

Learning to program with F#

Jon Sparring

November 16, 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	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	Testing programs	89
10.1	White-box testing	92
10.2	Black-box testing	95
10.3	Debugging by tracing	98
11	Exceptions	105
12	Input and Output	113
12.1	Interacting with the console	114
12.2	Storing and retrieving data from a file	115
12.3	Working with files and directories.	120
12.4	Reading from the internet	120
12.5	Programming intermezzo	121

II	Imperative programming	124
13	Graphical User Interfaces	126
13.1	Drawing primitives in Windows	126
13.2	Programming intermezzo	137
13.3	Buttons and stuff	141
14	Imperative programming	142
14.1	Introduction	142
14.2	Generating random texts	143
14.2.1	0'th order statistics	143
14.2.2	1'th order statistics	143
III	Declarative programming	144
15	Sequences and computation expressions	145
15.1	Sequences	145
16	Patterns	151
16.1	Pattern matching	151
17	Types and measures	154
17.1	Unit of Measure	154
18	Functional programming	158
IV	Structured programming	161
19	Namespaces and Modules	162
20	Object-oriented programming	164
V	Appendix	165
A	Number systems on the computer	166

A.1	Binary numbers	168
A.2	IEEE 754 floating point standard	168
B	Commonly used character sets	169
B.1	ASCII	169
B.2	ISO/IEC 8859	170
B.3	Unicode	170
C	A brief introduction to Extended Backus-Naur Form	174
D	F_b	178
E	Language Details	183
E.1	Arithmetic operators on basic types	183
E.2	Basic arithmetic functions	186
E.3	Precedence and associativity	187
E.4	Lightweight Syntax	189
F	The Some Basic Libraries	190
F.1	<code>System.String</code>	191
F.2	List, arrays, and sequences	191
F.3	Mutable Collections	194
F.3.1	Mutable lists	194
F.3.2	Stacks	194
F.3.3	Queues	194
F.3.4	Sets and dictionaries	194
	Bibliography	195
	Index	196

Chapter 13

Graphical User Interfaces

A *command-line interface (CLI)* is a method for communicating with the user through text. In contrast, a *graphical user interface (GUI)* extends the ways of communicating with the user to also include organising the screen space in windows, icons, and other visual elements, and a typical way to activate these elements are through a pointing device such as the mouse or by touch. Some of these elements may themselves be textual, and thus most operating systems offers access to a command-line interface in a window alongside other interface types.

- command-line interface
- CLI
- graphical user interface
- GUI

Fsharp includes a number of implementations of graphical user interfaces, but at time of writing only *WinForms* is supported on both the Microsoft .Net and the Mono platform, and hence, WinForms will be the subject of the following chapter.

- WinForms

WinForms is designed for *event driven programs*, which spends most time waiting for the user to perform an action, called an event, and for each event has a set of predefined responses to be performed by the program. For example, Figure 13.1 shows the program Safari, which is a graphical user interface for accessing web-servers. The program present information to the user in terms of text and images, and has areas that when activated by clicking with a mouse or similar allows the user to, e.g., go to other web-pages by type URL, to follow hyperlinks, and to generate new pages by entering search queries.

- event driven programs

13.1 Drawing primitives in Windows

WinForms is based on two namespaces: `System.Windows.Forms` and `System.Drawing`. To start making a graphical display on the screen, the first thing to do is open a window, which acts as a reserved screen space for our output. With WinForms, this may be done as shown in Listing 13.1, and the result is shown in Figure 13.3.

Listing 13.1, winforms/openWindow.fsx:
Create the window and turn over control to the operating system.

```
1 // Create a window
2 let win = new System.Windows.Forms.Form ()
3 // Start the event-loop.
4 System.Windows.Forms.Application.Run win
```

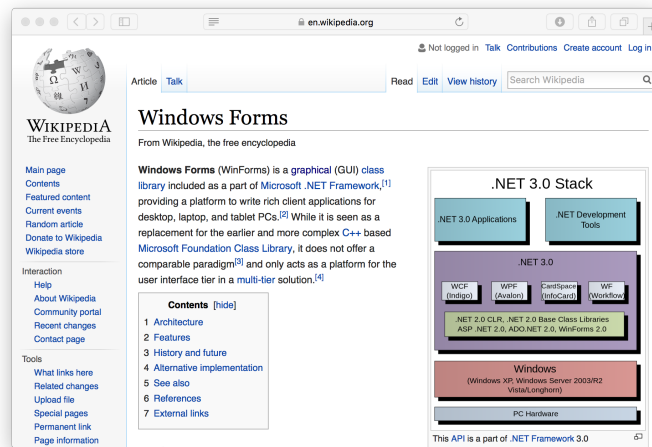


Figure 13.1: A web-browser is a graphical user interface for accessing a web-server and interacting with its services. Here the browser is showing the page https://en.wikipedia.org/wiki/Windows_Forms at time of writing.

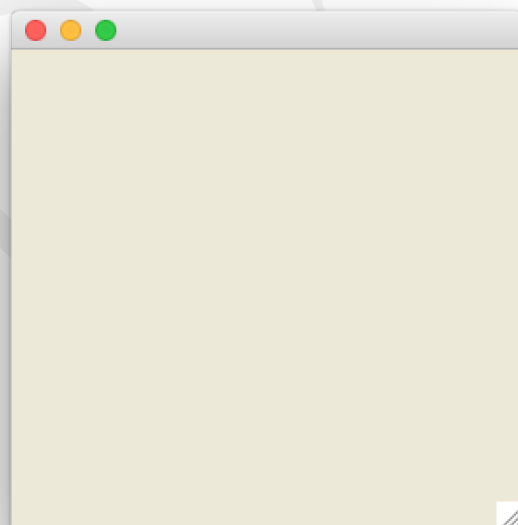


Figure 13.2: A window opened by Listing 13.1.

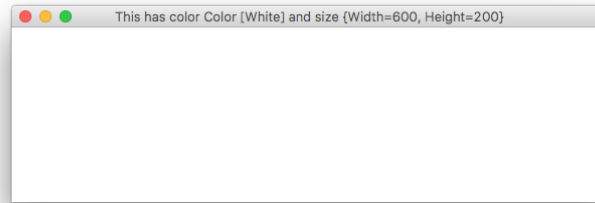


Figure 13.3: A window with user-specified size and background color, see Listing 13.2.

The `new System.Windows.Forms.Form ()` creates an object (See Chapter 20), but does not display the window on the screen. When the function `System.Windows.Forms.Application.Run` is applied to the object, then the control is handed over to the WinForm's *event-loop*, which continues until the window is closed by, e.g., pressing the icon designated by the operating system. On the mac OSX that is the red button in the top left corner of the window frame, and on Window it is the cross on the top right corner of the window frame.

