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. Because 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 do 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. Compared with Fortran, Algol 60 provided better ways to

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

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 United Kingdom 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."

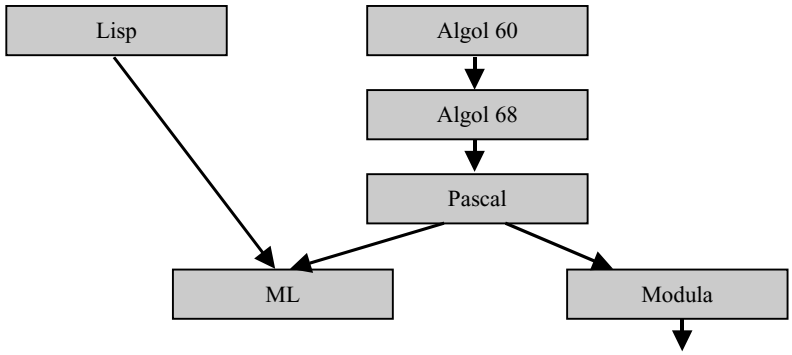
After approximately 20 years in Edinburgh, Robin returned to Cambridge University, where he held a Chair in Computer Science and was Head of the Computer Laboratory until his retirement in 1999.

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.

The following characteristics are some important features of Algol 60:

- 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

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press



- fewer ad hoc restrictions than previous languages. For example,
  - general expressions inside array indices
  - procedures that could be called with procedure parameters
- a primitive static type system, later improved on in Algol 68 and Pascal.

Here is an example program (subsequently explained) 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, note that the types of the parameters to the procedure (function) are declared in the line following the procedure name. Although 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 when a value is assigned 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 Algol 60, 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:

- The Algol 60 type discipline had some shortcomings. There are two examples that are illustrated in more detail in the exercises:

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

- 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*. Although 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 by-product 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 Subsection 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.

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

5.1.2 Algol 68

Algol 68 was intended to remove some of the difficulties found in Algol 60 and to improve the expressiveness of the language. However, in the end, the Algol 68 committee produced a design that was more problematic than that of Algol 60. One main problem is that, although programming in Algol 68 appears no more difficult than that of 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 with 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. (The general concepts of memory management, parameter passing, and related issues are covered in Chapter 7.) Algol 68 memory management involves a stack for local variables and heap storage for data that are intended to live beyond the current function call. As in C, Algol 68 data on the heap are explicitly allocated, but unlike C, heap data are 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 that 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, who used the 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* (Prentice-Hall, 1975). 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

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

approximately a 20-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)
```

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

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)
```

as 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 be called only 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 allow array parameters of only a fixed length. Because 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 1970s. 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 Subsection 9.3.1 as part of the discussion of program modules.

5.2 THE DEVELOPMENT OF C

Although 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, which is unrelated to the design of the language itself, is the popularity of the Unix operating system, which was written in C. When programs are written 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.

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

C was originally designed and implemented from 1969 to 1973, 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's 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 from 1977 to 1979, as part of a push to achieve portability of the Unix system, and in the mid-1980s when committee of the American National Standards Institute (ANSI) 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 whereas 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. Another consequence is that, because 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`.

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

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 to make an allowance for existing code in a typeless language. As C evolved further and was later standardized by an ANSI committee, backward compatibility with the 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, ACM 1993), "C is quirky, flawed, and an enormous success." Although 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 that of the Algol family with approximately the same degree of ease as for 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 is to examine concepts common to Algol-like languages, Lisp-like languages, and concurrent and object-oriented extensions of these languages. Therefore, we focus primarily on the core fragment of ML.

The following list enumerates our main reasons for looking at ML in some detail:

- 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.
- Because most readers are familiar with C and many have not written a lot of programs in significantly different languages, it is useful to have a language other than C to use for examples in the following chapters.

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

- 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 such as "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, as many type declarations are automatically deduced by the compiler, and flexible, as 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.

