1 Introduction to Programming Concepts

There is no royal road to geometry.
— Euclid's reply to Ptolemy, Euclid (*fl. c.* 300 B.C.)

Just follow the yellow brick road.
— *The Wonderful Wizard of Oz*, L. Frank Baum (1856–1919)

Programming is telling a computer how it should do its job. This chapter gives a gentle, hands-on introduction to many of the most important concepts in programming. We assume you have had some previous exposure to computers. We use the interactive interface of Mozart to introduce programming concepts in a progressive way. We encourage you to try the examples in this chapter on a running Mozart system.

This introduction only scratches the surface of the programming concepts we will see in this book. Later chapters give a deep understanding of these concepts and add many other concepts and techniques.

1.1 A calculator

Let us start by using the system to do calculations. Start the Mozart system by typing:

```
oz
```

or by double-clicking a Mozart icon. This opens an editor window with two frames. In the top frame, type the following line:

```
{Browse 9999*9999}
```

Use the mouse to select this line. Now go to the `Oz` menu and select `Feed Region`. This feeds the selected text to the system. The system then does the calculation `9999*9999` and displays the result, `99980001`, in a special window called the browser. The curly braces `{ ... }` are used for a procedure or function call. `Browse` is a procedure with one argument, which is called as `{Browse X}`. This opens the browser window, if it is not already open, and displays `X` in it.

1.2 Variables

While working with the calculator, we would like to remember an old result, so that we can use it later without retyping it. We can do this by declaring a variable:

```
declare
V=9999*9999
```

This declares `V` and binds it to `99980001`. We can use this variable later on:

```
{Browse V*V}
```

This displays the answer `9996000599960001`. Variables are just shortcuts for values. They cannot be assigned more than once. But you can declare another variable with the same name as a previous one. The previous variable then becomes inaccessible. Previous calculations that used it are not changed. This is because there are two concepts hiding behind the word "variable":

- The identifier. This is what you type in. Variables start with a capital letter and can be followed by any number of letters or digits. For example, the character sequence `Var1` can be a variable identifier.
- The store variable. This is what the system uses to calculate with. It is part of the system's memory, which we call its store.

The **`declare`** statement creates a new store variable and makes the variable identifier refer to it. Previous calculations using the same identifier are not changed because the identifier refers to another store variable.

1.3 Functions

Let us do a more involved calculation. Assume we want to calculate the factorial function $n!$, which is defined as $1 \times 2 \times \cdots \times (n-1) \times n$. This gives the number of permutations of n items, i.e., the number of different ways these items can be put in a row. Factorial of 10 is:

```
{Browse 1*2*3*4*5*6*7*8*9*10}
```

This displays `3628800`. What if we want to calculate the factorial of 100? We would like the system to do the tedious work of typing in all the integers from 1 to 100. We will do more: we will tell the system how to calculate the factorial of any n. We do this by defining a function:

```
declare
fun {Fact N}
   if N==0 then 1 else N*{Fact N-1} end
end
```

The **`declare`** statement creates the new variable `Fact`. The **`fun`** statement defines a function. The variable `Fact` is bound to the function. The function has one

argument `N`, which is a local variable, i.e., it is known only inside the function body. Each time we call the function a new local variable is created.

Recursion

The function body is an instruction called an **`if`** expression. When the function is called, then the **`if`** expression does the following steps:

- It first checks whether `N` is equal to 0 by doing the test `N==0`.
- If the test succeeds, then the expression after the **`then`** is calculated. This just returns the number 1. This is because the factorial of 0 is 1.
- If the test fails, then the expression after the **`else`** is calculated. That is, if `N` is not 0, then the expression `N*{Fact N-1}` is calculated. This expression uses `Fact`, the very function we are defining! This is called recursion. It is perfectly normal and no cause for alarm.

`Fact` uses the following mathematical definition of factorial:

$0! = 1$

$n! = n \times (n-1)!$ if $n > 0$

This definition is recursive because the factorial of `N` is `N` times the factorial of `N-1`. Let us try out the function `Fact`:

```
{Browse {Fact 10}}
```

This should display `3628800` as before. This gives us confidence that `Fact` is doing the right calculation. Let us try a bigger input:

```
{Browse {Fact 100}}
```

This will display a huge number (which we show in groups of five digits to improve readability):

```
                                                    933 26215
    44394 41526 81699 23885 62667 00490 71596 82643 81621 46859
    29638 95217 59999 32299 15608 94146 39761 56518 28625 36979
    20827 22375 82511 85210 91686 40000 00000 00000 00000 00000
```

This is an example of arbitrary precision arithmetic, sometimes called "infinite precision," although it is not infinite. The precision is limited by how much memory your system has. A typical low-cost personal computer with 256 MB of memory can handle hundreds of thousands of digits. The skeptical reader will ask: is this huge number really the factorial of 100? How can we tell? Doing the calculation by hand would take a long time and probably be incorrect. We will see later on how to gain confidence that the system is doing the right thing.

Combinations

Let us write a function to calculate the number of combinations of k items taken from n. This is equal to the number of subsets of size k that can be made from a set of size n. This is written $\binom{n}{k}$ in mathematical notation and pronounced "n choose k." It can be defined as follows using the factorial:

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

which leads naturally to the following function:

```
declare
fun {Comb N K}
   {Fact N} div ({Fact K}*{Fact N-K})
end
```

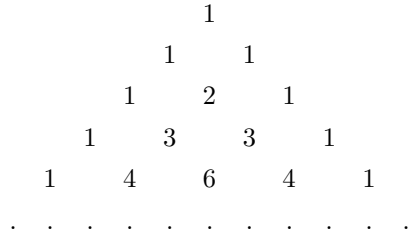
For example, `{Comb 10 3}` is 120, which is the number of ways that 3 items can be taken from 10. This is not the most efficient way to write `Comb`, but it is probably the simplest.