The window, which WinForms calls a form, has a long list of *methods* and *properties*. E.g., the background color may be set by `BackColor`, the title of the window may be set by `Text`, and you may get and set the size of the window with the `Size`. This is demonstrated in Listing

Listing 13.2, winforms/windowAttributes.fsx:
Create the window and changing its properties.

```
1 // Create a window
2 let win = new System.Windows.Forms.Form ()
3 // Set some properties
4 win.BackColor <- System.Drawing.Color.White
5 win.Size <- System.Drawing.Size (600, 200)
6 win.Text <- sprintf "This has color %A and size %A" win.BackColor win.Size
7 // Start the event-loop.
8 System.Windows.Forms.Application.Run win
```

These properties have been programmed as *accessors* implying that they may be used as mutable variables. The `System.Drawing.Color` is a general structure for specifying colors as 4 channels: alpha, red, green, blue, where each channel is an 8 bit unsigned integer, where the alpha channel specifies the transparency of a color, where values 0–255 denotes the range of fully transparent to fully opaque, and the remaining channels denote the amount of red, green, and blue where 0 is none and 255 is full intensity. Any color may be created using the `FromArgb` method, e.g., an opaque red is given by `System.Drawing.Color.FromArgb (255, 255, 0, 0)`. There are also many build-in colors, e.g., the same red color is also a known color and may be obtained as `System.Drawing.Color.Red`. For a given color, then the 4 alpha, red, green, and blue channel's values may be obtained as the A, R, G, B, see Listing 13.3

Listing 13.3, drawingColors.fsx:
Defining colors and accessing their values.

```
1 // open namespace for brevity
2 open System.Drawing
3 // Define a color from ARGB
4 let c = Color.FromArgb (0xFF, 0x7F, 0xFF, 0xD4) //Aquamarine
5 printfn "The color %A is (%x, %x, %x, %x)" c c.A c.R c.G c.B
6 // Define a list of named colors
7 let colors = [Color.Red; Color.Green; Color.Blue; Color.Black; Color.Gray;
               Color.White]
8 for col in colors do
9     printfn "The color %A is (%x, %x, %x, %x)" col col.A col.R col.G col.B
```

```
1 The color Color [A=255, R=127, G=255, B=212] is (ff, 7f, ff, d4)
2 The color Color [Red] is (ff, ff, 0, 0)
3 The color Color [Green] is (ff, 0, 80, 0)
4 The color Color [Blue] is (ff, 0, 0, ff)
5 The color Color [Black] is (ff, 0, 0, 0)
6 The color Color [Gray] is (ff, 80, 80, 80)
7 The color Color [White] is (ff, ff, ff, ff)
```

The `System.Drawing.Size` is a general structure for specifying sizes as height and width pair of integers.

WinForms supports drawing of simple graphics primitives. Simple examples are `System.Drawing.Pen` to specify the color to be drawn, `System.Drawing.Point` to specify a pair of coordinates, and `System.Drawing.Graphics.DrawLine`. `DrawLine` is different than the previous examples since it must be related to a specific device, and it is typically accessed as an event. Displaying graphics in WinForms is performed as the reaction to an event. E.g., windows are created by the program, moved, minimized, occluded by other windows, resized, etc., by the user or the program, and each action may require that the content of the window is refreshed. Thus, we must create a function that WinForms can call, when it determines that the content needs to be redrawn. This is known as a *call-back function*, and it is added to an existing form using the `Paint.Add` function. As an example, consider the problem of draw a triangle in a window. For this we need to make a function that can draw a triangle not once, but any time WinForms determines it necessary to draw and redraw the triangle. Drawing is done with reference to a coordinate system. WinForms operates with several coordinate systems, the most important is the *screen coordinates*. Screen coordinate (x, y) have their origin in the top-left corner, and x increases to the right, while y increases down.¹ Thus, we may draw a triangle as demonstrated in Listing 13.4.

· call-back
function

· screen
coordinates

¹Todo: Possibly something about client coordinates and world coordinates.

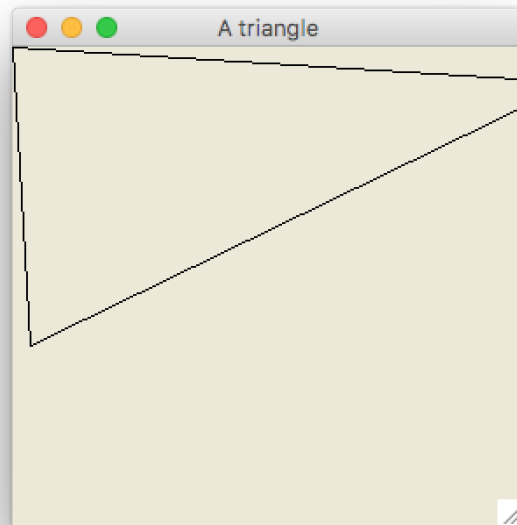


Figure 13.4: Drawing a triangle using Listing 13.4.

**Listing 13.4, winforms/triangle.fsx:
Adding line graphics to a window.**

```
1 // Choose some points and a color
2 let Points =
3     [|System.Drawing.Point (0,0);
4      System.Drawing.Point (10,170);
5      System.Drawing.Point (320,20);
6      System.Drawing.Point (0,0)|]
7 let penColor = System.Drawing.Color.Black
8 // Create window and setup drawing function
9 let pen = new System.Drawing.Pen (penColor)
10 let win = new System.Windows.Forms.Form ()
11 win.Text <- "A triangle"
12 win.Paint.Add (fun e -> e.Graphics.DrawLine (pen, Points))
13 // Start the event-loop.
14 System.Windows.Forms.Application.Run win
```

A walk-through of the code is as follows: First we create an array of points and a pen color, then we create a pen and a window. The method for drawing the triangle is added as an anonymous function using the created window's `Paint.Add` method. This function is to be called as a response to a paint event, and takes a `PaintEventArgs` object, which includes the `System.Drawing.Graphics` object. Since this object will be related to a specific device, when `Paint` is called then we may call the `DrawLine` function to sequentially draw lines between our array of points. Finally, we hand the form to the event-loop, which as one of the earliest events will open the window and call the `Paint` function we have associated with the form.

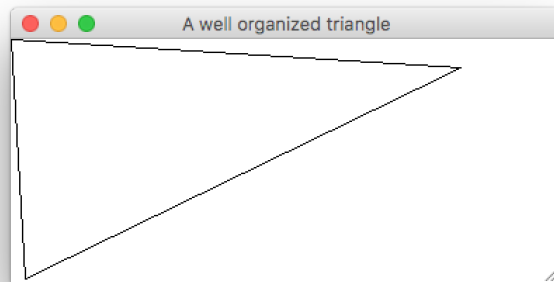


Figure 13.5: Better organization of the code for drawing a triangle, see Listing 13.5.

