Learning to program with F#

Jon Sporring

August 8, 2016

Contents

1 Preface **4**

2 Introduction **5**

I F# basics 7

3 Executing F# code **8**
- 3.1 Source code . . . 8
- 3.2 Executing programs . . . 8

4 Quick-start guide **10**

5 Using F# as a calculator **14**
- 5.1 Literals and basic types . . . 14
- 5.2 Operators on basic types . . . 19
- 5.3 Boolean arithmetic . . . 24
- 5.4 Integer arithmetic . . . 25
- 5.5 Floating point arithmetic . . . 26
- 5.6 Char and string arithmetic . . . 27

6 Constants, functions, and variables **30**
- 6.1 Values . . . 32
- 6.2 Non-recursive functions . . . 35
- 6.3 User-defined operators . . . 38
- 6.4 The Printf function . . . 40
- 6.5 Variables . . . 42

7 In-code documentation **46**

8 Controlling program flow **50**
- 8.1 For and while loops . . . 50
- 8.2 Conditional expressions . . . 53
  - 8.2.1 Programming intermezzo . . . 54
- 8.3 Pattern matching . . . 55
- 8.4 Recursive functions . . . 57

9 Ordered series of data **59**
- 9.1 Tuples . . . 60
- 9.2 Lists . . . 62
- 9.3 Arrays . . . 65
- 9.4 Sequences . . . 70

10 Testing programs **74**
10.1 White-box testing . . . 76
10.2 Back-box testing . . . 78

II Imperative programming **81**

11 Exceptions **83**

12 Input/Output **84**
12.1 Console I/O . . . 84
12.2 File I/O . . . 84

13 Graphical User Interfaces **86**

14 Imperative programming **87**
14.1 Introduction . . . 87
14.2 Generating random texts . . . 87
14.2.1 0'th order statistics . . . 87
14.2.2 1'th order statistics . . . 89

III Declarative programming **92**

15 Types and measures **93**
15.1 Unit of Measure . . . 93

16 Functional programming **96**

IV Structured programming **97**

17 Namespaces and Modules **98**

18 Object-oriented programming **100**

V Appendix **101**

A Number systems on the computer **102**
A.1 Binary numbers . . . 102
A.2 IEEE 754 floating point standard . . . 102

B Commonly used character sets **106**
B.1 ASCII . . . 106
B.2 ISO/IEC 8859 . . . 106
B.3 Unicode . . . 107

C A brief introduction to Extended Backus-Naur Form **110**

4 Language Details **113**

E The Collection **115**
E.1 `System.String` . . . 115
E.2 List, arrays, and sequences . . . 120
E.3 Mutable Collections . . . 122
E.3.1 Mutable lists . . . 122

E.3.2 Stacks . . . 122
E.3.3 Queues . . . 122
E.3.4 Sets and dictionaries . . . 123