Functional abstraction

The definition of `Comb` uses the existing function `Fact` in its definition. It is always possible to use existing functions when defining new functions. Using functions to build abstractions is called functional abstraction. In this way, programs are like onions, with layers upon layers of functions calling functions. This style of programming is covered in chapter 3.

1.4 Lists

Now we can calculate functions of integers. But an integer is really not very much to look at. Say we want to calculate with lots of integers. For example, we would like to calculate Pascal's triangle[1]:



1. Pascal's triangle is a key concept in combinatorics. The elements of the nth row are the combinations $\binom{n}{k}$, where k ranges from 0 to n. This is closely related to the binomial theorem, which states $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{(n-k)}$ for integer $n \geq 0$.

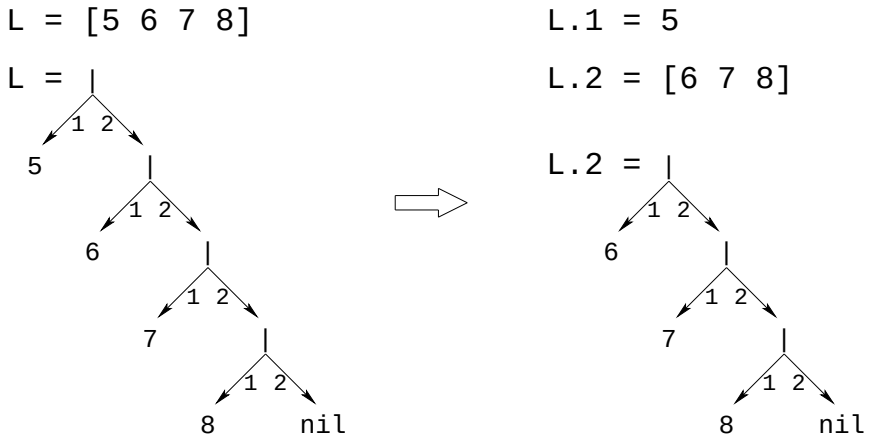


Figure 1.1: Taking apart the list `[5 6 7 8]`.

This triangle is named after scientist and philosopher Blaise Pascal. It starts with 1 in the first row. Each element is the sum of the two elements just above it to the left and right. (If there is no element, as on the edges, then zero is taken.) We would like to define one function that calculates the whole nth row in one swoop. The nth row has n integers in it. We can do it by using lists of integers.

A list is just a sequence of elements, bracketed at the left and right, like `[5 6 7 8]`. For historical reasons, the empty list is written `nil` (and not `[]`). Lists can be displayed just like numbers:

```
{Browse [5 6 7 8]}
```

The notation `[5 6 7 8]` is a shortcut. A list is actually a chain of links, where each link contains two things: one list element and a reference to the rest of the chain. Lists are always created one element at a time, starting with `nil` and adding links one by one. A new link is written `H|T`, where `H` is the new element and `T` is the old part of the chain. Let us build a list. We start with `Z=nil`. We add a first link `Y=7|Z` and then a second link `X=6|Y`. Now `X` references a list with two links, a list that can also be written as `[6 7]`.

The link `H|T` is often called a cons, a term that comes from Lisp.[2] We also call it a list pair. Creating a new link is called consing. If `T` is a list, then consing `H` and `T` together makes a new list `H|T`:

2. Much list terminology was introduced with the Lisp language in the late 1950s and has stuck ever since [137]. Our use of the vertical bar comes from Prolog, a logic programming language that was invented in the early 1970s [45, 201]. Lisp itself writes the cons as `(H . T)`, which it calls a dotted pair.

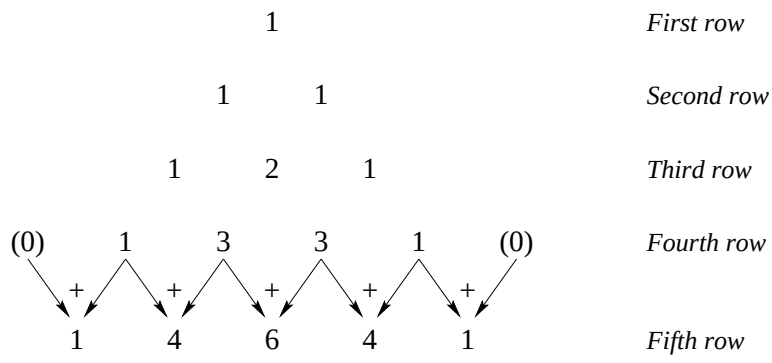


Figure 1.2: Calculating the fifth row of Pascal's triangle.

```
declare
H=5
T=[6 7 8]
{Browse H|T}
```

The list `H|T` can be written `[5 6 7 8]`. It has head `5` and tail `[6 7 8]`. The cons `H|T` can be taken apart, to get back the head and tail:

```
declare
L=[5 6 7 8]
{Browse L.1}
{Browse L.2}
```

This uses the dot operator "`.`", which is used to select the first or second argument of a list pair. Doing `L.1` gives the head of `L`, the integer `5`. Doing `L.2` gives the tail of `L`, the list `[6 7 8]`. Figure 1.1 gives a picture: `L` is a chain in which each link has one list element and `nil` marks the end. Doing `L.1` gets the first element and doing `L.2` gets the rest of the chain.

Pattern matching

A more compact way to take apart a list is by using the **`case`** instruction, which gets both head and tail in one step:

```
declare
L=[5 6 7 8]
case L of H|T then {Browse H} {Browse T} end
```

This displays `5` and `[6 7 8]`, just like before. The **`case`** instruction declares two local variables, `H` and `T`, and binds them to the head and tail of the list `L`. We say the **`case`** instruction does pattern matching, because it decomposes `L` according to the "pattern" `H|T`. Local variables declared with a **`case`** are just like variables declared with **`declare`**, except that the variable exists only in the body of the **`case`** statement, i.e., between the **`then`** and the **`end`**.

