
datatype tree = LEAF of int | NODE of (tree * tree);
```

The identifiers LEAF and NODE are constructors, and the elements of the datatype are all values that can be produced by applying constructors to legal (type-correct) arguments. In words, a tree is either the result of applying the constructor LEAF to an integer (signifying a leaf with that integer label), or the result of applying the constructor NODE to two trees. These two trees, of course, must be produced similarly using constructors LEAF and NODE.

The function below shows how the constructors may be used to define a function on trees.

```
— fun inTree(x, LEAF(y)) = x = y
    |  inTree(x, NODE(y,z)) = inTree(x, y) orelse inTree(x, z);
val inTree = fn : int * tree → bool
```

This function looks for a specific integer value x in a tree. If the tree has the form LEAF(y), then x is in the tree only if x=y. If the tree has the form NODE(y,z), with subtrees y and z, then x is in the tree only if x is in the subtree y or the subtree z. The type output by

the compiler shows that inTree is a function that, given an integer and a tree, returns a Boolean value.

An example of a polymorphic datatype declaration appears in Section 6.5.3, after the discussion of polymorphism in Chapter 6.

5.4.5 ML reference cells and assignment

None of the ML constructs discussed in earlier sections of this chapter have side effects. Each expression has a value, but evaluating an expression does not have the side effect of changing the value of any other expression. While most large ML programs are written in a style that avoids side effects when possible, most large ML programs do use assignment occasionally to change the value of a variable.

The way that assignable variables are presented in ML is different from the way that assignable variables appear in other programming languages. The main reasons for this are to preserve the uniformity of ML as a programming language and to separate side effects from pure expressions as much as possible.

ML assignment is restricted to reference cells. In ML, a reference cell has a different type than immutable values such as integers, strings, lists, and so on. Since reference cells have specific reference types, restrictions on ML assignment are enforced as part of the type system. This is part of the elegance of ML: almost all restrictions on the structure of programs are part of the type system, and the type system has a systematic, uniform definition.

L-values and R-values

Before looking at assignment in ML, let us think about the difference between memory locations and their contents. This distinction is part of machine architectures (memory locations contain data) and relevant to many programming languages. The following pseudo-code fragment illustrates the idea:

```
x : int;
y : int;
x := y + 3;
```

In the assignment, the *value* stored in variable y is added to 3 and the result stored in the *location* for x. The central point is that the two variables are used differently. The command only uses the value stored in y and does not depend on the location of y. In contrast, the command uses the location of x, but does not depend on the value stored in x before the assignment occurs.

The location of a variable is called its *L-value* and the value stored in this location is called the *R-value* of the variable. This is standard terminology that you will see in many books on programming languages. The two values are called L and R to stand for “left” and “right”, since typically we use L-values on the left-hand sides of an assignment statement and R-values on the right-hand side.

ML reference cells

In ML, L-values and R-values have different types. In other words, an assignable region of memory has a different type than a value that cannot be changed. In ML, an L-value, or assignable region of memory, is called a *reference cell*. The type of a reference cell indicates that it is a reference cell and specifies the type of value that it contains. For example, a reference cell that contains an integer has type int ref, meaning an "integer reference cell."

When a reference cell is created, it must be initialized to a value of the correct type. Therefore, ML does not have uninitialized variables or dangling pointers. When an assignment changes the value stored in a reference cell, the assignment must be consistent with the type of the reference cell: an integer reference cell will always contain an integer, a list reference cell will always contain (or refer to) a list, and so on.

Operations on reference cells

ML has operations to create reference cells, to access their contents, and to change their contents. These are ref, ! and :=, which behave as follows:

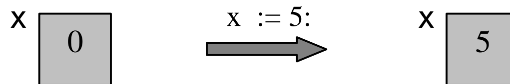
```
ref v  —  create a reference cell containing value v
! r    —  return the value contained in reference cell r
r := v —  place value v in reference cell r
```

Here are some examples:

```
— val x = ref 0;
val x = ref 0 : int ref
— x := 3*(!x) + 5;
val it = () : unit
— !x;
val it = 5 : int
```

The first input line binds identifier x to a new reference cell with contents 0. As the compiler output indicates, the value of x is this reference cell, which has type int ref. The next input line multiplies 3 times the contents of x, adds 5, and stores the resulting integer value in cell x. Since ML is expression oriented, this "statement" is an ML expression. The type of this expression is unit which, as described earlier, is the type used for expressions that are evaluated for side effect. The last input line is an expression reading the contents of x, which is the integer 5.

Since ML does not have any operations for computing the address of a value, there is no way to observe whether assignment is by value or by pointer. As a result, it is a convenient and accurate abstraction to regard a reference cell as a box holding a value of any size and regard assignment as an operation that places a value inside the box. For example, the code above that creates a reference cell named x and changes its contents can be visualized as follows, with the double arrow indicating changes as a result of assignment.



Since reference cells can be created for any type of value, we can define a string reference cell and change its contents by assignment.