Listing 13.5, winforms/triangleOrganized.fsx:
Improved organization of code for drawing a triangle. Compare with Listing 13.4.

```

1 open System.Windows.Forms
2 open System.Drawing
3
4 type coordinates = (float * float) list
5 type pen = Color * float
6
7 /// Create a form and add a paint function
8 let createForm backgroundColor (width, height) title draw =
9     let win = new Form ()
10    win.Text <- title
11    win.BackColor <- backgroundColor
12    win.ClientSize <- Size (width, height)
13    win.Paint.Add draw
14    win
15
16 /// Draw a polygon with a specific color
17 let drawPoints (coords : coordinates) (pen : pen) (e : PaintEventArgs) =
18     let pairToPoint (x : float, y : float) =
19         Point (int (round x), int (round y))
20     let color, width = pen
21     let Pen = new Pen (color, single width)
22     let Points = Array.map pairToPoint (List.toArray coords)
23     e.Graphics.DrawLines (Pen, Points)
24
25 // Setup drawing details
26 let title = "A well organized triangle"
27 let backgroundColor = Color.White
28 let size = (400, 200)
29 let coords = [(0.0, 0.0); (10.0, 170.0); (320.0, 20.0); (0.0, 0.0)]
30 let pen = (Color.Black, 1.0)
31
32 // Create form and start the event-loop.
33 let win = createForm backgroundColor size title (drawPoints coords pen)
34 Application.Run win

```

Listing 13.6, winforms/transformWindows.fsx:
Reusable code for drawing in windows.

```

1 open System.Windows.Forms
2 open System.Drawing
3
4 type coordinates = (float * float) list
5 type pen = Color * float
6 type polygon = coordinates * pen
7
8 /// Create a form and add a paint function
9 let createForm backgroundColor (width, height) title draw =
10     let win = new Form ()
11     win.Text <- title
12     win.BackColor <- backgroundColor
13     win.ClientSize <- Size (width, height)
14     win.Paint.Add draw
15     win
16
17 /// Draw a polygon with a specific color
18 let drawPoints (polygLst : polygon list) (e : PaintEventArgs) =
19     let pairToPoint (x : float, y : float) =
20         Point (int (round x), int (round y))
21
22     for polyg in polygLst do
23         let coords, (color, width) = polyg
24         let pen = new Pen (color, single width)
25         let Points = Array.map pairToPoint (List.toArray coords)
26         e.Graphics.DrawLines (pen, Points)
27
28 /// Translate a point
29 let translatePoint (dx, dy) (x, y) =
30     (x + dx, y + dy)
31
32 /// Translate point array
33 let translatePoints (dx, dy) arr =
34     List.map (translatePoint (dx, dy)) arr
35
36 /// Rotate a point
37 let rotatePoint theta (x, y) =
38     (x * cos theta - y * sin theta, x * sin theta + y * cos theta)
39
40 /// Rotate point array
41 let rotatePoints theta arr =
42     List.map (rotatePoint theta) arr

```

²Todo: requires the introduction of type declarations.

³Todo: Remember to talk about pen width.

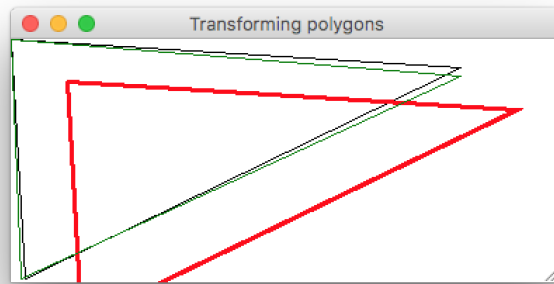


Figure 13.6: Transformed versions of the same triangle resulting from running the code in Listing 13.7.

Listing 13.7, winforms/transformWindows.fsx:

Code for drawing triangles using the reusable part shown in Listing 13.7.

```

44 // Setup drawing details
45 let title = "Transforming polygons"
46 let backgroundColor = Color.White
47 let size = (400, 200)
48 let points = [(0.0, 0.0); (10.0, 170.0); (320.0, 20.0); (0.0, 0.0)]
49 let polygLst =
50     [(points, (Color.Black, 1.0));
51      (translatePoints (40.0, 30.0) points, (Color.Red, 2.0));
52      (rotatePoints (1.0 * System.Math.PI / 180.0) points, (Color.Green, 1.0))
53     ]
54 // Create form and start the event-loop.
55 let win = createForm backgroundColor size title (drawPoints polygLst)
56 System.Windows.Forms.Application.Run win

```

Problem 13.1:

Given a triangle produce a Mandela drawing, where n rotated versions of the triangle is drawn around its center of mass.

Listing 13.8, winforms/rotationalSymmetry.fsx:
Create the window and changing its properties.

```
44 /// Calculate the mass center of a list of points
45 let centerOfPoints (points : (float * float) list) =
46     let addToAccumulator acc elm = (fst acc + fst elm, snd acc + snd elm)
47     let sum = List.fold addToAccumulator (0.0, 0.0) points
48     (fst sum / (float points.Length), snd sum / (float points.Length))
49
50 /// Generate repeated rotated point-color pairs
51 let rec rotatedLst points color width src dest nth n =
52     if n > 0 then
53         let newPoints =
54             points
55             |> translatePoints (- fst src, - snd src)
56             |> rotatePoints ((float n) * nth)
57             |> translatePoints dest
58         (newPoints, (color, width))
59         :: (rotatedLst points color width src dest nth (n - 1))
60     else
61         []
62
63 // Setup drawing details
64 let title = "Rotational Symmetry"
65 let backgroundColor = Color.White
66 let size = (600, 600)
67 let points = [(0.0, 0.0); (10.0, 170.0); (320.0, 20.0); (0.0, 0.0)]
68 let src = centerOfPoints points
69 let dest = ((float (fst size)) / 2.0, (float (snd size)) / 2.0)
70 let n = 36;
71 let nth = (360.0 / (float n)) * (System.Math.PI / 180.0)
72 let orgPoints =
73     points
74     |> translatePoints (fst dest - fst src, snd dest - snd src)
75 let polygLst =
76     rotatedLst points Color.Blue 1.0 src dest nth n
77     @ [(orgPoints, (Color.Red, 3.0))]
78
79 // Create form and start the event-loop.
80 let win = createForm backgroundColor size title (drawPoints polygLst)
81 Application.Run win
```