1.5 Functions over lists

Now that we can calculate with lists, let us define a function, `{Pascal N}`, to calculate the nth row of Pascal's triangle. Let us first understand how to do the calculation by hand. Figure 1.2 shows how to calculate the fifth row from the fourth. Let us see how this works if each row is a list of integers. To calculate a row, we start from the previous row. We shift it left by one position and shift it right by one position. We then add the two shifted rows together. For example, take the fourth row:

```
[1   3   3   1]
```

We shift this row left and right and then add them together element by element:

```
  [1   3   3   1   0]
+ [0   1   3   3   1]
```

Note that shifting left adds a zero to the right and shifting right adds a zero to the left. Doing the addition gives

```
[1   4   6   4   1]
```

which is the fifth row.

The main function

Now that we understand how to solve the problem, we can write a function to do the same operations. Here it is:

```
declare Pascal AddList ShiftLeft ShiftRight
fun {Pascal N}
   if N==1 then [1]
   else
     {AddList {ShiftLeft {Pascal N-1}} {ShiftRight {Pascal N-1}}}
   end
end
```

In addition to defining `Pascal`, we declare the variables for the three auxiliary functions that remain to be defined.

The auxiliary functions

To solve the problem completely, we still have to define three functions: `ShiftLeft`, which shifts left by one position, `ShiftRight`, which shifts right by one position, and `AddList`, which adds two lists. Here are `ShiftLeft` and `ShiftRight`:

```
fun {ShiftLeft L}
   case L of H|T then
      H|{ShiftLeft T}
   else [0] end
end

fun {ShiftRight L} 0|L end
```

`ShiftRight` just adds a zero to the left. `ShiftLeft` traverses `L` one element at a time and builds the output one element at a time. We have added an **`else`** to the **`case`** instruction. This is similar to an **`else`** in an **`if`**: it is executed if the pattern of the **`case`** does not match. That is, when `L` is empty, then the output is `[0]`, i.e., a list with just zero inside.

Here is `AddList`:

```
fun {AddList L1 L2}
   case L1 of H1|T1 then
      case L2 of H2|T2 then
         H1+H2|{AddList T1 T2}
      end
   else nil end
end
```

This is the most complicated function we have seen so far. It uses two **`case`** instructions, one inside another, because we have to take apart two lists, `L1` and `L2`. Now we have the complete definition of `Pascal`. We can calculate any row of Pascal's triangle. For example, calling `{Pascal 20}` returns the 20th row:

```
[1 19 171 969 3876 11628 27132 50388 75582 92378
 92378 75582 50388 27132 11628 3876 969 171 19 1]
```

Is this answer correct? How can we tell? It looks right: it is symmetric (reversing the list gives the same list) and the first and second arguments are 1 and 19, which are right. Looking at figure 1.2, it is easy to see that the second element of the nth row is always $n-1$ (it is always one more than the previous row and it starts out zero for the first row). In the next section, we will see how to reason about correctness.

Top-down software development

Let us summarize the methodology we used to write `Pascal`:

- The first step is to understand how to do the calculation by hand.
- The second step is to write a main function to solve the problem, assuming that some auxiliary functions are known (here, `ShiftLeft`, `ShiftRight`, and `AddList`).
- The third step is to complete the solution by writing the auxiliary functions.

The methodology of first writing the main function and filling in the blanks afterward is known as *top-down* software development. It is one of the best known approaches to program design, but it gives only part of the story as we shall see.

1.6 Correctness

A program is correct if it does what we would like it to do. How can we tell whether a program is correct? Usually it is impossible to duplicate the program's calculation by hand. We need other ways. One simple way, which we used before, is to verify that the program is correct for outputs that we know. This increases confidence in the program. But it does not go very far. To prove correctness in general, we have to reason about the program. This means three things:

- We need a mathematical model of the operations of the programming language, defining what they should do. This model is called the language's semantics.
- We need to define what we would like the program to do. Usually, this is a mathematical definition of the inputs that the program needs and the output that it calculates. This is called the program's specification.
- We use mathematical techniques to reason about the program, using the semantics. We would like to demonstrate that the program satisfies the specification.

A program that is proved correct can still give incorrect results, if the system on which it runs is incorrectly implemented. How can we be confident that the system satisfies the semantics? Verifying this is a major undertaking: it means verifying the compiler, the run-time system, the operating system, the hardware, and the physics upon which the hardware is based! These are all important tasks, but they are beyond the scope of the book. We place our trust in the Mozart developers, software companies, hardware manufacturers, and physicists.[3]

Mathematical induction

One very useful technique is mathematical induction. This proceeds in two steps. We first show that the program is correct for the simplest case. Then we show that, if the program is correct for a given case, then it is correct for the next case. If we can be sure that all cases are eventually covered, then mathematical induction lets us conclude that the program is always correct. This technique can be applied for integers and lists:

- For integers, the simplest case is 0 and for a given integer n the next case is $n+1$.
- For lists, the simplest case is `nil` (the empty list) and for a given list `T` the next case is `H|T` (with no conditions on `H`).

Let us see how induction works for the factorial function:

- `{Fact 0}` returns the correct answer, namely 1.

3. Some would say that this is foolish. Paraphrasing Thomas Jefferson, they would say that the price of correctness is eternal vigilance.

- Assume that `{Fact N-1}` is correct. Then look at the call `{Fact N}`. We see that the **`if`** instruction takes the **`else`** case (since `N` is not zero), and calculates `N*{Fact N-1}`. By hypothesis, `{Fact N-1}` returns the right answer. Therefore, assuming that the multiplication is correct, `{Fact N}` also returns the right answer.

This reasoning uses the mathematical definition of factorial, namely $n! = n \times (n-1)!$ if $n > 0$, and $0! = 1$. Later in the book we will see more sophisticated reasoning techniques. But the basic approach is always the same: start with the language semantics and problem specification, and use mathematical reasoning to show that the program correctly implements the specification.