Bibliography **124**

Index **125**

Chapter 4

Language Details

Minimal F# used in Part I

```
(*Whitespace*)
whitespace = ' ' {' '}
newline = '\n' | '\r' '\n'

(*Literal digits*)
dDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
bDigit = "0" | "1"
oDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"
xDigit =
  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
  | "A" | "B" | "C" | "D" | "E" | "F" | "a" | "b" | "c" | "d" | "e" | "f"

(*Literal integers*)
int = dInt | xint
sbyte = (dInt | xInt) "y"
byte = (dInt | xInt) "uy"
int32 = (dInt | xInt) ["l"]
uint32 = (dInt | xInt) ("u" | "ul")

dInt = dDigit {dDigit}
bitInt = "0" ("b" | "B") bDigit {bDigit}
octInt = "0" ("o" | "O") oDigit {oDigit}
hexInt = "0" ("x" | "X") xDigit {xDigit}
xint = bitInt | octInt | hexInt

(*Literal floats*)
float = dFloat | sFloat
dFloat = dInt "." {dDigit}
sFloat = (dInt | dFloat) ("e" | "E" ) ["+" | "-"] dInt
ieee64 = float | xInt "LF"

(*Literal chars*)
char = "'" codePoint | escapeChar "'" (*codePoint is any unicode codepoint*)
escapeChar =
  "\" ("b" | "n" | "r" | "t" | "\" | '"' | "'" | "a" | "f" | "v")
  | "\u" xDigit xDigit xDigit xDigit
  | "\U" xDigit xDigit xDigit xDigit xDigit xDigit xDigit xDigit
  | "\" dDigit dDigit dDigit

(*Literal strings*)
string = '"' { stringChar }  '"'
```

```
stringChar = char − '"'
verbatimString = '@"' {(char − ('"' | '\"' )) | '""'} '"'

(*Constants*)
const :=
  byte
  | sbyte
  | int32
  | uint32
  | int
  | ieee64
  | char
  | string
  | verbatimString
  | "false"
  | "true"
  | "()"

(*Operators*)
infixOrPrefixOp := "+" | "−" | "+." | "−." | "%" | "&" | "&&"
prefixOp = infixOrPrefixOp | "~" {"~"} | "!" {opChar} − "!="
infixOp =
  {"."} (
    infixOrPrefixOp
    | "−" {opChar}
    | "+" {opChar}
    | "||"
    | "<" {opChar}
    | ">" {opChar}
    | "="
    | " |" {opChar}
    | "&" {opChar}
    | "^" {opChar}
    | "*" {opChar}
    | "/" {opChar}
    | "%" {opChar}
    | "!=" )
  | ":=" | "::" | "$" | "?" (*$*)
opChar =
  "!" | "%" | "&" | "*" | "+" | "−" | ". " | "/"
  | "<" | "=" | ">" | "@" | "^" | "|" | "~"

(*Identifiers*)
ident = (letter | specialChar) {letter | dDigit | specialChar}
letter =
  "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M"
  | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "X" | "Y" |  "Z"
  | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m"
  | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "x" | "y" | "z"
specialChar = "_"

longIdent = ident | ident '.' longIdent (*no space around '.'*)
longIdentOrOp = [longIdent '.'] identOrOp (*no space around '.'*)
identOrOp =
  ident
  | "(" infixOp | prefixOp ")"
  | "(*)"

(*Keywords*)
```

```
identKeyword =
  "abstract" | "and" | "as" | "assert" | "base" | "begin" | "class" | "default"
  | "delegate" | "do" | "done" | "downcast" | "downto" | "elif" | "else" | "end"
  | "exception" | "extern" | "false" | "finally" | "for" | "fun" | "function"
  | "global" | "if" | "in" | "inherit" | "inline" | "interface" | "internal"
  | "lazy" | "let" | "match" | "member" | "module" | "mutable"
  | "namespace" | "new" | "null" | "of" | "open" | "or" | "override" | "private"
  | "public" | "rec" | "return" | "sig" | "static" | "struct" | "then" | "to"
  | "true" | "try" | "type" | "upcast" | "use" | "val" | "void" | "when"
  | "while" | "with" | "yield"

reservedIdentKeyword =
  "atomic" | "break" | "checked" | "component" | "const" | "constraint"
  | "constructor" | "continue" | "eager" | "fixed" | "fori" | "functor"
  | "include" "measure" | "method" | "mixin" | "object" | "parallel"
  | "params" | "process" | "protected" | "pure" | "recursive" | "sealed"
  | "tailcall" | "trait" | "virtual" | "volatile"

reservedIdentFormats = ident ( '!' | '#')

(*Symbolic Keywords*)
symbolicKeyword =
  "let!" | "use!" | "do!" | "yield!" | "return!" | "|" | "->" | "<-" | "." | ":"
  | "(" | ")" | "[" | "]" | "[<" | ">]" | "[|" | "|]" | "{" | "}" | "'" | "#"
  | ":?>" | ":?" | ":>" | ".." | "::" | ":=" | ";;" | ";" | "=" | "_" | "?"
  | "??" | "(*)" | "<@" | "@>" | "<@@" | "@@>"
reservedSymbolicSequence =  "~" | "'"

(*Comments*)
blockComment = "(*" {codePoint} "*)" (*codePoint is any unicode codepoint*)
lineComment = "//" {codePoint - newline} newline

(*Expressions*)
expr =
  | const (*a const value*)
  | "(" expr ")" (*block*)
  | longidentOrOp (*identifier or operator*)
  | expr '.' longIdentOrOp (*dot lookup expression, no space around '.'*)
  | expr expr (*application*)
  | expr infixOp expr (*infix application*)
  | prefixOp expr (*prefix application*)
  | expr ".[" expr "]" (*index lookup, no space before '.'*)
  | expr ".[" sliceRange "]" (*index lookup, no space before '.'*)
  | expr "<-" expr (*assingment*)
  | exprTuple (*tuple*)
  | "[" (exprSeq | rangeExpr) "]" (*list*)
  | "[|" (exprSeq | rangeExpr) "|]" (*array*)
  | expr ":" type (*type annotation*)
  | expr; expr (*sequence of expressions*)
  | "let" valueDefn "in" expr (*binding a value or variable*)
  | "let" ["rec"] functionDefn "in" expr (*binding a function or operator*)
  | "if" expr "then" expr {"elif" expr "then" expr} ["else" expr] (*conditional*)
  | "while" expr "do" expr ["done"] (*while*)

exprTuple = expr | expr "," exprTuple
exprSeq =  expr | expr ";" exprSeq
rangeExpr = expr ".." expr [".." expr]
sliceRange =
  expr
  | expr ".." (*no space between expr and ".."*)
  | ".." expr (*no space between expr and ".."*)
  | expr ".." expr (*no space between expr and ".."*)
  | '*'

(*Types*)
type =
  | longIdent (*named such as "int"*)
  | "(" type ")" (*paranthesized*)
  | type "->" type (*function*)
  | typeTuple (*tuple*)
  | typar (*variable*)
  | type longIdent (*named such as "int list"*)
  | type "[" typeArray "]" (*array, no spaces*)
typeTuple = type | type "*" typeTuple
typeArray = "," | "," typeArray

(*Patterns*)
pat =
  const (*constant*)
  | "_" (*wildcard*)
  | ident (*named*)
  | pat ":" type (*type constraint*)
  | "(" pat ")" (*paranthesized*)
  | patTuple (*tuple*)
  | patList (*list*)
  | patArray (*array*)

patTuple = pat | pat "," patTuple
patList := "[" [patSeq] "]"
patArray := "[|" [patSeq] "|]"
patSeq = pat | pat ";" patSeq

(*Value bindings*)
valueDefn = ["mutable"] pat "=" expr

(*Function bindings*)
functionDefn = identOrOp argumentPats [":" type] "=" expr
argumentPats = pat | pat argumentPats
```
[1]