4

⁴Todo: Add other things to draw: filled stuff, clearing, circles, text

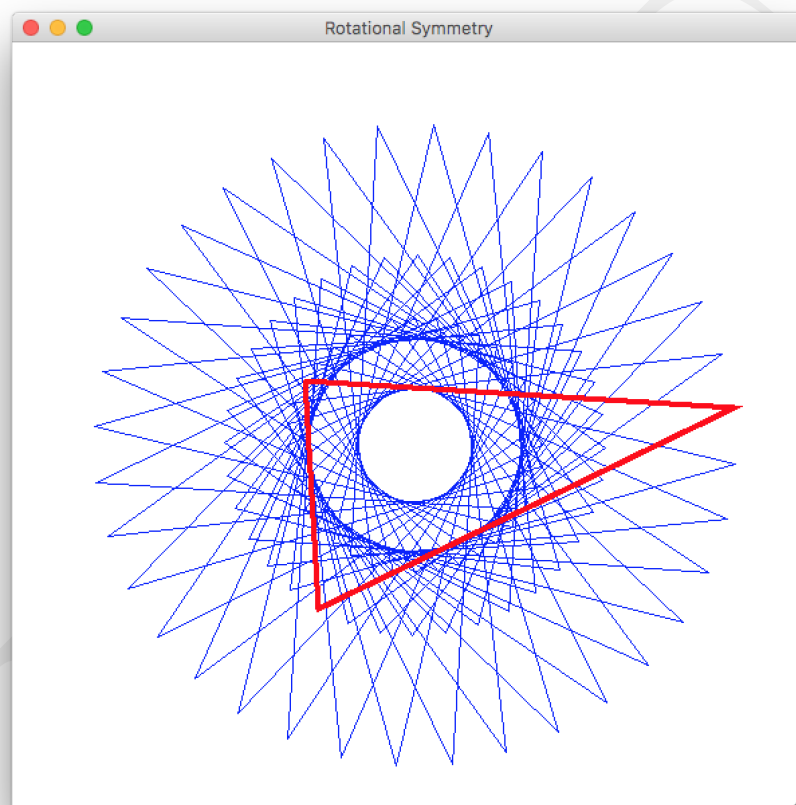


Figure 13.7: A symmetric figure resulting from Listing 13.8.

Function	Description
<code>DrawArc : Pen * Rectangle * Single * Single</code>	Draws an arc representing a portion of an ellipse specified by a <code>Rectangle</code> structure.
<code>DrawBezier : Pen * Point * Point * Point * Point</code>	Draws a Bézier spline defined by four <code>Point</code> structures.
<code>DrawClosedCurve : Pen * Point[]</code>	Draws a closed cardinal spline defined by an array of <code>Point</code> structures.
<code>DrawCurve : Pen * Point[]</code>	Draws a cardinal spline through a specified array of <code>Point</code> structures.
<code>DrawEllipse : Pen * Rectangle</code>	Draws an ellipse specified by a bounding <code>Rectangle</code> structure.
<code>DrawImage : Image * Point[]</code>	Draws the specified <code>Image</code> at the specified location and with the specified shape and size.
<code>DrawLines : Pen * Point[]</code>	Draws a series of line segments that connect an array of <code>Point</code> structures.
<code>DrawPie : Pen * Rectangle * Single * Single</code>	Draws a pie shape defined by an ellipse specified by a <code>Rectangle</code> structure and two radial lines.
<code>DrawPolygon : Pen * Point[]</code>	Draws a polygon defined by an array of <code>Point</code> structures.
<code>DrawRectangles : Pen * Rectangle[]</code>	Draws a series of rectangles specified by <code>Rectangle</code> structures.
<code>DrawString : String * Font * Brush * PointF</code>	Draws the specified text string at the specified location with the specified <code>Brush</code> and <code>Font</code> objects.
<code>FillClosedCurve : Brush * Point[]</code>	Fills the interior of a closed cardinal spline curve defined by an array of <code>Point</code> structures.
<code>FillEllipse : Brush * Rectangle</code>	Fills the interior of an ellipse defined by a bounding rectangle specified by a <code>Rectangle</code> structure.
<code>FillPie : Brush * Rectangle * Single * Single</code>	Fills the interior of a pie section defined by an ellipse specified by a <code>RectangleF</code> structure and two radial lines.
<code>FillPolygon : Brush * Point[]</code>	Fills the interior of a polygon defined by an array of points specified by <code>Point</code> structures.
<code>FillRectangle : Brush * Rectangle</code>	Fills the interior of a rectangle specified by a <code>Rectangle</code> structure.
<code>FillRegion : Brush * Region</code>	Fills the interior of a <code>Region</code> .

Table 13.1: Some methods of the `System.IO.Path` class.

13.2 Programming intermezzo

Problem 13.2:

Consider a curve consisting of piecewise straight lines all with the same length but with varying angles 0° , 90° , 180° , or 270° w.r.t. the horizontal axis. To draw this curve we need 3 basic operations: Draw (F), turn right ($+$), and turn left ($-$). The turning is w.r.t. the present direction. A Hilbert Curve is a spacefilling curve, which be expressed recursively as:

$$A \rightarrow -BF + AFA + FB- \quad (13.1)$$

$$B \rightarrow +AF - BFB - FA+ \quad (13.2)$$

starting with A . The order of the curve is the depth of the recursion, and to draw a 0'th order curve, we don't recurse at all, i.e., ignore all occurrences of the symbols A and B on the right-hand-side of (13.1), and get $-F + F + F-$. For the 1'st order curve, we recurse once, i.e.,

$$\begin{aligned} A &\rightarrow -BF + AFA + FB- \\ &\rightarrow -(+AF - BFB - FA+)F \\ &\quad + (-BF + AFA + FB-)F(-BF + AFA + FB-) \\ &\quad + F(+AF - BFB - FA+)- \\ &\rightarrow AF - BFB - FA + FBF + AFA + FB - F - BF + AFA + FBF + AF - BFB - FA \\ &\rightarrow F - F - F + FF + F + F - F - F + F + FF + F - F - F \end{aligned}$$

Make a program, that given an order produces an image of the Hilbert curve.

