

Learning to program with F#

Jon Sparring

September 17, 2016

Contents

1	Preface	5
2	Introduction	6
2.1	How to learn to program	6
2.2	How to solve problems	7
2.3	Approaches to programming	7
2.4	Why use F#	8
2.5	How to read this book	9
I	F# basics	10
3	Executing F# code	11
3.1	Source code	11
3.2	Executing programs	11
4	Quick-start guide	14
5	Using F# as a calculator	19
5.1	Literals and basic types	19
5.2	Operators on basic types	25
5.3	Boolean arithmetic	26
5.4	Integer arithmetic	27
5.5	Floating point arithmetic	30
5.6	Char and string arithmetic	31
5.7	Programming intermezzo	33

6	Constants, functions, and variables	35
6.1	Values	38
6.2	Non-recursive functions	43
6.3	User-defined operators	47
6.4	The Printf function	49
6.5	Variables	52
7	In-code documentation	57
8	Controlling program flow	62
8.1	For and while loops	62
8.2	Conditional expressions	66
8.3	Recursive functions	68
8.4	Programming intermezzo	71
9	Ordered series of data	75
9.1	Tuples	76
9.2	Lists	79
9.3	Arrays	81
10	Testing programs	87
10.1	White-box testing	90
10.2	Back-box testing	93
10.3	Debugging by tracing	95
11	Exceptions	102
12	Input and Output	109
12.1	Interacting with the console	110
12.2	Storing and retrieving data from a file	111
12.3	Working with files and directories.	116
12.4	Programming intermezzo	116

II	Imperative programming	119
13	Graphical User Interfaces	121
14	Imperative programming	122
14.1	Introduction	122
14.2	Generating random texts	123
14.2.1	0'th order statistics	123
14.2.2	1'th order statistics	123
III	Declarative programming	124
15	Sequences and computation expressions	125
15.1	Sequences	125
16	Patterns	131
16.1	Pattern matching	131
17	Types and measures	134
17.1	Unit of Measure	134
18	Functional programming	138
IV	Structured programming	141
19	Namespaces and Modules	142
20	Object-oriented programming	144
V	Appendix	145
A	Number systems on the computer	146
A.1	Binary numbers	148
A.2	IEEE 754 floating point standard	148

B	Commonly used character sets	149
B.1	ASCII	149
B.2	ISO/IEC 8859	150
B.3	Unicode	150
C	A brief introduction to Extended Backus-Naur Form	154
D	F_b	158
E	Language Details	163
E.1	Arithmetic operators on basic types	163
E.2	Basic arithmetic functions	166
E.3	Precedence and associativity	167
E.4	Lightweight Syntax	169
F	The Some Basic Libraries	170
F.1	System.String	171
F.2	List, arrays, and sequences	171
F.3	Mutable Collections	174
F.3.1	Mutable lists	174
F.3.2	Stacks	174
F.3.3	Queues	174
F.3.4	Sets and dictionaries	174
	Bibliography	175
	Index	176

Chapter 8

Controlling program flow

Non-recursive functions encapsulates code and allows for some control of flow, that is, if there is a piece of code, which we need to to have executed many times, then we can encapsulate it in the body of a function, and then call the function several times. In this chapter, we will look at more general control of flow via loops, conditional execution, and recursion, and therefore we look at further extension of the `expr` rule,

Listing 8.1: Expressions for controlling the flow of execution.

```
expr = ...
| "if" expr "then" expr {"elif" expr "then" expr} ["else" expr] (*
conditional*)
| "while" expr "do" expr ["done"] (*while loop*)
| "for" ident "=" expr "to" expr "do" expr ["done"] (*simple for loop*)
| "let" functionDefn "in" expr (*binding a function or operator*)
| "let" "rec" functionDefn {"and" functionDefn} "in" expr (*recursive fcts*)
```

8.1 For and while loops

Many programming constructs need to be repeated, and F# contains many structures for repetition such as the `for` and `while` loops, which have the syntax,

Listing 8.2: `for`- and `while`-loops.

```
expr = ...
| "while" expr "do" expr ["done"] (*while loop*)
| "for" ident "=" expr "to" expr "do" expr ["done"] (*simple for loop*)
```

