

# Learning to program with F#

Jon Sparring

August 4, 2016

# Contents

<b>1</b>	<b>Preface</b>	<b>4</b>
<b>2</b>	<b>Introduction</b>	<b>5</b>
<b>I</b>	<b>F# basics</b>	<b>7</b>
<b>3</b>	<b>Executing F# code</b>	<b>8</b>
3.1	Source code . . . . .	8
3.2	Executing programs . . . . .	8
<b>4</b>	<b>Quick-start guide</b>	<b>10</b>
<b>5</b>	<b>Using F# as a calculator</b>	<b>14</b>
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
<b>6</b>	<b>Constants, functions, and variables</b>	<b>30</b>
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
<b>7</b>	<b>In-code documentation</b>	<b>46</b>
<b>8</b>	<b>Controlling program flow</b>	<b>50</b>
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
<b>9</b>	<b>Ordered series of data</b>	<b>59</b>
9.1	Tuples . . . . .	60
9.2	Lists . . . . .	62
9.3	Arrays . . . . .	65
9.4	Sequences . . . . .	70

<b>II</b>	<b>Imperative programming</b>	<b>74</b>
<b>10</b>	<b>Exceptions</b>	<b>76</b>
10.1	Exception Handling . . . . .	76
<b>11</b>	<b>Testing programs</b>	<b>77</b>
11.1	White-box testing . . . . .	79
11.2	Back-box testing . . . . .	81
<b>12</b>	<b>Input/Output</b>	<b>78</b>
12.1	Console I/O . . . . .	78
12.2	File I/O . . . . .	78
<b>13</b>	<b>Graphical User Interfaces</b>	<b>80</b>
<b>14</b>	<b>Imperative programming</b>	<b>81</b>
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
<b>III</b>	<b>Declarative programming</b>	<b>86</b>
<b>15</b>	<b>Types and measures</b>	<b>87</b>
15.1	Unit of Measure . . . . .	87
<b>16</b>	<b>Functional programming</b>	<b>90</b>
<b>IV</b>	<b>Structured programming</b>	<b>91</b>
<b>17</b>	<b>Namespaces and Modules</b>	<b>92</b>
<b>18</b>	<b>Object-oriented programming</b>	<b>94</b>
<b>V</b>	<b>Appendix</b>	<b>95</b>
<b>A</b>	<b>Number systems on the computer</b>	<b>96</b>
A.1	Binary numbers . . . . .	96
A.2	IEEE 754 floating point standard . . . . .	96
<b>B</b>	<b>Commonly used character sets</b>	<b>100</b>
B.1	ASCII . . . . .	100
B.2	ISO/IEC 8859 . . . . .	100
B.3	Unicode . . . . .	101
<b>C</b>	<b>A brief introduction to Extended Backus-Naur Form</b>	<b>104</b>
<b>D</b>	<b>Language Details</b>	<b>107</b>
<b>E</b>	<b>The Collection</b>	<b>109</b>
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
<b>Bibliography</b>		<b>118</b>
<b>Index</b>		<b>119</b>

# 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 unintended manner. The term bug was used by Thomas Edison in 1878<sup>1</sup>, but made popular in computer science by Grace Hopper, who found a moth interfering 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.

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

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

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

**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?

<sup>1</sup>[https://en.wikipedia.org/wiki/Software\\_bug](https://en.wikipedia.org/wiki/Software_bug), possibly <http://edison.rutgers.edu/NamesSearch/DocImage.php3?DocId=LB003487>

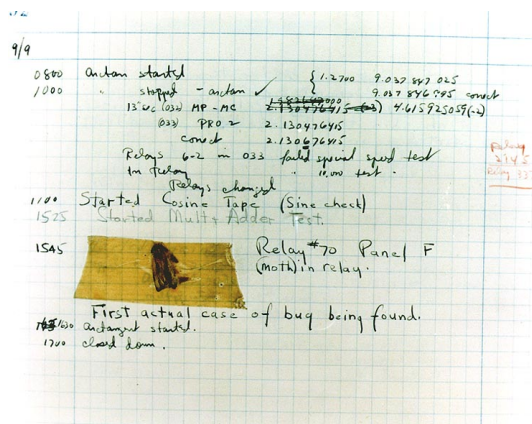


Figure 11.1: The first computer bug caught by Grace Hopper, U.S. Naval Historical Center Online Library Photograph NH 96566-KN.