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 semiautomated manner. His LCF project started at Stanford in 1970 and continued at Edinburgh through the 1980s, making substantial progress toward 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. Because a tactic may search indefinitely or reach some situation in which 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:

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

We may use the concept of tactic to understand some basic properties of the ML programming language. Because 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

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

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, because a tactic is a function from formulas to proofs, the type of a tactic would be a function type:

```
tactic : formula ⟶ proof
```

However, this type seems to say that if a tactic is applied to a function, the result will be a proof. However, 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 that maps A to B, written as

```
f : A ⟶ B
```

in ML, means that, 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: Because 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(tactic₁, tactic₂) =
        λformula. try tactic₁(formula)
                  else tactic₂(formula)
```

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

5.4 THE ML PROGRAMMING LANGUAGE

Because ML is used in the next few chapters of the book to illustrate properties of programming languages, we will study ML in a little more detail than we will study some other languages. The version of ML that we will use is called Standard ML 97 (SML97). Compilers for SML97 are available on the Internet without charge.

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

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 preceding lines 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

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

ill-typed expression

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

for example, parses correctly because this has the correct form for an if-then-else. However, the type checker rejects this expression because the ML type checker requires the then and else parts of an if-then-else expression to have the same types, as described in the next subsection. 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. Because 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 note that integer division in ML is written as 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.

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

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 with 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 by a val declaration, written as

```
— 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 $\lambda x.\ x + 5$ in lambda calculus. We will discuss function declarations and function expressions further in Subsection 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 Subsection 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

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

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 parts of ML are best summarized by a list of 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. The type 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, whereas 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 because 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, as 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, as a conditional expression must have a value whether

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

the test is true or false. For example, an expression

```
— val nonsense = if a then 3;
```

is not legal ML because 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, and NOT in Pascal or &&, ||, and ! in C, with some minor differences. Negation is written as not, conjunction (and) is written as andalso and disjunction (or) is written as 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 the 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 the 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.

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

Strings

Strings are written as a sequence of symbols between double quotes:

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

String concatenation is written as ^, 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
```

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

```
— 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, because 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 number.

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

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

Although arithmetic expressions in ML are a little more cumbersome than in 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 that is 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 with parentheses and tuple types are written with *. 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 has type τ_2. The type $\tau_1{}^*\tau_2{}^*\tau_3$ is a type of triples, the type $\tau_1{}^*\tau_2{}^*\tau_3{}^*\tau_4$ 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 selects 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}
```

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

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 with # functions like tuples, but are 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, which is described in Subsection 5.4.3.

Lists

ML lists can have any length, but all elements of a list must have the same type. We can write lists by listing their elements, separated by commas, between 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 with an ellipsis (three dots; . . .). As the last list example above shows, it is possible to write a list of functions.

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

As in 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.

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

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 by 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 by 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 data-type 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 recordlike 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, as some conditions on patterns are not context free and therefore cannot be expressed by 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, subsequently discussed in connection with function declarations, is that no variable can occur twice in any pattern.

Because a variable is a pattern, a value declaration can simply associate a value with a variable. For example, here is a declaration that binds 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
```

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

Note 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. When the pattern (x,y,z) is matched 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 as

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

Technically, the formal parameter of this function is a pattern (x, y) that 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 in which more than one clause is used is the following function, which computes the length of a list:

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

This code is subsequently explained. 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

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

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 be applied only 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 an 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, as the function g is not defined for lists that have fewer than two 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;
```

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

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 because 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 because multiple occurrences of variables express equality, and equality must be written explicitly into the body of a function.

5.4.4 ML Data-Type Declaration

The ML data-type declaration is a special form of type declaration that declares a type name and operations for building and making use of elements of the type. The ML data-type 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 data-type declarations in ML programs.

Example. An Enumerated Data Type: Types consisting of a finite set of tokens can be declared as ML data types. 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 the ways of constructing values with type color.

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

Example. A Tagged Union Data Type: ML constructors can be declared so that they must be applied to arguments when constructing elements of the data type. 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 B.S. students, names and undergraduate institutions of M.S. students, and names and faculty supervisors of Ph.D. students. Then we could define a type student that allows these three forms of tuples as follows:

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

In this data-type declaration, BS, MS, and PhD are each constructors. However, unlike in 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 by 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: Data-type 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 data type 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.

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

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

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

The identifiers LEAF and NODE are constructors, and the elements of the data type are all values that can be produced by the application of 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 with constructors LEAF and NODE.

The following function 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 data-type declaration appears in Subsection 6.5.3, after the discussion of polymorphism in Section 6.4.

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. Although 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. Because 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.

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

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 pseudocode 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 uses only 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, as 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  — creates a reference cell containing value v
! r    — returns the value contained in reference cell r
r := v — places value v in reference cell r
```

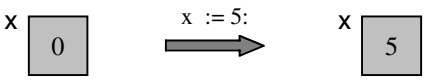
https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

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. Because 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.