As an example, consider counting from 1 to 10 with a `for`-loop,

· `for`

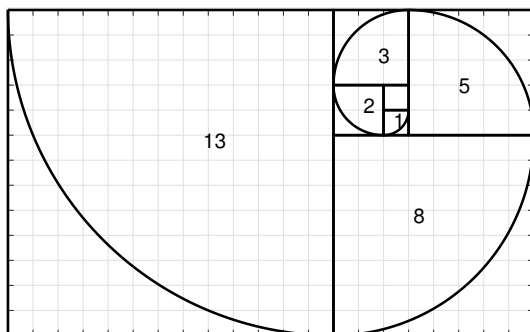


Figure 8.1: The Fibonacci spiral is an approximation of the golden spiral. Each square has side lengths of successive Fibonacci numbers, and the curve in each square is the circular arc with radius of the square it is drawn in

Listing 8.3, count.fsx:
Counting from 1 to 10 using a `for`-loop.

```
> for i = 1 to 10 do printf "%d " i done;
- printfn " ";
1 2 3 4 5 6 7 8 9 10

val it : unit = ()
```

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. Using lightweight syntax the block following the `do` keyword up to and including the `done` keyword may be replaced by a newline and indentation, e.g.,

· `do`
· `done`

Listing 8.4, countLightweight.fsx:
Counting from 1 to 10 using a `for`-loop, see Listing 8.3.

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

1 2 3 4 5 6 7 8 9 10
```

A more complicated example is,

Problem 8.1:

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

The Fibonacci numbers is the series of numbers 1, 1, 2, 3, 5, 8, 13..., where the $\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2)$, and they are related to Golden spirals shown in Figure 8.1. We could solve this problem with a `for`-loop as follows,

Listing 8.5, fibFor.fsx:

The n 'th Fibonacci number is the sum of the previous 2.

```
let fib n =
  let mutable prev = 1
  let mutable current = 1
  let mutable next = 0
  for i = 3 to n do
    next <- current + prev
    prev <- current
    current <- next
  next

printfn "fib(1) = 1"
printfn "fib(2) = 1"
for i = 3 to 10 do
  printfn "fib(%d) = %d" i (fib i)
```

```
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34
fib(10) = 55
```

The basic idea of the solution is that if we are given the $(n - 1)$ 'th and $(n - 2)$ 'th numbers, then the n 'th number is trivial to compute. And assume that `fib(1)` and `fib(2)` are given, then it is trivial to calculate the `fib(3)`. For the `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)`.

The `while`-loop is simpler than the `for`-loop and does not contain a builtin counter structure. Hence, if we are to repeat the count-to-10 program from Listing 8.3 example, it would look somewhat like, `while`

Listing 8.6, 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"
```

```
1 2 3 4 5 6 7 8 9 10
```

or equivalently using the lightweight syntax,

Listing 8.7, countWhileLightweight.fsx:

Count to 10 with a counter variable using lightweight syntax, see Listing 8.6.

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

1 2 3 4 5 6 7 8 9 10

In this case, the `for`-loop is to be preferred, since more lines of code typically means more chances of making a mistake. But the `while`-loop allows for other logical structures. E.g., let's find the biggest Fibonacci number less than 100,

Listing 8.8, fibWhile.fsx:

Search for the largest Fibonacci number less than a specified number.

```
let largestFibLeq n =
  let mutable prev = 1
  let mutable current = 1
  let mutable next = 0
  while next <= n do
    next <- prev + current
    prev <- current
    current <- next
  prev

printfn "largestFibLeq(1) = 1"
printfn "largestFibLeq(2) = 1"
for i = 3 to 10 do
  printfn "largestFibLeq(%d) = %d" i (largestFibLeq i)
```

largestFibLeq(1) = 1
largestFibLeq(2) = 1
largestFibLeq(3) = 3
largestFibLeq(4) = 3
largestFibLeq(5) = 5
largestFibLeq(6) = 5
largestFibLeq(7) = 5
largestFibLeq(8) = 8
largestFibLeq(9) = 8
largestFibLeq(10) = 8

Thus, `while`-loops are most often used, when the number of iteration cannot easily be decided, when entering the loop.

