# While and For Loops

Non-recursive functions encapsulate code and allow for control of execution flow. That is, if a piece of code needs to be executed many times, then we can encapsulate it in the body of a function and call this function several times. In this chapter, we will look at more general control of flow via loops and conditional execution. Recursion is another mechanism for controlling flow, but this is deferred to .

Many programming constructs need to be repeated, and F# contains many structures for repetition. A -loop has the following syntax:

/whileLoop.ebnf while <\*condition\*> do <\*expr\*> [\*done\*]

The <\*condition\*> is an expression that evaluates to true or false. A while-loop repeats the <\*expr\*> expression as long as the condition is true. Using lightweight syntax, the block following the keyword up to and including the keyword may be replaced by a newline and indentation.

The program in is an example of a while-loop which counts from 1 to 10. The variable i is customarily called the counter variable. The counting is done by performing the following computation: In line [[countWhileLoop]](#countWhileLoop), the counter variable is first given an initial value of 1. Then execution enters the while-loop and examines the condition. Since , the condition is true, and execution enters the body of the loop. The body prints the value of the counter to the screen and increases the counter by 1. Then execution returns to the top of the while-loop. Now the condition is , which is also true, and so execution enters the body and so on until the counter has reached the value 11, in which case the condition is false, and execution continues in line [[countWhileContinue]](#countWhileContinue).

In lightweight syntax, this would be as shown in . Notice that although the expression following the condition is preceded with a keyword, and do <\*expr\*> is a do-binding, the keyword is mandatory.

Counters are so common that a special syntax has been reserved for loops using counters. These are called -loops. For-loops come in several variants, and here we will focus on the one using an explicit counter. Its syntax is:

/forLoop.ebnf for <\*ident\*> = <\*firstExpr\*> to <\*lastExpr\*> do <\*bodyExpr\*> [\*done\*]

A for-loop initially binds the counter identifier <\*ident\*> to be the value <\*firstExpr\*>. Then execution enters the body, and <\*bodyExpr\*> is evaluated. Once done, the counter is increased, and execution evaluates <\*bodyExpr\*> once again. This is repeated as long as the counter is not greater than <\*lastExpr\*>. As for while-loops, when using lightweight syntax the block following the keyword up to and including the keyword may be replaced by a newline and indentation.

The counting example from using a -loop is shown in As this interactive script demonstrates, the identifier i takes all the values between 1 and 10, but in spite of its changing state, it is not mutable. Note also that the return value of the expression is , like the printf functions. The lightweight equivalent is shown in .

Counting backwards is sufficiently common that F# has a structure, which works exactly like a --loop except that the counter is decreased by 1 in each iteration. An example of this is shown in .

To further compare for- and while-loops, consider the following problem.

Write a program that calculates the ’th Fibonacci number.

Fibonacci numbers is a sequence of numbers starting with , and where the next number is calculated as the sum of the previous two. Hence the first ten numbers are: . Fibonacci numbers are related to Golden spirals shown in .

|  |
| --- |
|  |

The Fibonacci spiral is an approximation of the golden spiral. Each square has side lengths of successive Fibonacci numbers, and the curve in each square is the circular arc with a radius of the square it is drawn in.

Often the sequence is extended with a preceding number , to be , which we will do here as well.

We could solve this problem with a -loop, as shown in . The basic idea of the solution is that if we are given the ’th and ’th numbers, the ’th number is trivial to compute. And assuming that and are given, then it is trivial to calculate . For , we only need and , hence we may disregard . Thus, we realize that we can cyclicly update the previous, current, and next values by shifting values until we have reached the desired . This is implement in as the function fib, which takes an integer n as argument and returns the ’th Fibonacci number. The function does this iteratively using a -loop, where i is the counter value, and pair is the pair of the ’th and ’th Fibonacci numbers. In the body of the loop, the ’th and ’th numbers are assigned to pair. The -loop automatically updates i for next iteration. When the body of the for-loop is not evaluated, and is returned. This is of course wrong for , but we will ignore this for now.

shows a program similar to using a while-loop instead of for-loop. The programs are almost identical. In this case, the -loop is to be preferred, since more lines of code typically mean more chances of making a mistake. However, while-loops are somewhat easier to argue correctness about.