Listing 13.9, winforms/hilbert.fsx:
Create the window and changing its properties.

```
44 /// Turn 90 degrees left
45 let turnLeft (l, dir, c) = (l, dir + 3.141592/2.0, c)
46
47 /// Turn 90 degrees right
48 let turnRight (l, dir, c) = (l, dir - 3.141592/2.0, c)
49
50 /// Add a line to the curve of present direction
51 let draw (l, dir, (c : coordinates)) =
52     let nextPoint = rotatePoint dir (l, 0.0)
53     (l, dir, c @ [translatePoint c.[c.Length-1] nextPoint])
54
55 /// Find the maximum value of each coordinate element in a list
56 let maximum c =
57     let maxPoint (p1 : float*float) (p2 : float*float) =
58         (max (fst p1) (fst p2), max (snd p1) (snd p2))
59     List.fold maxPoint (-infinity, -infinity) c
60
61 /// Hilbert recursion production rules
62 let rec hilbertA n (l, dir, c) =
63     if n > 0 then
64         ((l, dir, c) |> turnLeft |> hilbertB (n-1) |> draw |> turnRight |>
65          hilbertA (n-1) |> draw |> hilbertA (n-1) |> turnRight |> draw |>
66          hilbertB (n-1) |> turnLeft)
67     else
68         (l, dir, c)
69 and hilbertB n (l, dir, c) =
70     if n > 0 then
71         ((l, dir, c) |> turnRight |> hilbertA (n-1) |> draw |> turnLeft |>
72          hilbertB (n-1) |> draw |> hilbertB (n-1) |> turnLeft |> draw |>
73          hilbertA (n-1) |> turnRight)
74     else
75         (l, dir, c)
76
77 // Calculate curve
78 let order = 5
79 let l = 20.0
80 let (_, dir, C) = hilbertA order (l, 0.0, [(0.0, 0.0)])
81
82 // Setup drawing details
83 let title = "Hilbert's curve"
84 let backgroundColor = Color.White
85 let cMax = maximum C
86 let size = (int (fst cMax)+1, int (snd cMax)+1)
87 let polygLst = [(C, (Color.Black, 3.0))]
88
89 // Create form and start the event-loop.
90 let win = createForm backgroundColor size title (drawPoints polygLst)
91 System.Windows.Forms.Application.Run win
```

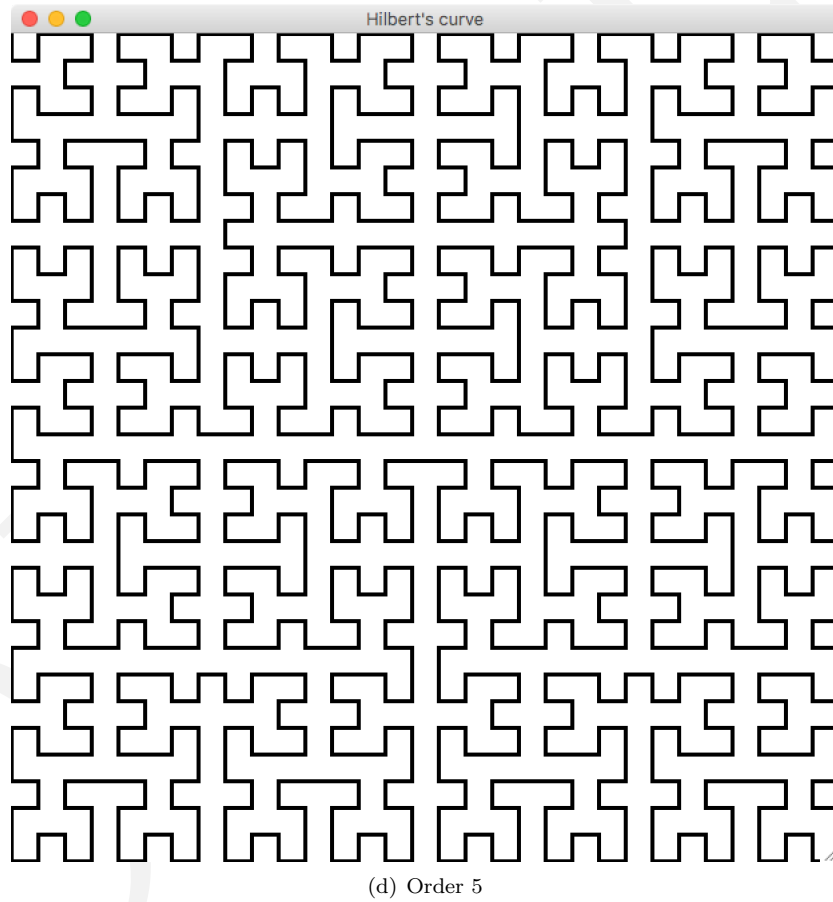
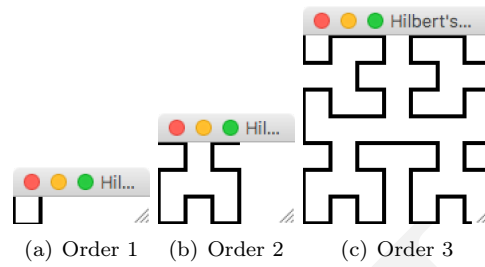


Figure 13.8: Hilbert curves of order 1, 2, 3, and 5 by code in Listing 13.9.

Listing 13.10, winforms/windowEvents.fsx:
Catching window, mouse, and keyboard events..

```
1 open System.Windows.Forms
2 open System.Drawing
3
4 type coordinates = (float * float) list
5 type pen = Color * float
6 type polygon = coordinates * pen
7
8 /// Create a form and add a paint function
9 let createForm backgroundColor (width, height) title draw =
10     let win = new Form ()
11     win.Text <- title
12     win.BackColor <- backgroundColor
13     win.ClientSize <- Size (width, height)
14     // Paint event
15     win.Paint.Add draw
16     // Window event
17     win.Move.Add (fun e -> printfn "Move: %A" win.Location)
18     win.Resize.Add (fun _ -> printfn "Resize: %A" win.DisplayRectangle)
19     // Mouse event
20     let mutable record = false;
21     win.MouseMove.Add (fun e -> if record then printfn "MouseMove: %A" e.
22         Location)
23     win.MouseDown.Add (fun e -> printfn "MouseDown: %A" e.Location; (record
24         <- true))
25     win.MouseUp.Add (fun e -> printfn "MouseUp: %A" e.Location; (record <-
26         false))
27     win.MouseClick.Add (fun e -> printfn "MouseClick: %A" e.Location)
28     // Keyboard event
29     win.KeyPreview <- true
30     win.KeyPress.Add (fun e -> printfn "KeyPress: %A" (e.KeyChar.ToString ()
31         ))
32     win
33
34 /// Draw a polygon with a specific color
35 let drawPoints (polygLst : polygon list) (e : PaintEventArgs) =
36     let pairToPoint (x : float, y : float) =
37         Point (int (round x), int (round y))
38
39     for polyg in polygLst do
40         let coords, (color, width) = polyg
41         let pen = new Pen (color, single width)
42         let Points = Array.map pairToPoint (List.toArray coords)
43         e.Graphics.DrawLines (pen, Points)
44
45 let backgroundColor = System.Drawing.Color.White
46 let title = "Window events"
47 let size = (200, 200)
48 let polygLst = []
49
50 // Create form and start the event-loop.
51 let win = createForm backgroundColor size title (drawPoints polygLst)
52 System.Windows.Forms.Application.Run win
```