1.7 Complexity

The `Pascal` function we defined above gets very slow if we try to calculate higher-numbered rows. Row 20 takes a second or two. Row 30 takes many minutes.[4] If you try it, wait patiently for the result. How come it takes this much time? Let us look again at the function `Pascal`:

```
fun {Pascal N}
   if N==1 then [1]
   else
     {AddList {ShiftLeft {Pascal N-1}} {ShiftRight {Pascal N-1}}}
   end
end
```

Calling `{Pascal N}` will call `{Pascal N-1}` two times. Therefore, calling `{Pascal 30}` will call `{Pascal 29}` twice, giving four calls to `{Pascal 28}`, eight to `{Pascal 27}`, and so forth, doubling with each lower row. This gives 2^{29} calls to `{Pascal 1}`, which is about half a billion. No wonder that `{Pascal 30}` is slow. Can we speed it up? Yes, there is an easy way: just call `{Pascal N-1}` once instead of twice. The second call gives the same result as the first. If we could just remember it, then one call would be enough. We can remember it by using a local variable. Here is a new function, `FastPascal`, that uses a local variable:

```
fun {FastPascal N}
   if N==1 then [1]
   else L in
      L={FastPascal N-1}
      {AddList {ShiftLeft L} {ShiftRight L}}
   end
end
```

We declare the local variable `L` by adding "`L in`" to the **`else`** part. This is just like using **`declare`**, except that the identifier can only be used between the **`else`** and the **`end`**. We bind `L` to the result of `{FastPascal N-1}`. Now we can use `L` wherever we need it. How fast is `FastPascal`? Try calculating row 30. This takes minutes

4. These times may vary depending on the speed of your machine.

with `Pascal`, but is done practically instantaneously with `FastPascal`. A lesson we can learn from this example is that using a good algorithm is more important than having the best possible compiler or fastest machine.

Run-time guarantees of execution time

As this example shows, it is important to know something about a program's execution time. Knowing the exact time is less important than knowing that the time will not blow up with input size. The execution time of a program as a function of input size, up to a constant factor, is called the program's *time complexity*. What this function is depends on how the input size is measured. We assume that it is measured in a way that makes sense for how the program is used. For example, we take the input size of `{Pascal N}` to be simply the integer `N` (and not, e.g., the amount of memory needed to store `N`).

The time complexity of `{Pascal N}` is proportional to 2^n. This is an exponential function in n, which grows very quickly as n increases. What is the time complexity of `{FastPascal N}`? There are n recursive calls and each call takes time proportional to n. The time complexity is therefore proportional to n^2. This is a polynomial function in n, which grows at a much slower rate than an exponential function. Programs whose time complexity is exponential are impractical except for very small inputs. Programs whose time complexity is a low-order polynomial are practical.

1.8 Lazy evaluation

The functions we have written so far will do their calculation as soon as they are called. This is called eager evaluation. There is another way to evaluate functions called lazy evaluation.[5] In lazy evaluation, a calculation is done only when the result is needed. This is covered in chapter 4 (see section 4.5). Here is a simple lazy function that calculates a list of integers:

```
fun lazy {Ints N}
   N|{Ints N+1}
end
```

Calling `{Ints 0}` calculates the infinite list `0|1|2|3|4|5|...`. This looks like an infinite loop, but it is not. The `lazy` annotation ensures that the function will only be evaluated when it is needed. This is one of the advantages of lazy evaluation: we can calculate with potentially infinite data structures without any loop boundary conditions. For example:

5. Eager and lazy evaluation are sometimes called data-driven and demand-driven evaluation, respectively.

```
L={Ints 0}
{Browse L}
```

This displays the following, i.e., nothing at all about the elements of `L`:

```
L<Future>
```

(The browser does not cause lazy functions to be evaluated.) The "`<Future>`" annotation means that `L` has a lazy function attached to it. If some elements of `L` are needed, then this function will be called automatically. Here is a calculation that needs an element of `L`:

```
{Browse L.1}
```

This displays the first element, namely 0. We can calculate with the list as if it were completely there:

```
case L of A|B|C|_ then {Browse A+B+C} end
```

This causes the first three elements of `L` to be calculated, and no more. What does it display?

Lazy calculation of Pascal's triangle

Let us do something useful with lazy evaluation. We would like to write a function that calculates as many rows of Pascal's triangle as are needed, but we do not know beforehand how many. That is, we have to look at the rows to decide when there are enough. Here is a lazy function that generates an infinite list of rows:

```
fun lazy {PascalList Row}
   Row|{PascalList
          {AddList {ShiftLeft Row} {ShiftRight Row}}}
end
```

Calling this function and browsing it will display nothing:

```
declare
L={PascalList [1]}
{Browse L}
```

(The argument `[1]` is the first row of the triangle.) To display more results, they have to be needed:

```
{Browse L.1}
{Browse L.2.1}
```

This displays the first and second rows.

Instead of writing a lazy function, we could write a function that takes `N`, the number of rows we need, and directly calculates those rows starting from an initial row:

```
fun {PascalList2 N Row}
   if N==1 then [Row]
   else
      Row|{PascalList2 N-1
             {AddList {ShiftLeft Row} {ShiftRight Row}}}
   end
end
```

We can display 10 rows by calling `{Browse {PascalList2 10 [1]}}`. But what if later on we decide that we need 11 rows? We would have to call `PascalList2` again, with argument 11. This would redo all the work of defining the first 10 rows. The lazy version avoids redoing all this work. It is always ready to continue where it left off.