**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 are 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

To illustrate software testing we'll start with a problem:

Given any date in the Gregorian calendar, calculate the day of week.

To solve the problem, we will use Gauss' algorithm for calculating the weekday of the 1st of January of a given year, and count from there. The algorithm relies on an enumeration of weekdays starting with Sunday = 0, Monday = 1, ..., and Saturday = 6. Our proposed solution is,

```
let januaryFirstDay (y : int) =
  let a = (y - 1) % 4
  let b = (y - 1) % 100
  let c = (y - 1) % 400
  (1 + 5 * a + 4 * b + 6 * c) % 7

let rec sum (lst : int list) j =
  if 0 <= j && j < lst.Length then
    lst.[0] + sum lst.[1..] (j - 1)
  else
    0

let date2Day d m y =
  let dayPrefix = ["Sun"; "Mon"; "Tues"; "Wednes"; "Thurs"; "Fri"; "Satur"]
  let feb = if (y % 4 = 0) && ((y % 100 <> 0) || (y % 400 = 0)) then 29 else 28
  let daysInMonth = [31; feb; 31; 30; 31; 30; 31; 31; 30; 31; 30; 31]
  let dayOne = januaryFirstDay y
```

```

let daysSince = (sum daysInMonth (m - 2)) + d - 1
let weekday = (dayOne + daysSince) % 7;
dayPrefix.[weekday] + "day"

```

**Listing 11.1:** A function that can calculate day-of-week from any date in the Gregorian calendar.

## 11.1 White-box testing

*White-box testing* considers the text of a program with the goal of testing every line. The procedure is as follows:

· white-box testing

1. Decide which are the units to test: The program shown in Listing 11.1 has 3 functions, and we will consider these each as a unit, but we might as well just have chosen `date2Day` as a single unit. The important part is that the union of units must cover the whole program text, and since `date2Day` calls both `januaryFirstDay` and `sum`, designing test cases for the two later is superfluous. However, we may have to do this anyway, when debugging, and we may choose at a later point to use these functions separately, and in both cases we will be able to reuse the testing of the smaller units.
2. Identify branching points: The function `januaryFirstDay` has no branching function, `sum` has one, and depending on the input values two paths through the code may be used, and `date2Day` has one, where the number of days in February is decided. Note that in order to test this, our test-date must be March 1 or later. In this example, there are only examples of `if`-branch points, but they may as well be loops and pattern matching expressions. In the following code, the branch points have been given a comment and a number,

```

// Unit: januaryFirstDay
let januaryFirstDay (y : int) =
    let a = (y - 1) % 4
    let b = (y - 1) % 100
    let c = (y - 1) % 400
    (1 + 5 * a + 4 * b + 6 * c) % 7

// Unit: sum
let rec sum (lst : int list) j =
    (* WB: 1 *)
    if 0 <= j && j < lst.Length then
        lst.[0] + sum lst.[1..] (j - 1)
    else
        0

// Unit: date2Day
let date2Day d m y =
    let dayPrefix = ["Sun"; "Mon"; "Tues"; "Wednes"; "Thurs"; "Fri"; "Satur"]
    (* WB: 1 *)
    let feb = if (y % 4 = 0) && ((y % 100 <> 0) || (y % 400 = 0)) then 29
               else 28
    let daysInMonth = [31; feb; 31; 30; 31; 30; 31; 31; 30; 31; 30; 31]
    let dayOne = januaryFirstDay y
    let daysSince = (sum daysInMonth (m - 2)) + d - 1
    let weekday = (dayOne + daysSince) % 7;
    dayPrefix.[weekday] + "day"