Because 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 to regard assignment as an operation that places a value inside the box. For example, the preceding code 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:



Because 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.

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

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, because 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, because the compiler prints the value of !j. Some important details are that a test i < 10 would not be legal, as this compares a reference cell to an integer. Similarly, i := i+1 is not legal as 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 with a semicolon. Parentheses are used to keep the preceding while loop 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 previously mentioned, 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

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

type correct only 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. Although most large ML programs do use reference cells and have 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*, as 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 data-type 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 is all statically typed, as 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

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

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 we did 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 before 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. The study of ML was summarized in this chapter in Subsection 5.4.6.

EXERCISES

5.1 Algol 60 Procedure Types

In Algol 60, the type of each formal parameter of a procedure must be given. However, *proc* is considered a type (the type of procedures). This is much simpler than the ML types of function arguments. However, this is really a type loophole; because calls to procedure parameters are not fully type checked, Algol 60 programs may produce run-time type errors.

Write a procedure declaration for `Q` that causes the following program fragment to produce a run-time type error:

```
proc P (proc Q)
   begin Q(true) end;
P(Q);
```

where true is a Boolean value. Explain why the procedure is statically type correct, but produces a run-time type error. (You may assume that adding a Boolean to an integer is a run-time type error.)

5.2 Algol 60 Pass-By-Name

The following Algol 60 code declares a procedure `P` with one pass-by-name integer parameter. Explain how the procedure call `P(A[i])` changes the values of `i` and `A` by substituting the actual parameters for the formal parameters, according to the Algol

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

60 copy rule. What integer values are printed by tprogram? By using pass-by-name parameter passing?

The line integer x does not declare local variables – this is just Algol 60 syntax declaring the type of the procedure parameter:

```
begin
    integer i;
    integer array A[1:2];

    procedure P(x);
        integer x;
        begin
            i := x;
            x := i
        end

    i := 1;
    A[1] := 2; A[2] := 3;
    P (A[i]);
    print (i, A[1], A[2])
end
```

5.3 Nonlinear Pattern Matching

ML patterns cannot contain repeated variables. This exercise explores this language design decision.

A declaration with a single pattern is equivalent to a sequence of declarations using destructors. For example,

```
val p = (5,2);
val (x,y) = p;
```

is equivalent to

```
val p = (5,2);
val x = #1(p);
val y = #2(p);
```

where #1(p) is the ML expression for the first component of pair p and #2 similarly returns the second component of a pair. The operations #1 and #2 are called *destructors* for pairs.

A function declaration with more than one pattern is equivalent to a function declaration that uses standard if-then-else and destructors. For example,

```
fun f nil = 0
|   f (x::y) = x;
```

is equivalent to

```
fun f(z) = if z=nil then 0 else hd(z);
```

where hd is the ML function that returns the first element of a list.

Questions:

(a) Write a function declaration that does not use pattern matching and that is equivalent to

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

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

ML pattern matching is applied in order, so that when this function 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.

(b) Does the method you used in part (a), combining destructors and if-then-else, work for this function?

```
fun eq(x,x) = true
|   eq(x,y) = false;
```

(c) How would you translate ML functions that contain patterns with repeated variables into functions without patterns? Give a brief explanation of a general method and show the result for the function eq in part (b).

(d) Why do you think the designers of ML prohibited repeated variables in patterns? (*Hint:* If f, g : int ⟶ int, then the expression f = g is not type-correct ML as the test for equality is not defined on function types.)