1.9 Higher-order programming

We have written an efficient function, `FastPascal`, that calculates rows of Pascal's triangle. Now we would like to experiment with variations on Pascal's triangle. For example, instead of adding numbers to get each row, we would like to subtract them, exclusive-or them (to calculate just whether they are odd or even), or many other possibilities. One way to do this is to write a new version of `FastPascal` for each variation. But this quickly becomes tiresome. Is it possible to have just a single version that can be used for all variations? This is indeed possible. Let us call it `GenericPascal`. Whenever we call it, we pass it the customizing function (adding, exclusive-oring, etc.) as an argument. The ability to pass functions as arguments is known as higher-order programming.

Here is the definition of `GenericPascal`. It has one extra argument `Op` to hold the function that calculates each number:

```
fun {GenericPascal Op N}
   if N==1 then [1]
   else L in
      L={GenericPascal Op N-1}
      {OpList Op {ShiftLeft L} {ShiftRight L}}
   end
end
```

`AddList` is replaced by `OpList`. The extra argument `Op` is passed to `OpList`. `ShiftLeft` and `ShiftRight` do not need to know `Op`, so we can use the old versions. Here is the definition of `OpList`:

```
fun {OpList Op L1 L2}
   case L1 of H1|T1 then
      case L2 of H2|T2 then
         {Op H1 H2}|{OpList Op T1 T2}
      end
   else nil end
end
```

Instead of doing the addition `H1+H2`, this version does `{Op H1 H2}`.

Variations on Pascal's triangle

Let us define some functions to try out `GenericPascal`. To get the original Pascal's triangle, we can define the addition function:

```
fun {Add X Y} X+Y end
```

Now we can run `{GenericPascal Add 5}`.[6] This gives the fifth row exactly as before. We can define `FastPascal` using `GenericPascal`:

```
fun {FastPascal N} {GenericPascal Add N} end
```

Let us define another function:

```
fun {Xor X Y} if X==Y then 0 else 1 end end
```

This does an *exclusive-or* operation, which is defined as follows:

X	Y	{Xor X Y}
0	0	0
0	1	1
1	0	1
1	1	0

Exclusive-or lets us calculate the parity of each number in Pascal's triangle, i.e., whether the number is odd or even. The numbers themselves are not calculated. Calling `{GenericPascal Xor N}` gives the result:

```
                1
              1   1
            1   0   1
          1   1   1   1
        1   0   0   0   1
      1   1   0   0   1   1
. . . . . . . . . . . . .
```

Some other functions are given in the exercises.

1.10 Concurrency

We would like our program to have several independent activities, each of which executes at its own pace. This is called *concurrency*. There should be no interference

6. We can also call `{GenericPascal Number.´+´ 5}`, since the addition operation `´+´` is part of the module `Number`. But modules are not introduced in this chapter.

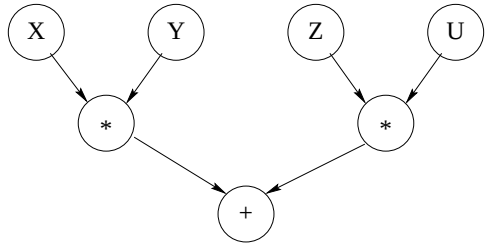


Figure 1.3: A simple example of dataflow execution.

among the activities, unless the programmer decides that they need to communicate. This is how the real world works outside of the system. We would like to be able to do this inside the system as well.

We introduce concurrency by creating threads. A thread is simply an executing program like the functions we saw before. The difference is that a program can have more than one thread. Threads are created with the **`thread`** instruction. Do you remember how slow the original `Pascal` function was? We can call `Pascal` inside its own thread. This means that it will not keep other calculations from continuing. They may slow down, if `Pascal` really has a lot of work to do. This is because the threads share the same underlying computer. But none of the threads will stop. Here is an example:

```
thread P in
   P={Pascal 30}
   {Browse P}
end
{Browse 99*99}
```

This creates a new thread. Inside this new thread, we call `{Pascal 30}` and then call `Browse` to display the result. The new thread has a lot of work to do. But this does not keep the system from displaying `99*99` immediately.

1.11 Dataflow

What happens if an operation tries to use a variable that is not yet bound? From a purely aesthetic point of view, it would be nice if the operation would simply wait. Perhaps some other thread will bind the variable, and then the operation can continue. This civilized behavior is known as dataflow. Figure 1.3 gives a simple example: the two multiplications wait until their arguments are bound and the addition waits until the multiplications complete. As we will see later in the book, there are many good reasons to have dataflow behavior. For now, let us see how dataflow and concurrency work together. Take, e.g.:

```
declare X in
thread {Delay 10000} X=99 end
{Browse start} {Browse X*X}
```

The multiplication `X*X` waits until `X` is bound. The first `Browse` immediately displays `start`. The second `Browse` waits for the multiplication, so it displays nothing yet. The `{Delay 10000}` call pauses for 10000 ms (i.e., 10 seconds). `X` is bound only after the delay continues. When `X` is bound, then the multiplication continues and the second browse displays 9801. The two operations `X=99` and `X*X` can be done in any order with any kind of delay; dataflow execution will always give the same result. The only effect a delay can have is to slow things down. For example:

```
declare X in
thread {Browse start} {Browse X*X} end
{Delay 10000} X=99
```

This behaves exactly as before: the browser displays 9801 after 10 seconds. This illustrates two nice properties of dataflow. First, calculations work correctly independent of how they are partitioned between threads. Second, calculations are patient: they do not signal errors, but simply wait.

Adding threads and delays to a program can radically change a program's appearance. But as long as the same operations are invoked with the same arguments, it does not change the program's results at all. This is the key property of dataflow concurrency. This is why dataflow concurrency gives most of the advantages of concurrency without the complexities that are usually associated with it. Dataflow concurrency is covered in chapter 4.

1.12 Explicit state

How can we let a function learn from its past? That is, we would like the function to have some kind of internal memory, which helps it do its job. Memory is needed for functions that can change their behavior and learn from their past. This kind of memory is called explicit state. Just like for concurrency, explicit state models an essential aspect of how the real world works. We would like to be able to do this in the system as well. Later in the book we will see deeper reasons for having explicit state (see chapter 6). For now, let us just see how it works.