Both `for`- and `while`-loops are often associated with variables, i.e., values that change while looping. If one mistakenly used values and rebinding, then the result would in most cases be of little use, e.g.,

Listing 8.9, forScopeError.fsx:

Lexical scope error. While rebinding is valid F# syntax, has little effect due to lexical scope.

```
let a = 1
for i = 1 to 10 do
    let a = a + 1
    printf "(%d, %d) " i a
printf "\n"
```

```
(1, 2) (2, 2) (3, 2) (4, 2) (5, 2) (6, 2) (7, 2) (8, 2) (9, 2) (10, 2)
```

I.e., the `let` expression rebinds `a` every iteration of the loop, but the value on the right-hand-side is taken lexically from above, where `a` has the value 1, so every time the result is the value 2.

8.2 Conditional expressions

Consider the task,

Problem 8.2:

Write a function that given n writes the sentence, "I have n apple(s)", where the plural 's' is added appropriately.

For this we need to test the value of n , and one option is to use conditional expressions. Conditional expression has the syntax, The grammar for conditional expressions is,

Listing 8.10: Conditional expressions.

```
expr = ...
| "if" expr "then" expr {"elif" expr "then" expr} ["else" expr] (*
conditional*)
```

and an example using conditional expressions to solve the above problem is,

Listing 8.11, 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)

-----

"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 expr following *if* and *elif* are *conditions*, i.e., expressions that evaluate to a boolean value. The expr following *then* and *else* are called *branches*, and all branches must have identical type, such that regardless which branch is chosen, then the type of the result of the conditional expression is the same. The result of the conditional expression is the first branch, for which its condition was true.

- *if*
- *elif*
- conditions
- *then*
- *else*
- branches

The sentence structure and its variants gives rise to a more compact solution, since the language to be returned to the user is a variant of "I have/or no/number apple(s)", i.e., under certain conditions should the sentence use "have" and "owe" etc.. 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 8.12, 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)

-----

"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 more dense, and for a small problem like this, it is 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 branches 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 branches 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# always to include an `else` branch.

8.3 Recursive functions

Recursion is a central concept in F#. A *recursive function* is a function, which calls itself. From a compiler point of view, this is challenging, since the function is used before the compiler has completed its analysis. However, for this there is a technical solution, and we will just concern ourselves with the logics of using recursion for programming. The syntax for defining recursive functions in F# is,

· recursive
function

Listing 8.13: Recursive functions.

```
expr = ...
| "let" "rec" functionDefn {"and" functionDefn} "in" expr
```

An example of a recursive function that counts from 1 to 10 similarly to Listing 8.3 is,¹

Listing 8.14, countRecursive.fsx:
Counting to 10 using recursion.

```
let rec prt a b =
    if a > b then
        printf "\n"
    else
        printf "%d " a
        prt (a + 1) b

prt 1 10
```

```
1 2 3 4 5 6 7 8 9 10
```

Here the `prt` calls itself repeatedly, such that the first call is `prt 1 10`, which calls `prt 2 10`, and so on until the last call `prt 10 10`. Calling `prt 11 10` would not result in recursive calls, since when `a` is higher than `b` then the *stopping criterium* is met and a newline is printed. For values of `a` smaller than or equal `b` then the recursive branch is executed. Since `prt` calls itself at the end of the recursion branch, then this is a *tail-recursive* function. Most compilers achieve high efficiency in terms of speed and memory, so **prefer tail-recursion whenever possible**. Using recursion to calculate the Fibonacci number as Listing 8.5.

· stopping
criterium
· tail-recursive
Advice

Listing 8.15, fibRecursive.fsx:
The n 'th Fibonacci number using recursive.

```
let rec fib n =
    if n < 1 then
        0
    elif n = 1 then
        1
    else
        fib (n - 1) + fib (n - 2)

for i = 0 to 10 do
    printfn "fib(%d) = %d" i (fib i)
```

```
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34
fib(10) = 55
```

¹Todo: A drawing showing the stack for the example would be good.

