1 Pattern Matching

Pattern matching is used to transform values and variables into a syntactical structure. The simplest example is value-bindings. The *let*-keyword was introduced in **??**, its extension with pattern matching is given as,

· let@let

Listing 1.1: Syntax for let-expressions with pattern matching.

```
[[<Literal>]]
let [mutable] <pat> [: <returnType>] = <bodyExpr> [in <expr>]
```

A typical use of this is to extract elements of tuples, as demonstrated in Listing 1.2.

Listing 1.2 letPattern.fsx:
Patterns in let expressions may be used to extract elements of tuples.

```
let a = (3,4)
let (x,y) = a
let (alsoX,_) = a
printfn "%A: %d %d %d" a x y alsoX
```

```
$ fsharpc --nologo letPattern.fsx && mono letPattern.exe
(3, 4): 3 4 3
```

Here we extract the elements of a pair twice. First by binding to `x` and `y`, and second by binding to `alsoX` while using the wildcard pattern to ignore the second element. Thus, again the wildcard pattern in value-bindings is used to underline a disregarded value.

Another common use of patterns is as an alternative to `if – then – else` expressions, particularly when parsing input for a function. Consider the example in Listing 1.3.

Listing 1.3 switch.fsx:
Using if – then – else to print discriminated unions.

```
type Medal = Gold | Silver | Bronze
let statement (m : Medal) : string =
  if m = Gold then "You won"
  elif m = Silver then "You almost won"
  else "Maybe you will win next time"

let m = Silver
printfn "%A : %s" m (statement m)
```

```
$ fsharpc --nologo switch.fsx && mono switch.exe
Silver : You almost won
```

In the example, a discriminated union and a function are defined. The function converts each case to a supporting statement, using an `if`-expression. The same can be done with the *match – with* expression and patterns, as demonstrated in Listing 1.4.

· match@match
· with@with

Listing 1.4 switchPattern.fsx:
Using match – with to print discriminated unions.

```
type Medal = Gold | Silver | Bronze
let statement (m : Medal) : string =
  match m with
    Gold -> "You won"
    | Silver -> "You almost won"
    | _ -> "Maybe you can win next time"

let m = Silver
printfn "%A : %s" m (statement m)
```

```
$ fsharpc --nologo switchPattern.fsx && mono switchPattern.exe
Silver : You almost won
```

Here we used a pattern for the discriminated union cases and a wildcard pattern as default. The lightweight syntax for `match`-expressions is,

Listing 1.5: Syntax for match-expressions.

```
match <inputExpr> with
 [| ]<pat> [when <guardExpr>] -> <caseExpr>
 | <pat> [when <guardExpr>] -> <caseExpr>
 | <pat> [when <guardExpr>] -> <caseExpr>
 ...
```

where `<inputExpr>` is the *input pattern* to find matches of, `<pat>` is a pattern to match with, `<guardExpr>` is an optional guard expression, and `<caseExpr>` is the resulting expression. Each set starting with `<pat>` is called a case. In lightweight syntax, the indentation must be equal to or higher than the indentation of `match`. All cases must return a value of the same type, and F# reports an error when the complete domain of the input pattern

· input pattern

is not covered by cases in `match`-expressions.

Patterns are also used in a version of *for*-loop expressions, and its lightweight syntax is given as, · for@for

Listing 1.6: Syntax for for-expressions with pattern matching.

```
for <pat> in <sourceExpr> do
  <bodyExpr>
```

Typically, `<sourceExpr>` is a list or an array. An example is given in Listing 1.7.

Listing 1.7 forPattern.fsx: Patterns may be used in for-loops.

```
for (_,y) in [(1,3); (2,1)] do
  printfn "%d" y
```

```
$ fsharpc --nologo forPattern.fsx && mono forPattern.exe
3
1
```

The wildcard pattern is used to disregard the first element in a pair while iterating over the complete list. It is good practice to **use wildcard patterns to emphasize unused values.** Advice

The final expression involving patterns to be discussed is the *anonymous functions*. Patterns for anonymous functions have the syntax, · anonymous functions

Listing 1.8: Syntax for anonymous functions with pattern matching.

```
fun <pat> [<pat> ...] -> <bodyExpr>
```

This is an extension of the syntax discussed in **??**. A typical use for patterns in *fun*-expressions is shown in Listing 1.9. · fun@fun

Listing 1.9 funPattern.fsx: Patterns may be used in fun-expressions.

```
let f = fun _ -> "hello"
printfn "%s" (f 3)
```

```
$ fsharpc --nologo funPattern.fsx && mono funPattern.exe
hello
```

Here we use an anonymous function expression and bind it to `f`. The expression has one argument of any type, which it ignores through the wildcard pattern. Some limitations apply to the patterns allowed in `fun`-expressions. The wildcard pattern in `fun`-expressions are often used for *mockup functions*, where the code requires the said function, but its · mockup functions