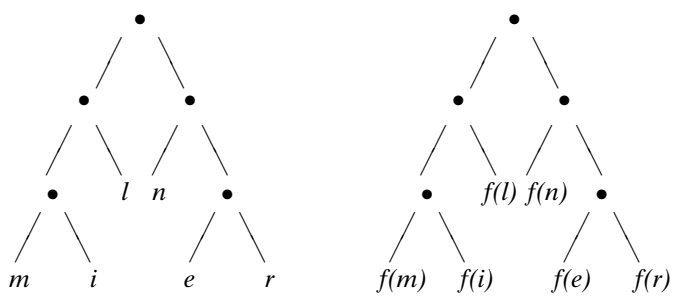
5.4 ML Map for Trees

(a) The binary tree data type

```
datatype 'a tree = LEAF of 'a |
                   NODE of 'a tree * 'a tree;
```

describes a binary tree for any type, but does not include the empty tree (i.e., each tree of this type must have at least a root node).

Write a function maptree that takes a function as an argument and returns a function that maps trees to trees by mapping the values at the leaves to new values, using the function passed in as a parameter. In more detail, if f is a function that can be applied to the leaves of tree t and t is the tree on the left, then maptree f t should result in the tree on the right:



For example, if f is the function fun f(x)=x+1 then

```
maptree f (NODE(NODE(LEAF 1,LEAF 2),LEAF 3))
```

should evaluate to NODE(NODE(LEAF 2,LEAF 3),LEAF 4). Explain your definition in one or two sentences.

(b) What is the type ML gives to your function? Why is it not the expected type ('a ⟶ 'a) ⟶ 'a tree ⟶ 'a tree?

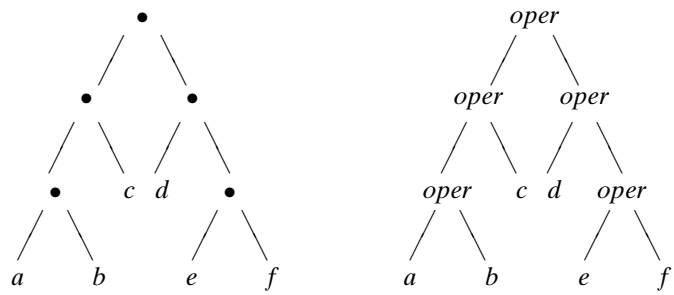
https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

5.5 ML Reduce for Trees

Assume that the data type tree is defined as in problem 4. Write a function

reduce : ('a * 'a → 'a) → 'a tree → 'a

that combines all the values of the leaves by using the binary operation passed as a parameter. In more detail, if oper : 'a * 'a → 'a and t is the nonempty tree on the left in this picture,



then reduce oper t should be the result we obtain by evaluating the tree on the right. For example, if f is the function,

fun f(x : int, y : int) = x + y;

then reduce f(NODE(NODE(LEAF 1, LEAF 2), LEAF 3)) = (1 + 2) + 3 = 6. Explain your definition of reduce in one or two sentences.

5.6 Currying

This problem asks you to show that the ML types 'a → ('b → 'c) and ('a * 'b) → 'c are essentially equivalent.

(a) Define higher-order ML functions

Curry: (('a * 'b) → 'c)→ ('a → ('b→ 'c))

and

UnCurry: ('a → ('b → 'c))→ (('a * 'b) → 'c)

(b) For all functions f : ('a * 'b) → 'c and g : 'a → ('b → 'c), the following two equalities should hold (if you wrote the right functions):

UnCurry(Curry(f)) = f
Curry(UnCurry(g)) = g

Explain why each is true for the functions you have written. Your answer can be three or four sentences long. Try to give the main idea in a clear, succinct way. (We are more interested in insight than in number of words.) Be sure to consider termination behavior as well.

5.7 Disjoint Unions

A *union type* is a type that allows the values from two different types to be combined in a single type. For example, an expression of type union(A, B) might have a value of type A or a value of type B. The languages C and ML both have forms of union types.

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

(a) Here is a C program fragment written with a union type:

```
...
union IntString {
   int i;
   char *s;
} x;
int y;
if ( ... ) x.i = 3 else x.s = "here, fido";
...
y = (x.i) + 5;
...
```

A C compiler will consider this program to be well typed. Despite the fact that the program type checks, the addition may not work as intended. Why not? Will the run-time system catch the problem?

