Learning to program with F#

Jon Sporring & Torben Mogensen

Department of Computer Science,
University of Copenhagen

June 28, 2017

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 24

5.3 Boolean arithmetic 26

5.4 Integer arithmetic 27

5.5 Floating point arithmetic 29

5.6 Char and string arithmetic 31

5.7 Programming intermezzo 32

6 Constants, functions, and variables **34**

6.1 Values . . . 37

6.2 Non-recursive functions . . . 42

6.3 User-defined operators . . . 46

6.4 The Printf function . . . 48

6.5 Variables . . . 51

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 . . . 84

10 Exceptions **89**

11 Input and Output **97**

11.1 Interacting with the console . . . 98

11.2 Storing and retrieving data from a file . . . 99

11.3 Working with files and directories. . . . 104

11.4 Reading from the internet . . . 104

11.5 Programming intermezzo . . . 105

II Imperative programming **108**

12 Graphical User Interfaces **110**

12.1 Drawing primitives in Windows . . . 110

12.2 Programming intermezzo . . . 123
12.3 Events, Controls, and Panels . . . 125

13 Imperative programming **150**
13.1 Introduction . . . 150
13.2 Generating random texts . . . 151
13.2.1 0'th order statistics . . . 151
13.2.2 1'th order statistics . . . 151

III Declarative programming **152**

14 Sequences and computation expressions **153**
14.1 Sequences . . . 153

15 Patterns **159**
15.1 Pattern matching . . . 159

16 Types and measures **162**
16.1 Unit of Measure . . . 162

17 Functional programming **166**

IV Structured programming **169**

18 Namespaces and Modules **170**

19 Object-oriented programming **172**
19.1 Object-oriented Analysis . . . 172

V Appendix **173**

A Number systems on the computer **174**
A.1 Binary numbers . . . 176
A.2 IEEE 754 floating point standard . . . 176

B Commonly used character sets **177**

B.1 ASCII . . . 177

B.2 ISO/IEC 8859 . . . 178

B.3 Unicode . . . 178

C A brief introduction to Extended Backus-Naur Form **182**

D F♭ **186**

E Language Details **191**

E.1 Arithmetic operators on basic types . . . 191

E.2 Basic arithmetic functions . . . 194

E.3 Precedence and associativity . . . 195

E.4 Lightweight Syntax . . . 197

F The Some Basic Libraries **198**

F.1 `System.String` . . . 199

F.2 List, arrays, and sequences . . . 199

F.3 WinForms Details . . . 203

F.4 Mutable Collections . . . 203

F.4.1 Mutable lists . . . 203

F.4.2 Stacks . . . 203

F.4.3 Queues . . . 203

F.4.4 Sets and dictionaries . . . 203

7 To Dos **204**

Bibliography **205**

Index **206**

Part IV

Structured programming

Chapter 18

Namespaces and Modules

Things to remember:

- difference between .fs and .fsx Spec-4.0 Chapter 12.1 and 12.3
- signature files and their usefulness

A script file consists of a sequence of *module elements* · module elements

```
script-file = implementation-file

implementation-file =
  namespace-decl-groupList
  | named-module
  | anonynmous-module

namespace-decl-groupList = namespace-decl-group | namespace-decl-group namespace-
    decl-groupList

named-module = "module" long-ident module-elems

anonymous-module = module-elems

module-elems = module-elem | module-elem module-elems

namespace-decl-group = "namespace" long-ident module-elems | "namespace" global
    module-elems

module-elem =
  module-function-or-value-defn type-defns
  | exception-defn
  | module-defn
  | module-abbrev
  | import-decl compiler-directive-decl
```

F# source code units are made up of declarations grouped using namespaces, type definitions, and module definitions. A file may contain multipe namespaces each defining types and modules, these in turn may contain function and value definitions, which in turn contains expressions.[1]

[1] Todo: **Spec-4.0 Chapter 10.**

With no leading namespace or module declaration, then F# will immediately insert a module, where the name of the module is the same as the file name with capitalized first letter.[2]

Namespaces is an optional hierarchial catergorization of modules, classes, and other namespaces primarily used to avoid naming conflicts. There is no default namespace, and namespaces may contain type definitions but not function and value definitions. Namespace do not work in script-fragments.[3]