```

**Listing 11.2:** In white-box testing, the branch points are identified.

3. For each unit, produce an input set that tests each branches: In our example the branch points depends on a boolean expression, and for good measure, we are going to test each term that can lead to branching. Thus,

Unit	Branch	Condition	Input	Expected output
januaryFirstDay	0	-	2016	5
sum	1	<code>0 &lt;= j &amp;&amp; j &lt; lst.Length</code>		
	1a	<code>true &amp;&amp; true</code>	<code>[1; 2; 3] 1</code>	3
	1b	<code>false &amp;&amp; true</code>	<code>[1; 2; 3] -1</code>	0
	1c	<code>true &amp;&amp; false</code>	<code>[1; 2; 3] 10</code>	0
	1d	<code>false &amp;&amp; false</code>	-	-
date2Day	1	<code>(y % 4 = 0) &amp;&amp; ((y % 100 &lt;&gt; 0)    (y % 400 = 0))</code>		
	-	<code>true &amp;&amp; (true    true)</code>	-	-
	1a	<code>true &amp;&amp; (true    false)</code>	8 9 2016	Thursday
	1b	<code>true &amp;&amp; (false    true)</code>	8 9 2000	Friday
	1c	<code>true &amp;&amp; (false    false)</code>	8 9 2100	Wednesday
	-	<code>false &amp;&amp; (true    true)</code>	-	-
	1d	<code>false &amp;&amp; (true    false)</code>	8 9 2015	Tuesday
	-	<code>false &amp;&amp; (false    true)</code>	-	-
	-	<code>false &amp;&amp; (false    false)</code>	-	-

The impossible cases have been intentionally blank, e.g., it is not possible for  $j < 0$  and  $j > n$  for some positive value  $n$ .

4. Write a program, that test all these cases and checks the output, e.g.,

```
printfn "White-box testing of date2Day.fsx"
printfn "  Unit: januaryFirstDay"
printfn "    Branch: 0 - %b" (januaryFirstDay 2016 = 5)

printfn "  Unit: sum"
printfn "    Branch: 1a - %b" (sum [1; 2; 3] 1 = 3)
printfn "    Branch: 1b - %b" (sum [1; 2; 3] -1 = 0)
printfn "    Branch: 1c - %b" (sum [1; 2; 3] 10 = 0)

printfn "  Unit: date2Day"
printfn "    Branch: 1a - %b" (date2Day 8 9 2016 = "Thursday")
printfn "    Branch: 1b - %b" (date2Day 8 9 2000 = "Friday")
printfn "    Branch: 1c - %b" (date2Day 8 9 2100 = "Wednesday")
printfn "    Branch: 1d - %b" (date2Day 8 9 2015 = "Tuesday")
```

```
White-box testing of date2Day.fsx
Unit: januaryFirstDay
  Branch: 0 - true
Unit: sum
  Branch: 1a - true
  Branch: 1b - true
  Branch: 1c - true
Unit: date2Day
  Branch: 1a - true
  Branch: 1b - true
  Branch: 1c - true
  Branch: 1d - true
```



---

Listing 11.3: `date2DayWhiteTest.fsx` - The tests identified by white-box analysis. The program from Listing 11.2 has been omitted for brevity.

Notice, that the output of the tests are organized such that they are enumerated per unit, hence we can rearrange as we like and still uniquely refer to a unit's test. Also, the output of the test program produces a list of tests, that should return true or success or a similar positively loaded word, but without further or only little detail, such that we at a glance can identify any test that produced unexpected results.

After the white-box testing has failed to find errors in the program, we have some confidence in the program, since we have run every line at least once. It is, however, in no way a guarantee, that the program is error free, which is why white-box testing is often accompanied with black-box testing to be described next.

## 11.2 Back-box 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 Inter-  
change, 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-oriented 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