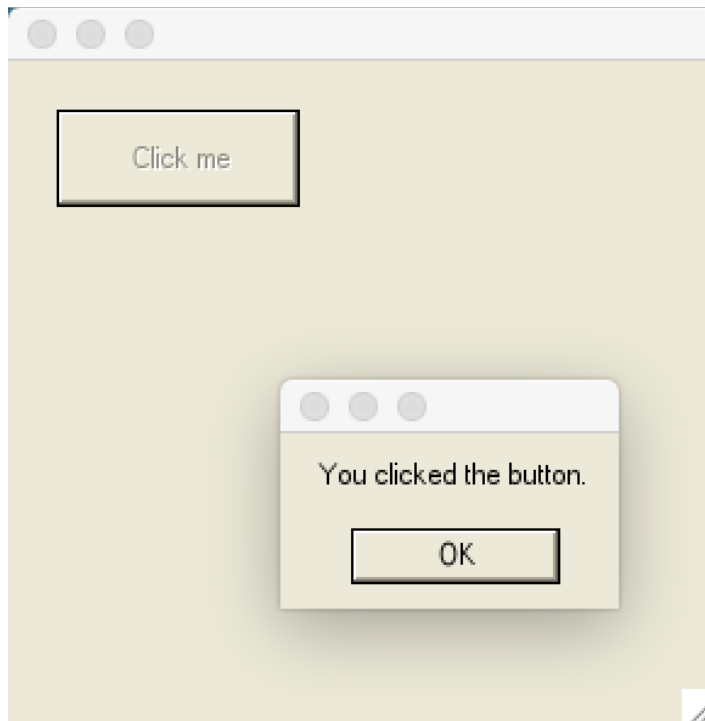


Figure 13.9: A button is pressed and the event handler calls the `MessageBox.Show` dialogue window by the code in Listing 13.11.

13.3 Buttons and stuff

Listing 13.11, `winforms/buttonControl.fsx`:
Create the button and an event.

```
1 /// A button event
2 let buttonClicked (e : System.EventArgs) =
3     ignore (System.Windows.Forms.MessageBox.Show "You clicked the button.")
4
5 // Create a button
6 let button = new System.Windows.Forms.Button ()
7 button.Size <- new System.Drawing.Size (100, 40)
8 button.Location <- new System.Drawing.Point (20, 20)
9 button.Text <- "Click me"
10 button.Click.Add buttonClicked
11
12 // Create a window and add button
13 let win = new System.Windows.Forms.Form ()
14 win.Controls.Add button
15
16 // Start the event-loop.
17 System.Windows.Forms.Application.Run win
```

Listing 13.12, winforms/panel.fsx:
Create a panel, label, text input controls.

```
1 open System
2 open System.Drawing
3 open System.Windows.Forms
4
5 // Initialize a form containing a panel, textbox, and a label
6 let form = new Form ()
7 let panel = new Panel();
8 let textBox = new TextBox();
9 let label = new Label();
10
11 // Customize the Form.
12 form.Text <- "A panel";
13 form.ClientSize <- new Size(400, 300);
14
15 // Customize the Panel control.
16 panel.Location <- new Point(56,72);
17 panel.Size <- new Size(264, 152);
18 panel.BorderStyle <- BorderStyle.Fixed3D;
19
20 // Customize the Label and TextBox controls.
21 label.Location <- new Point(16,16);
22 label.Text <- "label1";
23 label.Size <- new Size(104, 16);
24 textBox.Location <- new Point(16,32);
25 textBox.Text <- "Initial text";
26 textBox.Size <- new Size(152, 20);
27
28 // Add panel to form and label and textBox to panel.
29 form.Controls.Add(panel);
30 panel.Controls.Add(label);
31 panel.Controls.Add(textBox);
32
33 // Give control to WinForms' event loop
34 Application.Run form
```

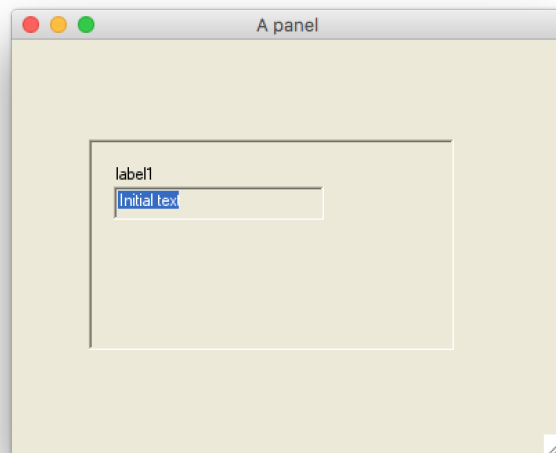


Figure 13.10: A panel including a label and a text input field, see Listing 13.12.

Listing 13.13, winforms/flowLayoutPanel.fsx:
Create a flowLayoutPanel, with checkbox and radiobuttons.

```

1 open System
2 open System.Windows.Forms
3
4 let flowLayoutPanel = new System.Windows.Forms.FlowLayoutPanel ();
5 let button1 = new System.Windows.Forms.Button ();
6 let button2 = new System.Windows.Forms.Button ();
7 let button3 = new System.Windows.Forms.Button ();
8 let button4 = new System.Windows.Forms.Button ();
9 let wrapContentsCheckBox = new System.Windows.Forms.CheckBox ();
10 let flowTopDownBtn = new System.Windows.Forms.RadioButton ();
11 let flowBottomUpBtn = new System.Windows.Forms.RadioButton ();
12 let flowLeftToRight = new System.Windows.Forms.RadioButton ();
13 let flowRightToLeftBtn = new System.Windows.Forms.RadioButton ();
14
15 //
16 // button1
17 //
18 button1.Location <- new System.Drawing.Point (3, 3);
19 button1.Name <- "button1";
20 button1.TabIndex <- 0;
21 button1.Text <- "button1";
22 //
23 // button2
24 //
25 button2.Location <- new System.Drawing.Point (84, 3);
26 button2.Name <- "button2";
27 button2.TabIndex <- 1;
28 button2.Text <- "button2";
29 //
30 // button3
31 //
32 button3.Location <- new System.Drawing.Point (3, 32);
33 button3.Name <- "button3";
34 button3.TabIndex <- 2;
35 button3.Text <- "button3";
36 //
37 // button4
38 //