4 5

[2]Todo: https://en.wikibooks.org/wiki/F_Sharp_Programming/Modules_and_Namespaces

[3]Todo: https://fsharpforfunandprofit.com/posts/organizing-functions/

[4]Todo: **Difference between namespaces and modules** https://stackoverflow.com/questions/795172/what-the-difference-between-a-namespace-and-a-module-in-f

[5]Todo: https://fsharpforfunandprofit.com/posts/organizing-functions/, https://fsharpforfunandprofit.com/posts/recipe-part3/

Chapter 19

Object-oriented programming

Object-oriented programming is a programming paradigm that focusses on *objects* such as a person, place, thing, event, and concept relevant for the problem. Objects may contain data and code, which in the object-oriented paradigm are called *attributeds* and *methods*. Object-oriented programming is an extension of data types, in the sense that objects contains both data and functions in a similar manner as a module, but object-oriented programming emphasizes the semantic unity of the data and functions. Thus, objects are *models* of real world entities, and object-oriented programming leads to a particular style of programming analysis and design called *object-oriented analysis and design*.

· Object-oriented programming
· objects
· attributeds
· methods
· models
· object-oriented analysis
· object-oriented design

Before we dive into the details of the language support for object-oriented programming ing F#, we will first introduce central elements of object-oriented analysis and design. The analysis serves as input to the design phase, where the analysis reveals *what* a program is supposed to do, and the design *how* it is supposed to be doing i. The analysis should be expressed in general terms irrespective of the technologic constraints, while the design should include technological constraints such as defined by the targeted language and hardware.

· what
· how

19.1 Object-oriented Analysis

The primary task of *object-oriented analysis* is to

· object-oriented analysis

- identify objects,
- describe object behaviour,
- describe object interactions, and
- describe some details of the object's inner workings.

We will now illustrate, how an object-oriented analysis could be performed by applying the above tasks to the for the above problem. Consider the following *problem statement*:

· problem statement

Problem 19.1:

Write a racing game, where each player controls his or her vehicle on a set track. Each vehicle must have individual features such as top acceleration, speed, and handling. The player must be able to turn the vehicle left and right, and to accelerate up and down. At the beginning of the game, each vehicle is placed behind the starting line. Once the start signal is given, then the players may start to operate their vehicles. The player who first completes 3 rounds wins.

Identification of objects: To identify objects we seek relevant persons, places, things, events, concept etc., which are almost always characterized by being *nouns* in the text. E.g., in the above the following nouns seems relevant:

· nouns

game, player, vehicle, track, feature, starting line, start signal

In the object-oriented paradigm, the objects has a type, which is called a *class*, and each value or variable of a particular class is called and *instance* of a class or simply just and *object*. Many languages include F# include support for *static* attributes and methods, which essentially implies that the class reverts to becoming a name spaces or a module, but we will ignore for the moment.

· class
· instance
· object
· static

A key point in object-oriented programming is that objects should to a large extend be independent and reusable. As an example the type `int` models the concept of integer numbers. It can hold integer values from -2,147,483,648 to 2,147,483,647, and a number of standard operations and functions are defined for it. We may use integers in many different programs, and it is certain that the original designers did not foresee our use, but strived to make a general type applicable for many uses. Such a design is a useful goal, when designing objects, that is, our objects should model the general concepts and be applicable in future uses.

Object behaviour and interactions: We are still far from having a program design, that we can implement in F#. To continue our object-oriented design, Let's consider some of the object candiate identified above, and verbalize how they would act as models of general concepts useful in our game.

player A player interacts with the game and could be a human or computer player. A player must in general be able to control the vehicle and receive information about the track and all vehicles or at least some information about the nearby vehicles and track. And the player must receive information about the state of the game, i.e., when does the race start and stop.

vehicle A vehicle is a model of a physical object, which moves around on the track under the influence of a player. A vehicle must have a number of attributes such as top acceleration, speed, and handling, and must be able to receive information about when to turn and accelerate. A vehicle must be able to determine its location in particular if it is on or off track and, and it must be able to determine if it has crashed into an obstacle such as another vehicle.

track A track is a fixed entity on which they vehicles race. It has a size and a shape, a starting and a finishing line, which may be the same, and vehicles are placeable on the track and can move on and possibly off the track.