Here we used the fact that including $\text{fib}(0) = 0$ in the Fibonacci series also produces it using the rule $\text{fib}(n) = \text{fib}(n - 2) + \text{fib}(n - 1)$, $n \geq 0$, which allowed us to define a function that is well defined for the complete set of integers. I.e., a negative argument returns 0. This is a general advice: **make functions that fails gracefully.**

Advice

Functions that recursively call each other are called *mutually recursive* functions. F# offers the **let-rec-and** notation for co-defining mutually recursive functions. As an example, consider the function `even : int -> bool`, which returns true if its argument is even and false otherwise, and the opposite function `odd : int -> bool`. A mutually recursive implementation of these functions can be developed from the following statements: `even 0 = true`, `odd 0 = false`, and `even n = odd (n-1)`:

· mutually recursive

Listing 8.16, mutuallyRecursive.fsx:

Using mutual recursion to implement even and odd functions.

```
let rec even x =
    if x = 0 then true
    else odd (x - 1)
and odd x =
    if x = 0 then false
    else even (x - 1);;

let w = 5;
printfn "%s %s %s" w "i" w "even" w "odd"
for i = 1 to w do
    printfn "%d %b %b" w i w (even i) w (odd i)
```

```
i  even  odd
1 false true
2  true false
3 false true
4  true false
5 false true
```

Notice that in the lightweight notation used here, that the **and** must be on the same indentation level as the original **let**.

Without the **and** keyword, F# will return an error at the definition of `even`. However, it is possible to implement mutual recursion by using functions as an argument, e.g.,

Listing 8.17, mutuallyRecursiveAlt.fsx:

Mutual recursion without the `and` keyword needs a helper function.

```
let rec evenHelper (notEven: int -> bool) x =
    if x = 0 then true
    else notEven (x - 1)

let rec odd x =
    if x = 0 then false
    else evenHelper odd (x - 1);;

let even x = evenHelper odd x

let w = 5;
printfn "%*s %*s %*s" w "i" w "Even" w "Odd"
for i = 1 to w do
    printfn "%*d %*b %*b" w i w (even i) w (odd i)
```

```
i  Even  Odd
1 false true
2  true false
3 false true
4  true false
5 false true
```

But, Listing 8.16 is clearly to be preferred over Listing 8.17.

In the above we used the `even` and `odd` function problems to demonstrate mutual recursion. There is, of course, a much simpler solution, which does not use recursion at all:

Listing 8.18: A better way to test for parity without recursion.

```
let even x = (x % 2 = 0)
let odd x = not (even x)
```

which is to be preferred anytime as the solution to the problem.

8.4 Programming intermezzo

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

Problem 8.3:

Given an integer on decimal form, write its equivalent value on 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 by integer division by 2 and checking the remainder, we may sequentially read off the least significant bit. This leads to the following algorithm,

Listing 8.19, dec2bin.fsx:

Using integer division and remainder to write any positive integer on binary form.

```
let dec2bin n =
  let rec dec2binHelper n =
    let mutable v = n
    let mutable str = ""
    while v > 0 do
      str <- (string (v % 2)) + str
      v <- v / 2
    str

  if n < 0 then
    "Illegal value"
  elif n = 0 then
    "0b0"
  else
    "0b" + (dec2binHelper n)

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))
```

```
-1 -> Illegal value
0 -> 0b0
1 -> 0b1
10 -> 0b1010
100 -> 0b1100100
1000 -> 0b1111101000
```

Another solution is to use recursion instead of the `while` loop:

Listing 8.20, dec2binRec.fsx:

Using recursion to write any positive integer on binary form, see also Listing 8.19.

```
let dec2bin n =
    let rec dec2binHelper n =
        if n = 0 then ""
        else (dec2binHelper (n / 2)) + string (n % 2)

    if n < 0 then
        "Illegal value"
    else
        "0b" +
        if n = 0 then
            "0"
        else
            dec2binHelper n

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))
```

```
-1 -> Illegal value
 0 -> 0b0
 1 -> 0b1
10 -> 0b1010
100 -> 0b1100100
1000 -> 0b1111101000
```

Listing 8.19 is a typical imperative solution, where the states `v` and `str` are iteratively updated until `str` finally contains the desired solution. Listing 8.20 is a typical functional programming solution, to be discussed in Part III, where the states are handled implicitly as new scopes created by recursively calling the helper function. Both solutions have been created using a local helper function, since both solutions require special treatment of the cases $n < 0$ and $n = 0$.

