1 Controlling Program Flow

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 **??**.

1.1 While and For Loops

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

· while@while

Listing 1.1: While loop.

```
while <condition> do <expr> [done]
```

The *condition* `<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 *do* keyword up to and including the *done* keyword may be replaced by a newline and indentation.

· condition

· do@do

· done@done

The program in Listing 1.5 is an example of a while-loop which counts from 1 to 10.

Listing 1.2 countWhile.fsx:
Count to 10 with a counter variable.

```
let mutable i = 1 in while i <= 10 do printf "%d " i; i <- i +
   1 done;
printf "\n"
```

```
$ fsharpc --nologo countWhile.fsx && mono countWhile.exe
1 2 3 4 5 6 7 8 9 10
```

The variable `i` is customarily called the counter variable. The counting is done by performing the following computation: In line 1, the counter variable is first given an initial value of 1. Then execution enters the while-loop and examines the condition. Since 1 <= 10, 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 2 <= 10, 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 11 <= 10 is false, and execution continues in line 2.

In lightweight syntax, this would be as shown in Listing 1.3.

Listing 1.3 countWhileLightweight.fsx:
Count to 10 with a counter variable using lightweight syntax.

```
let mutable i = 1
while i <= 10 do
  printf "%d " i
  i <- i + 1
printf "\n"
```

```
$ fsharpc --nologo countWhileLightweight.fsx
$ mono countWhileLightweight.exe
1 2 3 4 5 6 7 8 9 10
```

Notice that although the expression following the condition is preceded with a `do` keyword, and `do <expr>` is a `do`-binding, the keyword `do` is mandatory.

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

Listing 1.4: For loop.

```
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 *`do`* keyword up to and including the *`done`* keyword may be replaced by a newline and indentation. · do@`do` · done@`done`

The counting example from Listing 1.2 using a `for`-loop is shown in Listing 1.5

Listing 1.5 count.fsx:
Counting from 1 to 10 using a `for`-loop.

```
for i = 1 to 10 do printf "%d " i done
printfn ""
```

```
$ fsharpc --nologo count.fsx && mono count.exe
1 2 3 4 5 6 7 8 9 10
```

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 `for` expression is "`()`", like the `printf` functions. The lightweight equivalent is shown in **??**.

Listing 1.6 countLightweight.fsx:
Counting from 1 to 10 using a for-loop using the lightweight syntax.

```
for i = 1 to 10 do
  printf "%d " i
printfn ""
```

```
$ fsharpc --nologo countLightweight.fsx && mono
   countLightweight.exe
1 2 3 4 5 6 7 8 9 10
```

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

Problem 1.1
Write a program that calculates the n'th Fibonacci number.

Fibonacci numbers is a sequence of numbers starting with $1, 1$, and where the next number is calculated as the sum of the previous two. Hence the first ten numbers are: $1, 1, 2, 3, 5, 8, 13, 21, 34, 55$. Fibonacci numbers are related to Golden spirals shown in **??**. Often the sequence is extended with a preceding number 0, to be $0, 1, 1, 2, 3, \ldots$, which we

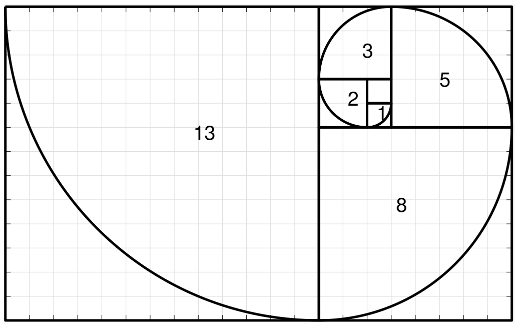

Figure 1.1: The Fibonacci spiral is an approximation of the golden spiral. Each square has side lengths of successive Fibonacci numbers, and the curve in each square is the circular arc with a radius of the square it is drawn in.

will do here as well.

We could solve this problem with a `for`-loop, as shown in **??**.

Listing 1.7 fibFor.fsx:
The n'th Fibonacci number calculated using a for-loop.