The correctness of fib in can be proven using a . An is a statement that is always true at a particular point in a program, and a loop invariant is a statement which is true at the beginning and end of a loop. In line [[fibWhileInvariant]](#fibWhileInvariant) in , we may state the invariant: The variable pair is the pair of the ’th and ’th Fibonacci numbers. This is provable by induction:

Base case:

Before entering the while loop, i is 1, pair is (0, 1). Thus, the invariant is true.

Induction step:

Assuming that pair is the ’th and ’th Fibonacci numbers, the body first assigns a new value to pair as the ’th and ’th Fibonacci numbers, then increases by one such that at the end of the loop the pair again contains the the ’th and ’th Fibonacci numbers.

Thus, since our invariant is true for the first case, and any iteration following an iteration where the invariant is true, is also true, then it is true for all iterations.

Thus we know that the second value in pair holds the value of the ’th Fibonacci number, and since we further may prove that when line [[fibWhileInvariantContinue]](#fibWhileInvariantContinue) is reached, then it is proven that fib returns the ’th Fibonacci number.

While-loops also allow for logical structures other than for-loops, such as the case when the number of iteration cannot easily be decided when entering the loop. As an example, consider a slight variation of the above problem, where we wish to find the largest Fibonacci number less or equal some number. A solution to this problem is shown in . The strategy here is to iteratively calculate Fibonacci numbers until we’ve found one larger than the argument n, and then return the previous. This could not be calculated with a for-loop.

## Programming Intermezzo: Automatic Conversion of Decimal to Binary Numbers

Using loops and conditional expressions, we are now able to solve the following problem:

Given an integer on decimal form, write its equivalent value on the binary form.

To solve this problem, consider odd numbers: They all have the property that the least significant bit is 1, e.g., , in contrast to even numbers such as . Division by 2 is equal to right-shifting by 1, e.g., . Thus, through dividing by 2 and checking the remainder, we may sequentially read off the least significant bit. This leads to the algorithm shown in . In the code, the states v and str are iteratively updated until str finally contains the desired solution.

To prove that calculates the correct sequence, we use induction. First we realize that for , the while-loop is skipped, and the result is trivially true. We will concentrate on line [[dec2binWhile]](#dec2binWhile) in and will prove the following loop invariant: The string str contains all the bits of n to the right of the bit pattern remaining in variable v.

Base case :

If only uses the lowest bit, then or . If , then it is trivially correct. Considering the case : Before entering into the loop, v is 1, and str is the empty string, so the invariant is true. The condition of the while-loop is , so execution enters the loop. Since integer division of 1 by 2 gives 0 with remainder 1, str is set to "1" and v to 0. Now we reexamine the while-loop’s condition, , which is false, so we exit the loop. At this point, v is 0 and str is "1", so all bits have been shifted from n to str, and none are left in v. Thus the invariant is true. Finally, the program returns "0b1".

Induction step:

Consider the case of , and assume that the invariant is true when entering the loop, i.e., that bits already have been shifted to str and that . In this case, v contains the remaining bits of n, which is the integer division v = n / 2\*\*m. Since , v is non-zero, and the loop conditions is true, so we enter the loop body. In the loop body we concatenate the rightmost bit of v to the left of str using v % 2, and right-shift v one bit to the right with v <- v / 2. Thus, when returning to the condition the invariant is true, since the right-most bit in v has been shifted to str. This continues until all bits have been shifted to str and v = 0, in which case the loop terminates, and "0b"+str is returned.

Thus we have proven that dec2bin correctly converts integers to strings representing binary numbers.