Learning to program with F#

Jon Sporring

August 3, 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

II Imperative programming 74

10 Exceptions 76
10.1 Exception Handling . . . 76

11 Testing programs 77

12 Input/Output 78
12.1 Console I/O . . . 78
12.2 File I/O . . . 78

13 Graphical User Interfaces 80

14 Imperative programming 81
14.1 Introduction . . . 81
14.2 Generating random texts . . . 81
14.2.1 0'th order statistics . . . 81
14.2.2 1'th order statistics . . . 83

III Declarative programming 86

15 Types and measures 87
15.1 Unit of Measure . . . 87

16 Functional programming 90

IV Structured programming 91

17 Namespaces and Modules 92

18 Object-oriented programming 94

V Appendix 95

A Number systems on the computer 96
A.1 Binary numbers . . . 96
A.2 IEEE 754 floating point standard . . . 96

B Commonly used character sets 100
B.1 ASCII . . . 100
B.2 ISO/IEC 8859 . . . 100
B.3 Unicode . . . 101

C A brief introduction to Extended Backus-Naur Form 104

D Language Details 107

E The Collection 109
E.1 System.String . . . 109
E.2 List, arrays, and sequences . . . 114
E.3 Mutable Collections . . . 116
E.3.1 Mutable lists . . . 116
E.3.2 Stacks . . . 116
E.3.3 Queues . . . 116

E.3.4 Sets and dictionaries 117

Bibliography **118**

Index **119**

Chapter 11

Testing programs

A software bug is an error in a computer program that causes it to produce an incorrect result or behave in an unintented manner. The term bug was used by Thomas Edison in 1878[1], but made popular in computer science by Grace Hopper, who found a moth interferring with the electronic circuits of the Harward Mark II electromechanical computer and coined the term *bug* for errors in computer programs. The original bug is shown in Figure 11.1.

· bug

To illustrate basic concepts of software quality consider a hypothetical route planning system. Essential factors of its quality is,

· functionality

Functionality: Does the software compile and run without internal errors. Does it solve the problem, it was intended to solve? E.g., does the route planning software finde a suitable route from point a to b?

· reliability

Reliability: Does the software work reliably over time? E.g., does the route planning software work in case of internet dropouts?

· usability

Usability: Is the software easy and intuitive to use by humans? E.g., is it easy to enter adresses and alternative routes in the software's interface?

· efficiency

Efficiency: How many computer and human resources does the software require? E.g., does it take milliseconds or hours to find a requested route? Can the software run on a mobile platform with limited computer speed and memory?

· maintainability

[1] `https://en.wikipedia.org/wiki/Software_bug`, possibly `http://edison.rutgers.edu/NamesSearch/DocImage.php3?DocId=LB003487`

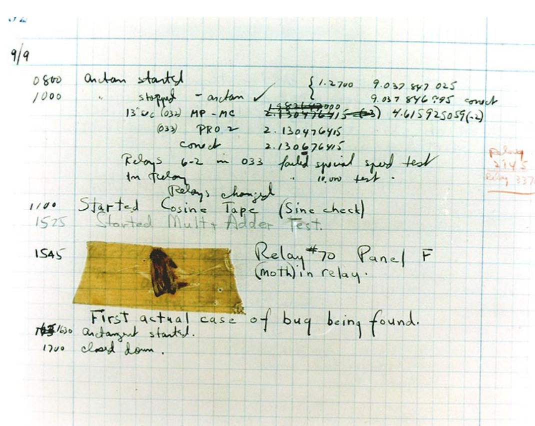

Figure 11.1: The first computer bug caught by Grace Hopper, U.S. Naval Historical Center Online Library Photograph NH 96566-KN.

Maintainability: In case of the discovery of new bugs, is it easy to test and correct the software? Is it easy to extend the software with new functionality? E.g., is it easy to update the map with updated roadmaps and new information? Can the system be improved to work both for car drivers and bicyclists?

· portability

Portability: Is it easy to port the software to new systems such as new server architecture and screen sizes? E.g., if the routing software originally was written for IOS devices, will it be easy to port to Android systems?