[1] **Somewhere I should possibly talk about Lightweight Syntax, Spec-4.0 Chapter 15.1**

Operator	Associativity	Description
ident "<"types ">"	Left	High-precedence type application
ident "("expr ")"	Left	High-predence application
"."	Left	
prefixOp	Left	All prefix operators
"" rule|	Left	Pattern matching rule
ident expr, "lazy'' expr, "assert'' epxr	Left	
"**"opChar	Right	Exponent like
"*"opChar, "/"opChar, "%"opChar	Left	Infix multiplication like
"−"opChar, "+"opChar	Left	Infix addition like
":?''	None	
"::''	Right	
"^'' opChar	Right	
"!="opChar, "<"opChar, ">"opChar, "=", "|"opChar, "&"opChar, "$"opChar	Left	Infix addition like
":>", ":?>"	Right	
"&", "&&"	Left	Boolean and like
"or", "||"	Left	Boolean or like
","	None	
":="	Right	
"−>"	Right	
"if"	None	
"function", "fun", "match", "try"	None	
"let"	None	
";"	Right	
"|"	Left	
"when"	Right	
"as"	Right	

Table 4.1: Precedence and associativity of operators. Operators in the same row has same precedence. See Listing 6.28 for the definition of `prefixOp`

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

.[], 28
abs, 20
acos, 20
asin, 20
atan2, 20
atan, 20
bignum, 17
byte[], 17
byte, 17
ceil, 20
char, 14
cosh, 20
cos, 20
decimal, 17
double, 17
eprintfn, 41
eprintf, 41
exn, 14
exp, 20
failwithf, 41
float32, 17
float, 14
floor, 20
fprintfn, 41
fprintf, 41
ignore, 41
int16, 17
int32, 17
int64, 17
int8, 17
int, 14
it, 14
log10, 20
log, 20
max, 20
min, 20
nativeint, 17
obj, 14
pown, 20
printfn, 41
printf, 40, 41
round, 20
sbyte, 17
sign, 20
single, 17
sinh, 20
sin, 20
sprintf, 41
sqrt, 20
stderr, 41
stdout, 41
string, 14
tanh, 20
tan, 20
uint16, 17
uint32, 17
uint64, 17
uint8, 17
unativeint, 17
unit, 14