Let us compare the two solutions more closely: The computation performed is the same in both solutions, i.e., integer division and remainder is used repeatedly, but since the recursive solution is slightly shorter, then one could argue that it is better, since shorter programs typically have fewer errors. However, shorter program also typically means that understanding them is more complicated, since shorter programs often rely on realisations that the author had while programming, which may not be properly communicated by the code nor comments. Speedwise, there is little difference between the two methods: 10,000 conversions of `System.Int32.MaxValue`, i.e., the number 2,147,483,647, takes about 1.1 sec for both methods on an 2,9 GHz Intel Core i5 machine.

Notice also, that in Listing 8.20, the prefix `"0b"` is only written once. This is advantageous for later debugging and updating, e.g., if we later decide to alter the program to return a string without the prefix or with a different prefix, then we would only need to change one line instead of two. However, the program has gotten slightly more difficult to read, since the string concatenation operator and the `if` statement are now intertwined. There is thus no clear argument for preferring one over the other by this argument.

Proving that Listing 8.20 computes the correct sequence is easily done using the induction proof technique: The result of `dec2binHelper 0` is clearly an empty string. For calls to `dec2binHelper n` with $n > 0$, we check that the right-most bit is correctly converted by the remainder function, and that

this string is correctly concatenated with `dec2binHelper` applied to the remaining bits. A simpler way to state this is to assume that `dec2binHelper` has correctly programmed, so that in the body of `dec2binHelper`, then recursive calls to `dec2binHelper` returns the correct value. Then we only need to check that the remaining computations are correct. Proving that Listing 8.19 calculates the correct sequence essentially involves the same steps: If $v = 0$ then the `while` loop is skipped, and the result is the initial value of `str`. For each iteration of the `while` loop, assuming that `str` contains the correct conversions of the bits up till now, we check that the remainder operator correctly concatenates the next bit, and that `v` is correctly updated with the remaining bits. We finally check that the loop terminates, when no more 1-bits are left in `v`. Comparing the two proofs, the technique of assuming that the problem has been solved, i.e., that recursive calls will work, helps us focus on the key issues for the proof. Hence, we conclude that the recursive solution is most elegantly proved, and thus preferred.

Bibliography

- [1] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.
- [2] Programming Research Group. Specifications for the ibm mathematical formula translating system, fortran. Technical report, Applied Science Division, International Business Machines Corporation, 1954.
- [3] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.
- [4] X3: ASA Sectional Committee on Computers and Information Processing. American standard code for information interchange. Technical Report ASA X3.4-1963, American Standards Association (ASA), 1963. <http://worldpowersystems.com/projects/codes/X3.4-1963/>.
- [5] George Pólya. *How to solve it*. Princeton University Press, 1945.

Index

. [], 32
ReadKey, 110
ReadLine, 110
Read, 110
System.Console.ReadKey, 110
System.Console.ReadLine, 110
System.Console.Read, 110
System.Console.WriteLine, 110
System.Console.Write, 110
WriteLine, 110
Write, 110
abs, 166
acos, 166
asin, 166
atan2, 166
atan, 166
bignum, 23
bool, 19
byte[], 23
byte, 23
ceil, 166
char, 19
cosh, 166
cos, 166
decimal, 23
double, 23
eprintfn, 52
eprintf, 52
exn, 19
exp, 166
failwithf, 52
float32, 23
float, 19
floor, 166
fprintfn, 52
fprintf, 52
ignore, 52
int16, 23
int32, 23
int64, 23
int8, 23
int, 19
it, 19
log10, 166
log, 166

max, 166
min, 166
nativeint, 23
obj, 19
pown, 166
printfn, 52
printf, 49, 52
round, 166
sbyte, 23
sign, 166
single, 23
sinh, 166
sin, 166
sprintf, 52
sqrt, 166
stderr, 52, 110
stdin, 110
stdout, 52, 110
string, 19
tanh, 166
tan, 166
uint16, 23
uint32, 23
uint64, 23
uint8, 23
unativeint, 23
unit, 19

American Standard Code for Information Inter-
change, 149

and, 26
anonymous function, 46
array sequence expressions, 130
Array.toArray, 84
Array.toList, 84
ASCII, 149
ASCIIbetical order, 31, 150

base, 20, 148
Basic Latin block, 150
Basic Multilingual plane, 150
basic types, 19
binary, 148
binary number, 21
binary operator, 25
binary64, 148

- binding, 14
- bit, 21, 148
- black-box testing, 88
- block, 41
- blocks, 150
- boolean and, 167
- boolean or, 167
- branches, 67
- branching coverage, 90
- bug, 87
- byte, 148