The above mentioned concepts are ordered based on the requirements of the system. Functionality and reliability ares perhaps the most important concepts, since if the software does not solve the specified problem, then the software designing process has failed. However, many times the problem definition will evolve along with the software development process. But as a bare minimum, the software should run without internal errors and not crash under well defined set of circumstances. Further, it is often the case, that software designed for the general public requires a lot of attention to the usability of the software, since in many cases non-experts are expected to be able to use the software little or no prior training. On the other hand, software used internally in companies will be used by a small number of people, who become experts in using the software, and it is often less important that the software is easy to understand by non-experts. An example is text processing software Microsoft Word versus Gnu Emacs and LaTeX. Word is designed to be used by non-experts for small documents such as letters and notes, and relies heavily on interfacing with the system using click-interaction. On the other hand, Emacs and LaTeX are for experts for longer and professionally typeset documents, and relies heavily on keyboard shortcuts and text-codes for typesetting document entities.

The purpose of *software testing* is to find bugs. For errors found we engage in *debugging*, which is the process of diagnosing and correcting bugs. Once we have a failed software test, i.e., one that does not find any bugs, then we have strengthened our belief in the software, but it is important to note, that software testing and debugging rarely removes all bugs, and with each correction or change of software, there is a fair chance of introducing new bugs.

· software testing
· debugging

In this chapter, we will focus on two approaches to software testing, which emphasizes functionality: *white-box* and *black-box testing*. An important concept in this context is *unit testing*, where the program is considered in smaller pieces, called units, and for which accompanying programs for testing can be made, which tests these units automatically.

· white-box testing
· black-box testing
· unit testing

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, 100
and, 24
anonymous function, 37
array sequence expressions, 73
Array.toArray, 68
Array.toList, 68
ASCII, 100
ASCIIbetical order, 27, 100

base, 14, 96
Basic Latin block, 101
Basic Multilingual plane, 101
basic types, 14
binary, 96
binary number, 16
binary operator, 20
binary64, 96
binding, 10
bit, 16, 96
block, 34
blocks, 101
boolean and, 23
boolean or, 23
branches, 54
byte, 96

character, 16
class, 19, 28
code point, 16, 101

compiled, 8
computation expressions, 62, 65
conditions, 54
Cons, 65
console, 8
currying, 38

debugging, 9
decimal number, 14, 96
decimal point, 14, 96
Declarative programming, 5
digit, 14, 96
dot notation, 28
double, 96
downcasting, 19

EBNF, 14, 104
encapsulate code, 35
encapsulation, 38, 43
exception, 26
exclusive or, 26
executable file, 8
expression, 10, 19
expressions, 6
Extended Backus-Naur Form, 14, 104
Extensible Markup Language, 46

floating point number, 14
format string, 10
fractional part, 14, 19
function, 12
Functional programming, 6, 81
functions, 6

generic function, 36

Head, 65
hexadecimal, 96
hexadecimal number, 16
HTML, 48
Hyper Text Markup Language, 48

IEEE 754 double precision floating-point format, 96
Imperativ programming, 81
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, 101
Latin1, 100
least significant bit, 96
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, 81
member, 19, 60
method, 28
module elements, 92
modules, 8
most significant bit, 96
Mutable data, 42

namespace, 19
namespace pollution, 88
NaN, 98
nested scope, 12, 34
newline, 17
not, 24
not a number, 98

obfuscation, 62
object, 28
Object oriented programming, 81
Object-orientered programming, 6
objects, 6
octal, 96
octal number, 16
operand, 35
operands, 20
operator, 20, 23, 35
or, 24
overflow, 25
overshadow, 12
overshadows, 34

pattern matching, 55, 64
precedence, 23
prefix operator, 20
Procedural programming, 81
procedure, 38

production rules, 104

ragged multidimensional list, 65
range expression, 63
reals, 96
recursive function, 57
reference cells, 44
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
state, 5
statement, 10
statements, 5, 81
states, 81
stopping criterium, 57
string, 10, 16
Structured programming, 6
subnormals, 98

Tail, 65
tail-recursive, 57
terminal symbols, 104
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, 101
Unicode Standard, 101
unit of measure, 87
unit-less, 88
unit-testing, 9
upcasting, 19
UTF-16, 101
UTF-8, 101

variable, 42
verbatim, 18

whitespace, 17
whole part, 14, 19
word, 96

XML, 46
xor, 26

yield bang, 71