American Standard Code for Information Interchange, 106
and, 24
anonymous function, 37
array sequence expressions, 73
Array.toArray, 68
Array.toList, 68
ASCII, 106
ASCIIbetical order, 27, 106

base, 14, 102
Basic Latin block, 107
Basic Multilingual plane, 107
basic types, 14
binary, 102
binary number, 16
binary operator, 20
binary64, 102
binding, 10
bit, 16, 102
black-box testing, 75
block, 34
blocks, 107
boolean and, 23
boolean or, 23
branches, 54
branching coverage, 76
bug, 74
byte, 102

character, 16
class, 19, 28
code point, 16, 107
compiled, 8
computation expressions, 62, 65
conditions, 54
Cons, 65
console, 8
coverage, 76
currying, 38

debugging, 9, 75
decimal number, 14, 102
decimal point, 14, 102
Declarative programming, 5
digit, 14, 102
dot notation, 28
double, 102
downcasting, 19

EBNF, 14, 110
efficiency, 74
encapsulate code, 35
encapsulation, 38, 43
exception, 26
exclusive or, 26
executable file, 8
expression, 10, 19
expressions, 6
Extended Backus-Naur Form, 14, 110
Extensible Markup Language, 46

floating point number, 14
format string, 10
fractional part, 14, 19
function, 12
function coverage, 76
Functional programming, 6, 87
functionality, 74
functions, 6

generic function, 36

Head, 65
hexadecimal, 102
hexadecimal number, 16
HTML, 48
Hyper Text Markup Language, 48

IEEE 754 double precision floating-point format, 102
Imperativ programming, 87
Imperative programming, 5
implementation file, 8
infix notation, 23
infix operator, 19
integer division, 25
integer number, 14
interactive, 8
IsEmpty, 65
Item, 65

jagged arrays, 68

keyword, 10

Latin-1 Supplement block, 107
Latin1, 106
least significant bit, 102
Length, 65
length, 60
lexeme, 12
lexical scope, 12, 36
lexically, 32
lightweight syntax, 30, 32
list, 62
list sequence expression, 73
List.Empty, 65
List.toArray, 65
List.toList, 65
literal, 14
literal type, 17

machine code, 87
maintainability, 75
member, 19, 60
method, 28
module elements, 98
modules, 8
most significant bit, 102
Mutable data, 42

namespace, 19
namespace pollution, 94
NaN, 104
nested scope, 12, 34
newline, 17
not, 24
not a number, 104

obfuscation, 62
object, 28
Object oriented programming, 87
Object-orientered programming, 6
objects, 6
octal, 102
octal number, 16
operand, 35
operands, 20
operator, 20, 23, 35

or, 24
overflow, 25
overshadow, 12
overshadows, 34

pattern matching, 55, 64
portability, 75
precedence, 23
prefix operator, 20
Procedural programming, 87
procedure, 38
production rules, 110

ragged multidimensional list, 65
range expression, 63
reals, 102
recursive function, 57
reference cells, 44
reliability, 74
remainder, 25
rounding, 19
run-time error, 26

scientific notation, 16
scope, 12, 33
script file, 8
script-fragments, 8
Seq.initInfinite, 73
Seq.item, 71
Seq.take, 71
Seq.toArray, 73
Seq.toList, 73
side-effect, 67
side-effects, 38, 44
signature file, 8
slicing, 68
software testing, 75
state, 5
statement, 10
statement coverage, 76
statements, 5, 87
states, 87
stopping criterium, 57
string, 10, 16
Structured programming, 6
subnormals, 104

Tail, 65
tail-recursive, 57
terminal symbols, 110
truth table, 24
tuple, 60
type, 10, 14
type casting, 18
type declaration, 10
type inference, 9, 10
type safety, 36

unary operator, 20
underflow, 25
Unicode, 16
unicode general category, 107
Unicode Standard, 107
unit of measure, 93
unit testing, 75
unit-less, 94
unit-testing, 9
upcasting, 19
usability, 74
UTF-16, 107
UTF-8, 107

variable, 42
verbatim, 18

white-box testing, 75, 76
whitespace, 17
whole part, 14, 19
word, 102

XML, 46
xor, 26

yield bang, 71