```
let fib n =
  let mutable pair = (0, 1)
  for i = 2 to n do
    pair <- (snd pair, (fst pair) + (snd pair))
  snd pair

printfn "fib(1) = %d" (fib 1)
printfn "fib(2) = %d" (fib 2)
printfn "fib(3) = %d" (fib 3)
printfn "fib(10) = %d" (fib 10)
```

```
$ fsharpc --nologo fibFor.fsx && mono fibFor.exe
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(10) = 55
```

The basic idea of the solution is that if we are given the $(n-1)$'th and $(n-2)$'th numbers, the n'th number is trivial to compute. And assuming that fib(1) and fib(2) are given, then it is trivial to calculate fib(3). For fib(4), we only need fib(3) and fib(2), hence we may disregard fib(1). Thus, we realize that we can cyclicly update the previous, current, and next values by shifting values until we have reached the desired fib(n). This is implement in **??** as the function `fib`, which takes an integer `n` as argument and returns the n'th Fibonacci number. The function does this iteratively using a `for`-loop, where `i` is the counter value, and `pair` is the pair of the $i-1$'th and i'th Fibonacci numbers. In the body of the loop, the i'th and $i+1$'th numbers are assigned to `pair`. The `for`-loop automatically updates `i` for next iteration. When $n < 2$ the body of the for-loop is not evaluated, and 1 is returned. This is of course wrong for $n < 1$, but we will ignore this for now.

?? shows a program similar to **??** using a while-loop instead of for-loop.

Listing 1.8 fibWhile.fsx:
The n'th Fibonacci number calculated using a while-loop.

```fsharp
let fib (n : int) : int =
  let mutable pair = (0, 1)
  let mutable i = 1
  while i < n do
    pair <- (snd pair, fst pair + snd pair)
    i <- i + 1
  snd pair

printfn "fib(1) = %d" (fib 1)
printfn "fib(2) = %d" (fib 2)
printfn "fib(3) = %d" (fib 3)
printfn "fib(10) = %d" (fib 10)
```

```
$ fsharpc --nologo fibWhile.fsx && mono fibWhile.exe
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(10) = 55
```

The programs are almost identical. In this case, the `for`-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 *loop invariant*. An *invariant* 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 **??** in **??**, we may state the invariant: The variable `pair` is the pair of the $i-1$'th and i'th Fibonacci numbers. This is provable by induction:

· loop invariant
· invariant

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 $i-1$'th and i'th Fibonacci numbers, the body first assigns a new value to `pair` as the i'th and $i+1$'th Fibonacci numbers, then increases i by one such that at the end of the loop the `pair` again contains the the $i-1$'th and i'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 i'th Fibonacci number, and since we further may prove that $i = n$ when line **??** is reached, then it is proven that `fib` returns the n'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 **??**.

Listing 1.9 fibWhileLargest.fsx:
Search for the largest Fibonacci number less than a specified number.

```
let largestFibLeq n =
  let mutable pair = (0, 1)
  while snd pair <= n do
    pair <- (snd pair, fst pair + snd pair)
  fst pair

for i = 1 to 10 do
  printfn "largestFibLeq(%d) = %d" i (largestFibLeq i)
```

```
$ fsharpc --nologo fibWhileLargest.fsx && mono
   fibWhileLargest.exe
largestFibLeq(1) = 1
largestFibLeq(2) = 2
largestFibLeq(3) = 3
largestFibLeq(4) = 3
largestFibLeq(5) = 5
largestFibLeq(6) = 5
largestFibLeq(7) = 5
largestFibLeq(8) = 8
largestFibLeq(9) = 8
largestFibLeq(10) = 8
```

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.

1.2 Conditional Expressions

Programs often contain code which should only be executed under certain conditions. This can be expressed with `if`-expressions, whose syntax is as follows.

· if@`if`
· then@`then`
· elif@`elif`
· else@`else`

Listing 1.10: Conditional expressions.

```
if <cond> then <expr> {elif <cond> then <expr>} [else <expr>]
```

The condition `<con>` is an expression resulting in a Boolean value, and there can be zero or more `elif` conditions, as indicated by `{}`. Each expression `<expr>` is called a *branch*, and all branches must have the same type, such that regardless of which branch is chosen, the type of the result of the conditional expression is the same. Then the expression of the first if-branch, whose condition is true, is evaluate. If all conditions are false then the `else`-branch is evaluated. If no `else` expression is present, then "`()`" will be returned. See **??** for a simple example.

· branch

Listing 1.11 condition.fsx:
Conditions evaluate their branches depending on the value of the condition.

```
if true then printfn "hi" else printfn "bye"
if false then printfn "hi" else printfn "bye"
```

```
$ fsharpc --nologo condition.fsx && mono condition.exe
hi
bye
```

The lightweight syntax allows for newlines entered everywhere, but indentation must be used to express scope.

To demonstrate conditional expressions, let us write a program which writes the sentence “I have n apple(s)”, where the plural ’s’ is added appropriately for various n’s. This is done in **??**, using the lightweight syntax.

Listing 1.12 conditionalLightweight.fsx:
Using conditional expression to generate different strings.