For example, we would like to see how often the `FastPascal` function is used. Is there some way `FastPascal` can remember how many times it was called? We can do this by adding explicit state.

A memory cell

There are lots of ways to define explicit state. The simplest way is to define a single memory cell. This is a kind of box in which you can put any content. Many programming languages call this a "variable." We call it a "cell" to avoid confusion

with the variables we used before, which are more like mathematical variables, i.e., just shortcuts for values. There are three functions on cells: `NewCell` creates a new cell, `:=` (assignment) puts a new value in a cell, and `@` (access) gets the current value stored in the cell. Access and assignment are also called read and write. For example:

```
declare
C={NewCell 0}
C:=@C+1
{Browse @C}
```

This creates a cell `C` with initial content 0, adds one to the content, and displays it.

Adding memory to `FastPascal`

With a memory cell, we can let `FastPascal` count how many times it is called. First we create a cell outside of `FastPascal`. Then, inside of `FastPascal`, we add 1 to the cell's content. This gives the following:

```
declare
C={NewCell 0}
fun {FastPascal N}
   C:=@C+1
   {GenericPascal Add N}
end
```

(To keep it short, this definition uses `GenericPascal`.)

1.13 Objects

A function with internal memory is usually called an *object*. The extended version of `FastPascal` we defined in the previous section is an object. It turns out that objects are very useful beasts. Let us give another example. We will define a counter object. The counter has a cell that keeps track of the current count. The counter has two operations, `Bump` and `Read`, which we call its interface. `Bump` adds 1 and then returns the resulting count. `Read` just returns the count. Here is the definition:

```
declare
local C in
   C={NewCell 0}
   fun {Bump}
      C:=@C+1
      @C
   end
   fun {Read}
      @C
   end
end
```

The **`local`** statement declares a new variable `C` that is visible only up to the matching **`end`**. There is something special going on here: the cell is referenced

by a local variable, so it is completely invisible from the outside. This is called encapsulation. Encapsulation implies that users cannot mess with the counter's internals. We can guarantee that the counter will always work correctly no matter how it is used. This was not true for the extended `FastPascal` because anyone could look at and modify the cell.

It follows that as long as the interface to the counter object is the same, the user program does not need to know the implementation. The separation of interface and implementation is the essence of data abstraction. It can greatly simplify the user program. A program that uses a counter will work correctly for any implementation as long as the interface is the same. This property is called polymorphism. Data abstraction with encapsulation and polymorphism is covered in chapter 6 (see section 6.4).

We can bump the counter up:

```
{Browse {Bump}}
{Browse {Bump}}
```

What does this display? `Bump` can be used anywhere in a program to count how many times something happens. For example, `FastPascal` could use `Bump`:

```
declare
fun {FastPascal N}
   {Browse {Bump}}
   {GenericPascal Add N}
end
```

1.14 Classes

The last section defined one counter object. What if we need more than one counter? It would be nice to have a "factory" that can make as many counters as we need. Such a factory is called a *class*. Here is one way to define it:

```
declare
fun {NewCounter}
C Bump Read in
   C={NewCell 0}
   fun {Bump}
      C:=@C+1
      @C
   end
   fun {Read}
      @C
   end
   counter(bump:Bump read:Read)
end
```

`NewCounter` is a function that creates a new cell and returns new `Bump` and `Read` functions that use the cell. Returning functions as results of functions is another form of higher-order programming.

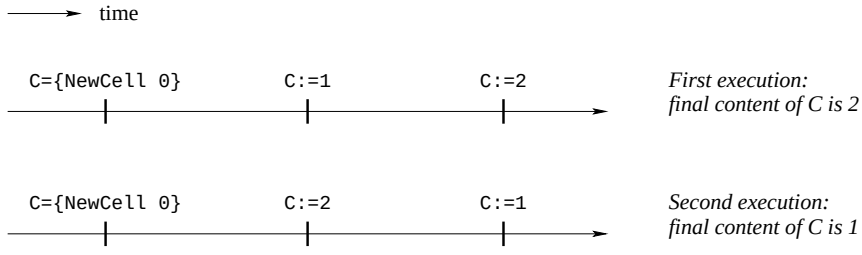


Figure 1.4: All possible executions of the first nondeterministic example.

We group the `Bump` and `Read` functions together into a record, which is a compound data structure that allows easy access to its parts. The record `counter(bump:Bump read:Read)` is characterized by its label `counter` and by its two fields, called `bump` and `read`. Let us create two counters:

```
declare
Ctr1={NewCounter}
Ctr2={NewCounter}
```

Each counter has its own internal memory and its own `Bump` and `Read` functions. We can access these functions by using the "`.`" (dot) operator. `Ctr1.bump` accesses the `Bump` function of the first counter. Let us bump the first counter and display its result:

```
{Browse {Ctr1.bump}}
```

Toward object-oriented programming

We have given an example of a simple class, `NewCounter`, that defines two operations, `Bump` and `Read`. Operations defined inside classes are usually called methods. The class can be used to make as many counter objects as we need. All these objects share the same methods, but each has its own separate internal memory. Programming with classes and objects is called object-based programming.

Adding one new idea, inheritance, to object-based programming gives object-oriented programming. Inheritance means that a new class can be defined in terms of existing classes by specifying just how the new class is different. For example, say we have a counter class with just a `Bump` method. We can define a new class that is the same as the first class except that it adds a `Read` method. We say the new class inherits from the first class. Inheritance is a powerful concept for structuring programs. It lets a class be defined incrementally, in different parts of the program. Inheritance is quite a tricky concept to use correctly. To make inheritance easy to use, object-oriented languages add special syntax for it. Chapter 7 covers object-oriented programming and shows how to program with inheritance.