```
— val y = ref "Apple";
val y = ref "Apple" : string ref
— y := "Fried green tomatoes";
val it = () : unit
— !y;
val it = "Fried green tomatoes" : string
```

As in the integer example, the associated reference cell can be visualized as a box that can contain any string.

As you know, different strings may require different amounts of memory. Therefore, it does not seem likely that the memory cell bound to y can hold any string of any length. In fact, when the declaration val y = ref "Apple" is processed, storage is allocated to contain the string “Apple” and the reference cell y is initialized to a pointer to this location. When the assignment y := "Fried green tomatoes" is executed, the contents of the cell y are changed to a pointer to “Fried green tomatoes”. Comparing the integer and string examples, ML assignment is implemented as ordinary value assignment for some types of cells and pointer assignment for others. However, since ML has no way of finding the address of an expression, this implementation difference is completely hidden from the programmer. If a compiler writer wanted to implement integer assignment as pointer assignment, all programs would behave in exactly the same way.

Here is one last simple code example to show how ML reference cells may be used in an iterative loop. This loop sums the numbers between 1 and 10.

```
val i = ref 0;
val j = ref 0;
while !i < 10 do (i := !i + 1;j := !j + !i);
!j;
```

In the first two lines, the identifiers i and j are bound to new reference cells initialized to value 0. The while loop increments i until !i, the contents of i, is not less than 10. The final expression reveals the final value of j, since the compiler prints the value of !j. Some

important details are that a test “i < 10” would not be legal, since this compares a reference cell to an integer. Similarly, “i := i+1” is not legal since a reference cell cannot be added to 1; only integers or real numbers can be added.

As illustrated in this example, two imperative expressions can be combined using a semicolon. Parentheses are used to keep the while loop above from parsing as a loop followed by j := !j+i. In fact, a semicolon can be used to combine any two expressions. The expression

```
e1; e2
```

is equivalent to

```
(fn x => e2) e1
```

where x is chosen not to appear in e2. As a result, the value of e1; e2 is the value of e2, after e1 has been evaluated.

Typing imperative operations: As mentioned above, reference cells have a different type than the values they contain. Here is the typing rule:

If expression e has type τ, then the expression ref e has type τ ref.

The function ! can be applied to any argument of type τ ref and assignment x := e is only type correct if x has type τ ref and e has type τ, for some type τ. In summary:

x : int — not assignable (like a constant in other languages)

y : int ref — assignable reference cell

5.4.6 ML Summary

ML is a programming language that encourages programming with functions. It is easy to define functions with function arguments and function return results. In addition, most data structures in ML programs are not assignable. Although it is possible to construct reference cells for any type of value and modify reference cells by assignment, side effects occur only when reference cells are used. While most large ML programs do use reference cells and side effects, the pure parts of ML are expressive enough that reference cells are used sparingly.

ML has an expressive type system. There are basic types for many common kinds of computable values, such as Booleans, integers, strings, and reals. There are also *type constructors*, which are type operators that can be applied to any type. The type constructors include tuples, records, and lists. In ML, it is possible to define tuples of lists of functions, for example. There is no restriction on the types of values that can be placed in data structures.

The ML type system is often called a “strong type system,” since every expression has a type and there are no mechanisms for subverting the type system. When the ML type checker determines that an expression has type int, for example, then any successful evaluation of that expression is guaranteed to produce an integer. There are no dangling

pointers that refer to unallocated locations in memory and no casts that allow values of one type to be treated as values of another type without conversion.

ML has several forms that allow programmers to define their own types and type constructors. In this chapter, we looked at datatype declarations, which can be used to define ML versions of enumerated types (types consisting of a finite list of values), disjoint unions (types whose elements are drawn from the union of two or more types), and recursively-defined types. Another important aspect of the ML type system is polymorphism, which we will study in the next chapter, along with other aspects of the ML type system. We will discuss additional type definition and module forms in Chapter 9.

5.5 **Chapter Summary**

In this chapter, we discussed some of the basic properties of Algol-like languages and examined some of the advances and problem areas in Algol 60, Algol 68, Pascal, and C. The Algol family of languages established the command-oriented syntax, with blocks, local declarations, and recursive functions, that are used in most current programming languages. The Algol family of languages are all statically typed, since each expression has a type that is determined by its syntactic form and the compiler checks before running the program to make sure that the types of operations and operands agree. In looking at the improvements from Algol 60 to Algol 68 to Pascal, we saw improvements in the static type systems.

The C programming language is similar to Algol 60, Algol 68, and Pascal in some respects: command-oriented syntax, blocks, local declarations, and recursive functions. However, C also shares some features with its untyped precursor BCPL, such as pointer arithmetic. C is also more restricted than most Algol-based languages in that functions cannot be declared inside nested blocks: all functions are declared outside the main program. This simplifies storage management for C, as we will see in Chapter 7.

In the second half of this chapter, we looked at the ML programming language in more detail than the Algol family of languages. One reason to study ML is that this language combines many if the important features of the Algol family with features of Lisp; this language provides a good summary of the important language features that developed prior to 1980.

The part of ML that we covered in this chapter comes from what is called “core ML.” This is ML without the module features that were added in the 1980s. Core ML has the following types

unit, Booleans, integers, strings, reals, tuples, lists, records

and the following constructs

patterns, declarations, functions, polymorphism, overloading, type declarations, reference cells, exceptions.We discussed most of these in this chapter, except polymorphism, which is covered in Chapter 6, and exceptions, which are studied in Chapter 8. We summarized the study of ML in this chapter in Section 5.4.6.

5.6 **Exercises**

(* Insert exercises from LaTeX files *)