From the above we see that the object candidates 'feature' seems to be a natural part of the description of the vehicle's attributes, and similarly, 'starting line' may be an intricate part of a track. Also, many of the *verbs* used in the problem statement and in our extended verbalisation of the general concepts indicate methods that are used to interact with the object. Here it is important to maintain an object centered perspective, i.e., for a general purpose vehicle object, we need not include information about the player, analogous to a value of type `int` need not know anything the program, in which it is being used. In contrast, the candidate 'game' is not as easily dismissed and could be used as a class which contains all the above, i.e.,

· verbs

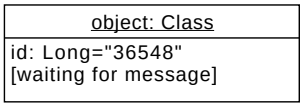

Figure 19.1: An example of UML, produced by using UMLet [1]

game A game is the total sum of all the players, the vehicles, the tracks, and their interactions. A game controls the flow of a particular game including inviting players to race, sending the start signal, and monitoring when a game is finished and who won.

With this description we see that 'start signal' can be included as a natural part of the game object. Being confident that a good working hypothesis of the essential objects for the solution, we continue our investigating into further details about the objects and their interactions.

Analysis details: To describe the objects and their interaction we will use a *class diagram*. A class diagram is a schematic drawing of the program highlighting its object-oriented structure and we will use the *Universal Modelling Language 2* (*UML*) [3] standard. A class is drawn as a

· class diagram

· Universal Modelling Language 2

· UML

Things to remember:

- upcast and downcast "`upcast`", "`:>`", "`downcast`", "`:?>`"
- boxing `(box 5) :?> int;;`, see Spec-4.0 chapter 18.2.6.
- obj type Spec-4.0 chapter 18.1
- boxing Spec-4.0 Section 18.2.6

[1]

[1]Todo: **In object oriented programming: functions and data are combined. Contrast the Anemic Domain Model (`https://www.martinfowler.com/bliki/AnemicDomainModel.html`)**

Chapter 7

To Dos

- Remove EBNF from main body of the text, possibly extend the appendix
- Add appendix on regular expressions
- Add Torben's notes on functional programming
- Rewrite list chapter (add sequences?)
- Add a chapter comparing the 3 paradigms
- Write structured programming part
- Write chapter on pattern matching (if not already in Torben's notes)
- Move modules and namespaces earlier
- Should we add something about assemblies (`https://msdn.microsoft.com/en-us/library/hk5f40ct(v=vs.90).aspx`, `https://msdn.microsoft.com/en-us/library/ms973231.aspx`, `https://stackoverflow.com/questions/2972732/what-are-net-assemblies`)
- Add something on piping (if not already in Torben's notes)
- Add abstraction of computer: places <-> memory/disk. Mutable objects are abstractions of places `https://www.infoq.com/presentations/Value-Values`. Facts does not rime with set and get.
- Hickey: Difference between syntax and semantics. Values or locations, add a good figure. Functional programming: All values are freely shareable.

Bibliography

[1] M. Auer, J. Poelz, A. Fuernweger, L. Meyer, and T. Tschurtschenthaler. Umlet, free uml tool for fast uml diagrams. `http://www.umlet.com`, Version 1.0 was released June 21, 2002.

[2] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[3] Object Management Group. Uml version 2.0. `http://www.omg.org/spec/UML/2.0/`.

[4] Programming Research Group. Specifications for the ibm mathematical formula translating system, fortran. Technical report, Applied Science Division, International Business Machines Corporation, 1954.

[5] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[6] 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/`.

[7] George Pólya. *How to solve it*. Princeton University Press, 1945.

Index

attributeds, 172

class, 173
class diagram, 174

how, 172

instance, 173

methods, 172
models, 172
module elements, 170

nouns, 173

object, 173
object-oriented analysis, 172
object-oriented design, 172
Object-oriented programming, 172
objects, 172

problem statement, 172

static, 173
structure diagram, 174

UML, 174
Universal Modelling Language 2, 174

verbs, 173

what, 172