1.15 Nondeterminism and time

We have seen how to add concurrency and state to a program separately. What happens when a program has both? It turns out that having both at the same time is a tricky business, because the same program can give different results from one execution to the next. This is because the order in which threads access the state can change from one execution to the next. This variability is called nondeterminism. Nondeterminism exists because we lack knowledge of the exact time when each basic operation executes. If we would know the exact time, then there would be no nondeterminism. But we cannot know this time, simply because threads are independent. Since they know nothing of each other, they also do not know which instructions each has executed.

Nondeterminism by itself is not a problem; we already have it with concurrency. The difficulties occur if the nondeterminism shows up in the program, i.e., if it is observable. An observable nondeterminism is sometimes called a race condition. Here is an example:

```
declare
C={NewCell 0}
thread
   C:=1
end
thread
   C:=2
end
```

What is the content of `C` after this program executes? Figure 1.4 shows the two possible executions of this program. Depending on which one is done, the final cell content can be either 1 or 2. The problem is that we cannot say which. This is a simple case of observable nondeterminism. Things can get much trickier. For example, let us use a cell to hold a counter that can be incremented by several threads:

```
declare
C={NewCell 0}
thread I in
   I=@C
   C:=I+1
end
thread J in
   J=@C
   C:=J+1
end
```

What is the content of `C` after this program executes? It looks like each thread just adds 1 to the content, making it 2. But there is a surprise lurking: the final content can also be 1! How is this possible? Try to figure out why before continuing.

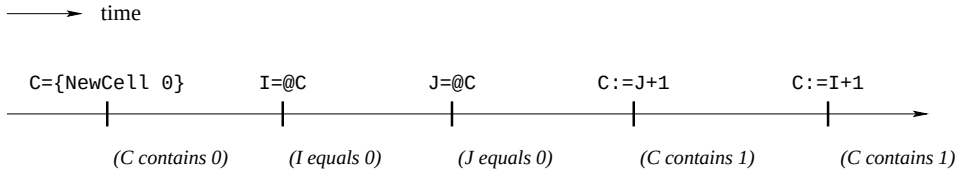


Figure 1.5: One possible execution of the second nondeterministic example.

Interleaving

The content can be 1 because thread execution is interleaved. That is, threads take turns each executing a little. We have to assume that any possible interleaving can occur. For example, consider the execution of figure 1.5. Both `I` and `J` are bound to 0. Then, since `I+1` and `J+1` are both 1, the cell gets assigned 1 twice. The final result is that the cell content is 1.

This is a simple example. More complicated programs have many more possible interleavings. Programming with concurrency and state together is largely a question of mastering the interleavings. In the history of computer technology, many famous and dangerous bugs were due to designers not realizing how difficult this really is. The Therac-25 radiation therapy machine is an infamous example. Because of concurrent programming errors, it sometimes gave its patients radiation doses that were thousands of times greater than normal, resulting in death or serious injury [128].

This leads us to a first lesson for programming with state and concurrency: if at all possible, do not use them together! It turns out that we often do not need both together. When a program does need to have both, it can almost always be designed so that their interaction is limited to a very small part of the program.

1.16 Atomicity

Let us think some more about how to program with concurrency and state. One way to make it easier is to use atomic operations. An operation is *atomic* if no intermediate states can be observed. It seems to jump directly from the initial state to the result state. Programming with atomic actions is covered in chapter 8.

With atomic operations we can solve the interleaving problem of the cell counter. The idea is to make sure that each thread body is atomic. To do this, we need a way to build atomic operations. We introduce a new language entity, called a lock, for this. A lock has an inside and an outside. The programmer defines the instructions that are inside. A lock has the property that only one thread at a time can be executing inside. If a second thread tries to get in, then it will wait until the first gets out. Therefore what happens inside the lock is atomic.

We need two operations on locks. First, we create a new lock by calling the function `NewLock`. Second, we define the lock's inside with the instruction **`lock`** `L` **`then ... end`**, where `L` is a lock. Now we can fix the cell counter:

```
declare
C={NewCell 0}
L={NewLock}
thread
   lock L then I in
      I=@C
      C:=I+1
   end
end
thread
   lock L then J in
      J=@C
      C:=J+1
   end
end
```

In this version, the final result is always 2. Both thread bodies have to be guarded by the same lock, otherwise the undesirable interleaving can still occur. Do you see why?

1.17 Where do we go from here?

This chapter has given a quick overview of many of the most important concepts in programming. The intuitions given here will serve you well in the chapters to come, when we define in a precise way the concepts and the computation models they are part of. This chapter has introduced the following computation models:

- Declarative model (chapters 2 and 3). Declarative programs define mathematical functions. They are the easiest to reason about and to test. The declarative model is important also because it contains many of the ideas that will be used in later, more expressive models.
- Concurrent declarative model (chapter 4). Adding dataflow concurrency gives a model that is still declarative but that allows a more flexible, incremental execution.
- Lazy declarative model (section 4.5). Adding laziness allows calculating with potentially infinite data structures. This is good for resource management and program structure.
- Stateful model (chapter 6). Adding explicit state allows writing programs whose behavior changes over time. This is good for program modularity. If written well, i.e., using encapsulation and invariants, these programs are almost as easy to reason about as declarative programs.
- Object-oriented model (chapter 7). Object-oriented programming is a programming style for stateful programming with data abstractions. It makes it easy to use

powerful techniques such as polymorphism and inheritance.

- Shared-state concurrent model (chapter 8). This model adds both concurrency and explicit state. If programmed carefully, using techniques for mastering interleaving such as monitors and transactions, this gives the advantages of both the stateful and concurrent models.

In addition to these models, the book covers many other useful models such as the declarative model with exceptions (section 2.7), the message-passing concurrent model (chapter 5), the relational model (chapter 9), and the specialized models of part II.

1.18 Exercises

1. *A calculator.* Section 1.1 uses the system as a calculator. Let us explore the possibilities:

   (a) Calculate the exact value of 2^{100} without using any new functions. Try to think of shortcuts to do it without having to type `2*2*2*...*2` with one hundred 2s. *Hint*: use variables to store intermediate results.