```

Listing 13.14, winforms/flowLayoutPanel.fsx:
Create a flowLayoutPanel, with checkbox and radiobuttons.

```
43 //
44 // wrapContentsCheckBox
45 //
46 wrapContentsCheckBox.Location <- new System.Drawing.Point (46, 162);
47 wrapContentsCheckBox.Name <- "wrapContentsCheckBox";
48 wrapContentsCheckBox.TabIndex <- 1;
49 wrapContentsCheckBox.Text <- "Wrap Contents";
50 wrapContentsCheckBox.Checked <- true
51 wrapContentsCheckBox.CheckedChanged.Add (fun _ -> flowLayoutPanel.
    WrapContents <- wrapContentsCheckBox.Checked)
52 //
53 // flowTopDownBtn
54 //
55 flowTopDownBtn.Location <- new System.Drawing.Point (45, 193);
56 flowTopDownBtn.Name <- "flowTopDownBtn";
57 flowTopDownBtn.TabIndex <- 2;
58 flowTopDownBtn.Text <- "Flow TopDown";
59 flowTopDownBtn.Checked <- flowLayoutPanel.FlowDirection = FlowDirection.
    TopDown;
60 flowTopDownBtn.CheckedChanged.Add (fun _ -> flowLayoutPanel.FlowDirection
    <- FlowDirection.TopDown);
61 //
62 // flowBottomUpBtn
63 //
64 flowBottomUpBtn.Location <- new System.Drawing.Point (44, 224);
65 flowBottomUpBtn.Name <- "flowBottomUpBtn";
66 flowBottomUpBtn.TabIndex <- 3;
67 flowBottomUpBtn.Text <- "Flow BottomUp";
68 flowBottomUpBtn.Checked <- flowLayoutPanel.FlowDirection = FlowDirection.
    BottomUp
69 flowBottomUpBtn.CheckedChanged.Add (fun _ -> flowLayoutPanel.FlowDirection
    <- FlowDirection.BottomUp);
70 //
71 // flowLeftToRight
72 //
73 flowLeftToRight.Location <- new System.Drawing.Point (156, 193);
74 flowLeftToRight.Name <- "flowLeftToRight";
75 flowLeftToRight.TabIndex <- 4;
76 flowLeftToRight.Text <- "Flow LeftToRight";
77 flowLeftToRight.Checked <- flowLayoutPanel.FlowDirection = FlowDirection.
    LeftToRight;
78 flowLeftToRight.CheckedChanged.Add (fun _ -> flowLayoutPanel.FlowDirection
    <- FlowDirection.LeftToRight);
79 //
80 // flowRightToLeftBtn
81 //
82 flowRightToLeftBtn.Location <- new System.Drawing.Point (155, 224);
83 flowRightToLeftBtn.Name <- "flowRightToLeftBtn";
84 flowRightToLeftBtn.TabIndex <- 5;
85 flowRightToLeftBtn.Text <- "Flow RightToLeft";
86 flowRightToLeftBtn.Checked <- flowLayoutPanel.FlowDirection =
    FlowDirection.RightToLeft;
87 flowRightToLeftBtn.CheckedChanged.Add (fun _ -> flowLayoutPanel.
    FlowDirection <- FlowDirection.RightToLeft);
```


Listing 13.15, winforms/flowLayoutPanel.fsx:
Create a flowLayoutPanel, with checkbox and radiobuttons.

```
88 //
89 // flowLayoutPanel
90 //
91 flowLayoutPanel.Controls.Add (button1);
92 flowLayoutPanel.Controls.Add (button2);
93 flowLayoutPanel.Controls.Add (button3);
94 flowLayoutPanel.Controls.Add (button4);
95 flowLayoutPanel.Location <- new System.Drawing.Point (47, 55);
96 flowLayoutPanel.BorderStyle <- BorderStyle.Fixed3D;
97 flowLayoutPanel.Name <- "flowLayoutPanel";
98 flowLayoutPanel.TabIndex <- 0;
99 flowLayoutPanel.WrapContents <- wrapContentsCheckBox.Checked
00 //
01 // Form1
02 //
03 let Form1 = new Form ()
04 Form1.ClientSize <- new System.Drawing.Size (292, 266);
05 Form1.Controls.Add (flowRightToLeftBtn);
06 Form1.Controls.Add (flowLeftToRight);
07 Form1.Controls.Add (flowBottomUpBtn);
08 Form1.Controls.Add (flowTopDownBtn);
09 Form1.Controls.Add (wrapContentsCheckBox);
10 Form1.Controls.Add (flowLayoutPanel);
11 Form1.Name <- "Form1";
12 Form1.Text <- "Form1";
13 Application.Run Form1
```

5

...

⁵Todo: Click.Add expects a function System.EventArgs -> unit therefore the ignore function.

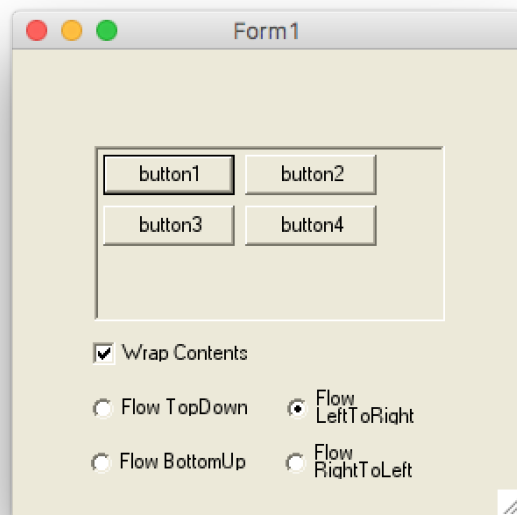


Figure 13.11: Demonstration of the `FlowLayoutPanel` panel, `CheckBox`, and `RadioButton` controls, see Listing 13.15.

Function	Description
<code>DataGridView</code>	Display data on a table.
<code>TextBox</code>	Display editable text.
<code>Label</code>	Display text.
<code>LinkLabel</code>	Display clickable text.
<code>ProgressBar</code>	Display the current progress as a bar.
<code>WebBrowser</code>	Enable navigation of the web.
<code>CheckedListBox</code>	Display a scrollable check box list.
<code>ComboBox</code>	Display a drop-down list.
<code>ListBox</code>	Display a list of text and icons.
<code>PictureBox</code>	Display a bitmap image
<code>CheckBox</code>	Display a checkbox and a label of text.
<code>RadioButton</code>	Display an on-off radio button
<code>TrackBar</code>	Enable the user to input value by moving a cursor on a slider bar
<code>DateTimePicker</code>	Enable the user to select a date from a graphical calendar
<code>ColorDialog</code>	Enable the user to pick a color
<code>FontDialog</code>	Enable the user to pick a font and its attributes
<code>OpenFileDialog</code>	Enable the user to navigate the file system and select a file..
<code>PrintDialog</code>	Enable the user to select a printer and its attributes.
<code>SaveDialog</code>	Enable the user to navigate the file system and specify a filename.
<code>MenuStrip</code>	Allow the user to choose from a custom menu
<code>Button</code>	Display a clickable button with text
<code>Tooltip</code>	Briefly display a pop-up window, when the user rests the pointer on the control
<code>SoundPlayer</code>	Play sounds in the .wav format.

Table 13.2: Some controls available in WinForms.

Function	Description
<code>Panel</code>	Groups a set of controls in a scrollable frame.
<code>GroupBox</code>	Group a set of controls in a non-scrollable frame.
<code>TabControl</code>	Group controls in tabpages, A tabpage is selected by clicking on its tab.
<code>SplitContainer</code>	Group controls into two resizable panels.
<code>TableLayoutPanel</code>	Group controls into a grid.
<code>FlowLayoutPanel</code>	Group controls into a set of flowable panels. The panels may flow horizontally or vertically as a response to window resizing.