(b) In ML, a union type union(A,B) would be written in the form datatype UnionAB = tag_a of A | tag_b of B and the preceding if statement could be written as

```
datatype IntString = tag_int of int | tag_str of string;
...
val x = if ... then tag_int(3) else tag_str("here, fido");
...
let val tag_int (m) = x in m + 5 end;
```

Can the same bug occur in this program? Will the run-time system catch the problem? The use of tags enables the compiler to give a useful warning message to the programmer, thereby helping the programmer to avoid the bug, even before running the program. What message is given and how does it help?

5.8 Lazy Evaluation and Functions

It is possible to evaluate function arguments at the time of the call (*eager evaluation*) or at the time they are used (*lazy evaluation*). Most programming languages (including ML) use eager evaluation, but we can simulate lazy evaluation in an eager language such as ML by using higher-order functions.

Consider a *sequence* data structure that starts with a known value and continues with a function (known as a *thunk*) to compute the rest of the sequence:

```
— datatype 'a Seq = Nil
                  | Cons of 'a * (unit -> 'a Seq);

— fun head (Cons (x, _)) = x;
val head = fn : 'a Seq -> 'a

— fun tail (Cons (_, xs)) = xs();
val tail = fn : 'a Seq -> 'a Seq

— fun BadCons (x, xs) = Cons (x, fn() =>xs);
val BadCons = fn : 'a * 'a Seq -> 'a Seq
```

Note that BadCons does not actually work, as xs was already evaluated on entering the function. Instead of calling BadCons(x, xs), you would need to use Cons(x, fn() =>xs) for lazy evaluation.

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

This lazy sequence data type provides a way to create infinite sequences, with each infinite sequence represented by a function that computes the next element in the sequence. For example, here is the sequence of infinitely many 1s:

```
— val ones = let fun f() = Cons(1,f) in f() end;
```

We can see how this works by defining a function that gets the *n*th element of a sequence and by looking at some elements of our infinite sequence:

```
— fun get(n,s) = if n=0 then head(s) else get(n-1,tail(s));
val get = fn : int * 'a Seq -> 'a
— get(0,ones);
val it = 1 : int
— get(5,ones);
val it = 1 : int
— get(245, ones);
val it = 1 : int
```

We can define the infinite sequence of all natural numbers by

```
— val natseq = let fun f(n)() = Cons(n,f(n+1)) in f(0) () end;
```

Using sequences, we can represent a function as a potentially infinite sequence of ordered pairs. Here are two examples, written as infinite lists instead of as ML code (note that ∼ is a negative sign in ML):

```
  add1 = (0, 1) :: (∼ 1, 0) :: (1, 2) :: (∼ 2, ∼ 1) :: (2, 3) :: . . .
double = (0, 0) :: (∼ 1, ∼ 2) :: (1, 2) :: (∼ 2, ∼ 4) :: (2, 4) :: . . .
```

Here is ML code that constructs the infinite sequences and tests this representation of functions by applying the sequences of ordered pairs to sample function arguments.

```
— fun make_ints(f)=
    let
      fun make_pos (n) = Cons( (n, f(n)), fn()=>make_pos(n + 1))
      fun make_neg (n) = Cons( (n, f(n)), fn()=>make_neg(n - 1))
    in
      merge (make_pos (0), make_neg(~1))
    end;
val make_ints = fn : (int -> 'a) -> (int * 'a) Seq

— val add1 = make_ints (fn(x) => x+1);
val add1 = Cons ((0,1),fn) : (int * int) Seq

— val double = make_ints (fn(x) => 2*x);
val double = Cons ((0,0),fn) : (int * int) Seq

— fun apply (Cons( (x1,fx1), xs) , x2) =
    if (x1=x2) then fx1
      else apply(xs(), x2);
val apply = fn : (''a * 'b) Seq * ''a -> 'b

— apply(add1, ~4);
val it = ~3 : int
```

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press

```
— apply(double, 7);
val it = 14 : int
```

(a) Write *merge* in ML. Merge should take two sequences and return a sequence containing the values in the original sequences, as used in the `make_ints` function.

(b) Using the representation of functions as a potentially infinite sequence of ordered pairs, write *compose* in ML. Compose should take a function f and a function g and return a function h such that $h(x) = f(g(x))$.

(c) It is possible to represent a partial function whose domain is not the entire set of integers as a sequence. Under what conditions will your *compose* function not halt? Is this acceptable?

https://doi.org/10.1017/CBO9780511804175.006 Published online by Cambridge University Press