   (b) Calculate the exact value of 100! without using any new functions. Are there any possible shortcuts in this case?

2. *Calculating combinations.* Section 1.3 defines the function `Comb` to calculate combinations. This function is not very efficient because it might require calculating very large factorials. The purpose of this exercise is to write a more efficient version of `Comb`.

   (a) As a first step, use the following alternative definition to write a more efficient function:

   $$\binom{n}{k} = \frac{n \times (n-1) \times \cdots \times (n-k+1)}{k \times (k-1) \times \cdots \times 1}$$

   Calculate the numerator and denominator separately and then divide them. Make sure that the result is 1 when $k = 0$.

   (b) As a second step, use the following identity:

   $$\binom{n}{k} = \binom{n}{n-k}$$

   to increase efficiency even more. That is, if $k > n/2$, then do the calculation with $n - k$ instead of with k.

3. *Program correctness.* Section 1.6 explains the basic ideas of program correctness and applies them to show that the factorial function defined in section 1.3 is correct. In this exercise, apply the same ideas to the function `Pascal` of section 1.5 to show that it is correct.

4. *Program complexity.* What does section 1.7 say about programs whose time complexity is a high-order polynomial? Are they practical or not? What do you

think?

5. *Lazy evaluation.* Section 1.8 defines the lazy function `Ints` that lazily calculates an infinite list of integers. Let us define a function that calculates the sum of a list of integers:

```
fun {SumList L}
   case L of X|L1 then X+{SumList L1}
   else 0 end
end
```

What happens if we call `{SumList {Ints 0}}`? Is this a good idea?

6. *Higher-order programming.* Section 1.9 explains how to use higher-order programming to calculate variations on Pascal's triangle. The purpose of this exercise is to explore these variations.

(a) Calculate individual rows using subtraction, multiplication, and other operations. Why does using multiplication give a triangle with all zeros? Try the following kind of multiplication instead:

```
fun {Mul1 X Y} (X+1)*(Y+1) end
```

What does the 10th row look like when calculated with `Mul1`?

(b) The following loop instruction will calculate and display 10 rows at a time:

```
for I in 1..10 do {Browse {GenericPascal Op I}} end
```

Use this loop instruction to make it easier to explore the variations.

7. *Explicit state.* This exercise compares variables and cells. We give two code fragments. The first uses variables:

```
local X in
   X=23
   local X in
      X=44
   end
   {Browse X}
end
```

The second uses a cell:

```
local X in
   X={NewCell 23}
   X:=44
   {Browse @X}
end
```

In the first, the identifier `X` refers to two different variables. In the second, `X` refers to a cell. What does `Browse` display in each fragment? Explain.

8. *Explicit state and functions.* This exercise investigates how to use cells together with functions. Let us define a function `{Accumulate N}` that accumulates all its inputs, i.e., it adds together all the arguments of all calls. Here is an example:

```
{Browse {Accumulate 5}}
{Browse {Accumulate 100}}
{Browse {Accumulate 45}}
```

This should display 5, 105, and 150, assuming that the accumulator contains zero at the start. Here is a wrong way to write `Accumulate`:

```
declare
fun {Accumulate N}
Acc in
   Acc={NewCell 0}
   Acc:=@Acc+N
   @Acc
end
```

What is wrong with this definition? How would you correct it?

9. *Memory store.* This exercise investigates another way of introducing state: a memory store. The memory store can be used to make an improved version of `FastPascal` that remembers previously calculated rows.

(a) A memory store is similar to the memory of a computer. It has a series of memory cells, numbered from 1 up to the maximum used so far. There are four functions on memory stores: `NewStore` creates a new store, `Put` puts a new value in a memory cell, `Get` gets the current value stored in a memory cell, and `Size` gives the highest-numbered cell used so far. For example:

```
declare
S={NewStore}
{Put S 2 [22 33]}
{Browse {Get S 2}}
{Browse {Size S}}
```

This stores `[22 33]` in memory cell 2, displays `[22 33]`, and then displays 2. Load into the Mozart system the memory store as defined in the supplements file on the book's Web site. Then use the interactive interface to understand how the store works.

(b) Now use the memory store to write an improved version of `FastPascal`, called `FasterPascal`, that remembers previously calculated rows. If a call asks for one of these rows, then the function can return it directly without having to recalculate it. This technique is sometimes called memoization since the function makes a "memo" of its previous work. This improves its performance. Here's how it works:

- First make a store S available to `FasterPascal`.
- For the call `{FasterPascal N}`, let m be the number of rows stored in S, i.e., rows 1 up to m are in S.
- If $n > m$, then compute rows $m + 1$ up to n and store them in S.
- Return the nth row by looking it up in S.

Viewed from the outside, `FasterPascal` behaves identically to `FastPascal` except that it is faster.

(c) We have given the memory store as a library. It turns out that the memory store can be defined by using a memory cell. We outline how it can be done and you can write the definitions. The cell holds the store contents as a list of

the form `[N1|X1 ... Nn|Xn]`, where the cons `Ni|Xi` means that cell number `Ni` has content `Xi`. This means that memory stores, while they are convenient, do not introduce any additional expressive power over memory cells.

(d) Section 1.13 defines a counter object. Change your implementation of the memory store so that it uses this counter to keep track of the store's size.

10. *Explicit state and concurrency.* Section 1.15 gives an example using a cell to store a counter that is incremented by two threads.

(a) Try executing this example several times. What results do you get? Do you ever get the result 1? Why could this be?

(b) Modify the example by adding calls to `Delay` in each thread. This changes the thread interleaving without changing what calculations the thread does. Can you devise a scheme that always results in 1?

(c) Section 1.16 gives a version of the counter that never gives the result 1. What happens if you use the delay technique to try to get a 1 anyway?