Table 13.3: Some controls for grouping other controls.

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

. [], 31
ReadKey, 114
ReadLine, 114
Read, 114
System.Console.ReadKey, 114
System.Console.ReadLine, 114
System.Console.Read, 114
System.Console.WriteLine, 114
System.Console.Write, 114
System.Drawing.Color, 129
WriteLine, 114
Write, 114
abs, 186
acos, 186
asin, 186
atan2, 186
atan, 186
bignum, 23
bool, 19
byte[], 23
byte, 23
ceil, 186
char, 19
cosh, 186
cos, 186
decimal, 23
double, 23
eprintfn, 51
eprintf, 51
exn, 19
exp, 186
failwithf, 51
float32, 23
float, 19
floor, 186
fprintfn, 51
fprintf, 51
ignore, 51
int16, 23
int32, 23
int64, 23
int8, 23
int, 19
it, 19
log10, 186
log, 186
max, 186
min, 186
nativeint, 23
obj, 19
pown, 186
printfn, 51
printf, 48, 51
round, 186
sbyte, 23
sign, 186
single, 23
sinh, 186
sin, 186
sprintf, 51
sqrt, 186
stderr, 51, 114
stdin, 114
stdout, 51, 114
string, 19
tanh, 186
tan, 186
uint16, 23
uint32, 23
uint64, 23
uint8, 23
unativeint, 23
unit, 19

accessors, 129
aliasing, 55
American Standard Code for Information Inter-
change, 169
and, 26
anonymous function, 45
array sequence expressions, 149
Array.toList, 85
ASCII, 169
ASCIIbetical order, 31, 170

base, 19, 168
Basic Latin block, 170
Basic Multilingual plane, 170
basic types, 19
binary, 168
binary number, 21

- binary operator, 25
- binary64, 168
- binding, 14
- bit, 21, 168
- black-box testing, 90
- block, 40
- blocks, 170
- boolean and, 187
- boolean or, 187
- branches, 67
- branching coverage, 92
- bug, 89
- byte, 168

- call-back function, 130
- character, 21
- class, 24, 32
- CLI, 126
- code point, 21, 170
- command-line interface, 126
- compiled, 11
- computation expressions, 79, 84
- conditions, 67
- Cons, 81
- console, 11
- coverage, 92
- currying, 46

- debugging, 13, 90, 98
- decimal number, 19, 168
- decimal point, 20, 168
- Declarative programming, 8
- digit, 20, 168
- dot notation, 32
- double, 168
- downcasting, 24

- EBNF, 20, 174
- efficiency, 90
- encapsulate code, 42
- encapsulation, 46, 53
- environment, 99
- event driven programs, 126
- event-loop, 128
- exception, 28
- exclusive or, 29
- executable file, 11
- expression, 14, 24
- expressions, 8
- Extended Backus-Naur Form, 20, 174
- Extensible Markup Language, 57

- file, 113
- floating point number, 20
- flushing, 117

- format string, 14
- fractional part, 20, 24
- function, 17
- function coverage, 92
- Functional programming, 8, 143
- functional programming, 8
- functionality, 89
- functions, 8

- generic function, 44
- graphical user interface, 126
- GUI, 126

- hand tracing, 98
- Head, 81
- hexadecimal, 168
- hexadecimal number, 21
- HTML, 60
- Hyper Text Markup Language, 60

- IEEE 754 double precision floating-point format, 168
- Imperativ programming, 142
- Imperative programming, 8
- implementation file, 11
- infix notation, 25
- infix operator, 24
- integer, 20
- integer division, 28
- interactive, 11
- IsEmpty, 81
- Item, 81

- jagged arrays, 86

- keyword, 14

- Latin-1 Supplement block, 170
- Latin1, 170
- least significant bit, 168
- Length, 81
- length, 76
- lexeme, 17
- lexical scope, 16, 44
- lexically, 38
- lightweight syntax, 35, 38
- list, 79
- list sequence expression, 149
- List.Empty, 81
- List.toArray, 81
- List.toList, 81
- literal, 19
- literal type, 23

- machine code, 142
- maintainability, 90

- member, 24, 76
- method, 32
- methods, 128
- mockup code, 98
- module elements, 162
- modules, 11
- most significant bit, 168
- Mutable data, 51
- mutually recursive, 70

- namespace, 24
- namespace pollution, 157
- NaN, 168
- nested scope, 40
- newline, 22
- not, 26
- not a number, 168

- obfuscation, 79
- object, 32
- Object oriented programming, 142
- Object-oriented programming, 8
- objects, 8
- octal, 168
- octal number, 21
- operand, 43
- operands, 25
- operator, 25, 43
- option type, 111
- or, 26
- overflow, 27

- pattern matching, 151, 158
- portability, 90
- precedence, 25
- prefix operator, 25
- Procedural programming, 142
- procedure, 46
- production rules, 174
- properties, 128

- ragged multidimensional list, 84
- raise an exception, 106
- range expression, 80
- reals, 168
- recursive function, 68
- reference cells, 54
- reliability, 89
- remainder, 28
- rounding, 24
- run-time error, 29

- scientific notation, 20
- scope, 40
- screen coordinates, 130

- script file, 11
- script-fragment, 17
- script-fragments, 11
- Seq.initInfinite, 149
- Seq.item, 146
- Seq.take, 146
- Seq.toArray, 149
- Seq.toList, 149
- side-effect, 85
- side-effects, 46, 54
- signature file, 11
- slicing, 85
- software testing, 90
- state, 8
- statement, 14
- statement coverage, 92
- statements, 8, 142
- states, 142
- stopping criterium, 69
- stream, 114
- string, 14, 22
- Structured programming, 8
- subnormals, 168

- Tail, 81
- tail-recursive, 69
- terminal symbols, 174
- tracing, 98
- truth table, 26
- tuple, 76
- type, 15, 19
- type declaration, 15
- type inference, 13, 15
- type safety, 43
- typecasting, 23

- unary operator, 25
- underflow, 27
- Unicode, 21
- unicode general category, 170
- Unicode Standard, 170
- Uniform Resource Identifiers, 121
- Uniform Resource Locator, 120
- unit of measure, 154
- unit testing, 90
- unit-less, 155
- unit-testing, 13
- upcasting, 24
- URI, 121
- URL, 120
- usability, 90
- UTF-16, 172
- UTF-8, 172

- variable, 51

verbatim, 23

white-box testing, 90, 92

whitespace, 22

whole part, 20, 24

wild card, 38

WinForms, 126

word, 168

XML, 57

xor, 29

yield bang, 146