```
let applesIHave n =
  if n < -1 then
    "I owe " + (string -n) + " apples"
  elif n < 0 then
    "I owe " + (string -n) + " apple"
  elif n < 1 then
    "I have no apples"
  elif n < 2 then
    "I have 1 apple"
  else
    "I have " + (string n) + " apples"

printfn "%A" (applesIHave -3)
printfn "%A" (applesIHave -1)
printfn "%A" (applesIHave 0)
printfn "%A" (applesIHave 1)
printfn "%A" (applesIHave 2)
printfn "%A" (applesIHave 10)
```

```
$ fsharpc --nologo conditionalLightWeight.fsx
$ mono conditionalLightWeight.exe
"I owe 3 apples"
"I owe 1 apple"
"I have no apples"
"I have 1 apple"
"I have 2 apples"
"I have 10 apples"
```

The sentence structure and its variants give rise to a more compact solution, since the language to be returned to the user is a variant of "I have/owe no/number apple(s)", i.e., certain conditions determine whether the sentence should use “have” and “owe” and so forth. So, we could instead make decisions on each of these sentence parts, and then built the final sentence from its parts. This is accomplished in the following example:

Listing 1.13 conditionalLightweightAlt.fsx:
Using sentence parts to construct the final sentence.

```
let applesIHave n =
  let haveOrOwe = if n < 0 then "owe" else "have"
  let pluralS = if (n = 0) || (abs n) > 1 then "s" else ""
  let number = if n = 0 then "no" else (string (abs n))

  "I " + haveOrOwe + " " + number + " apple" + pluralS

printfn "%A" (applesIHave -3)
printfn "%A" (applesIHave -1)
printfn "%A" (applesIHave 0)
printfn "%A" (applesIHave 1)
printfn "%A" (applesIHave 2)
printfn "%A" (applesIHave 10)
```

```
$ fsharpc --nologo conditionalLightWeightAlt.fsx
$ mono conditionalLightWeightAlt.exe
"I owe 3 apples"
"I owe 1 apple"
"I have no apples"
"I have 1 apple"
"I have 2 apples"
"I have 10 apples"
```

While arguably shorter, this solution is also denser, and most likely more difficult to debug and maintain.

Note that both `elif` and `else` branches are optional, which may cause problems. For example, both `let a = if true then 3` and `let a = if true then 3 elif false then 4` will be invalid, since F# is not smart enough to realize that the type of the expression is uniquely determined. Instead, F# looks for the `else` to ensure all cases have been covered, and that `a` always will be given a unique value of the same type regardless of the branch taken in the conditional statement. Hence, `let a = if true then 3 else 4` is the only valid expression of the 3. In practice, F# assumes that the omitted branch returns "`()`", and thus it is fine to say `let a = if true then ()` and `if true then printfn "hej"`. Nevertheless, it is good practice in F# to always include an `else` branch.

1.3 Programming Intermezzo: Automatic Conversion of Decimal to Binary Numbers

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

Problem 1.2

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., $1_2 = 1, 101_2 = 5$, in contrast to even numbers such as $110_2 = 6$. Division by 2 is equal to right-shifting by 1, e.g., $1_2/2 = 0.1_2 = 0.5, 101_2/2 = 10.1_2 =$

$2.5, 110_2/2 = 11_2 = 3$. 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 **??**.

Listing 1.14 dec2bin.fsx:
Using integer division and remainder to write any positive integer in binary form.

```
let dec2bin n =
  if n < 0 then
    "Illegal value"
  elif n = 0 then
    "0b0"
  else
    let mutable v = n
    let mutable str = ""
    while v > 0 do
      str <- (string (v % 2)) + str
      v <- v / 2
    "0b" + str


printfn "%4d -> %s" -1 (dec2bin -1)
printfn "%4d -> %s" 0 (dec2bin 0)
for i = 0 to 3 do
  printfn "%4d -> %s" (pown 10 i) (dec2bin (pown 10 i))
```

```
$ fsharpc --nologo dec2bin.fsx && mono dec2bin.exe
  -1 -> Illegal value
   0 -> 0b0
   1 -> 0b1
  10 -> 0b1010
 100 -> 0b1100100
1000 -> 0b1111101000
```

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 $v < 1$, the while-loop is skipped, and the result is trivially true. We will concentrate on line **??** 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 $n = 000\dots000x$: If n only uses the lowest bit, then $n = 0$ or $n = 1$. If $n = 0$, then it is trivially correct. Considering the case $n = 1$: 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 $1 > 0$, 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, $0 > 0$, 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 $n > 1$, and assume that the invariant is true when entering the loop, i.e., that m bits already have been shifted to `str` and that $n > 2^m$. In this case, `v` contains the remaining bits of `n`, which is the integer division `v = n`

/ 2**m. Since $n > 2^m$, 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.