- character, 21
- class, 24, 33
- code point, 21, 150
- compiled, 11
- computation expressions, 79, 81
- conditions, 67
- Cons, 81
- console, 11
- coverage, 90
- currying, 47

- debugging, 13, 88, 95
- decimal number, 20, 148
- decimal point, 20, 148
- Declarative programming, 8
- digit, 20, 148
- dot notation, 33
- double, 148
- downcasting, 24

- EBNF, 20, 154
- efficiency, 88
- encapsulate code, 43
- encapsulation, 47
- environment, 96
- exception, 29
- exclusive or, 30
- executable file, 11
- expression, 14, 25
- expressions, 8
- Extended Backus-Naur Form, 20, 154
- Extensible Markup Language, 57

- file, 109
- floating point number, 20
- format string, 14
- fractional part, 20, 24
- function, 17
- function coverage, 90
- Functional programming, 8, 123
- functional programming, 8
- functionality, 87
- functions, 8

- generic function, 44

- hand tracing, 95
- Head, 81
- hexadecimal, 148
- hexadecimal number, 21
- HTML, 60
- Hyper Text Markup Language, 60

- IEEE 754 double precision floating-point format, 148
- Imperativ programming, 122
- Imperative programming, 8
- implementation file, 11
- infix notation, 26
- infix operator, 25
- integer, 20
- integer division, 28
- interactive, 11
- IsEmpty, 81
- Item, 81

- jagged arrays, 84

- keyword, 14

- Latin-1 Supplement block, 150
- Latin1, 150
- least significant bit, 148
- Length, 81
- length, 76
- lexeme, 17
- lexical scope, 16, 45
- lexically, 39
- lightweight syntax, 36, 39
- list, 79
- list sequence expression, 130
- List.Empty, 81
- List.toArray, 81
- List.toList, 81
- literal, 19
- literal type, 23

- machine code, 122
- maintainability, 88
- member, 24, 76
- method, 33
- mockup code, 95
- module elements, 142
- modules, 11
- most significant bit, 148
- Mutable data, 52
- mutually recursive, 70

- namespace, 24
- namespace pollution, 137

- NaN, 148
- nested scope, 41
- newline, 22
- not, 26
- not a number, 148
- obfuscation, 79
- object, 33
- Object oriented programming, 122
- Object-oriented programming, 8
- objects, 8
- octal, 148
- octal number, 21
- operand, 44
- operands, 25
- operator, 25, 26, 44
- or, 26
- overflow, 28
- pattern matching, 131, 138
- portability, 88
- precedence, 25, 26
- prefix operator, 25
- Procedural programming, 122
- procedure, 47
- production rules, 154
- ragged multidimensional list, 81
- raise an exception, 102
- range expression, 79
- reals, 148
- recursive function, 68
- reference cells, 54
- reliability, 87
- remainder, 28
- rounding, 24
- run-time error, 29
- scientific notation, 20
- scope, 41
- script file, 11
- script-fragment, 17
- script-fragments, 11
- Seq.initInfinite, 129
- Seq.item, 127
- Seq.take, 127
- Seq.toArray, 129
- Seq.toList, 129
- side-effect, 83
- side-effects, 47, 55
- signature file, 11
- slicing, 84
- software testing, 88
- state, 8
- statement, 14
- statement coverage, 90
- statements, 8, 122
- states, 122
- stopping criterium, 69
- stream, 110
- string, 14, 22
- Structured programming, 8
- subnormals, 148
- Tail, 81
- tail-recursive, 69
- terminal symbols, 154
- tracing, 95
- truth table, 26
- tuple, 76
- type, 15, 19
- type declaration, 15
- type inference, 13, 15
- type safety, 44
- typecasting, 23
- unary operator, 25
- underflow, 28
- Unicode, 21
- unicode general category, 150
- Unicode Standard, 150
- unit of measure, 134
- unit testing, 88
- unit-less, 135
- unit-testing, 13
- upcasting, 24
- usability, 88
- UTF-16, 152
- UTF-8, 152
- variable, 52
- verbatim, 23
- white-box testing, 88, 90
- whitespace, 22
- whole part, 20, 24
- wild card, 39
- word, 148
- XML, 57
- xor, 30
- yield bang, 127