ME 2017-10-14

15 | Patterns Matching

Start with tuple patterns, wildcards and records.

Patterns are very often used when defining functions. Consider the example in Listing 15.1.

Listing 15.1 switch.fsx:
Using – – 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 can win next time"

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

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

Here we have defined a discriminated union and a function to convert each case to a supporting statement using an `if`-expression. The same can be done with the *match – with* expression and patterns. This is demonstrated in Listing 15.2.

- match
- with

Listing 15.2 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 called named pattern for the discriminated union cases and a wildcard pattern as default. The lightweight syntax for `match`-expressions is,

Listing 15.3 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# will complain about in match-expressions, if not the complete domain of the input pattern is covered by cases.

· input pattern

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

· for

Listing 15.4 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 15.5.

Listing 15.5 forPattern.fsx: Patterns may be used in -loops.

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

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

Here 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

· let

Value-bindings using the *let*-keyword may also use patterns, and the syntax is,

Listing 15.6 Syntax for `let`-expressions with pattern matching.

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

This is an extension of the syntax discussed in Section 6.1. A typical use of this is to extract elements of tuples as demonstrated in Listing 15.7.

Listing 15.7 letPattern.fsx:
Patterns may be used in -loops.

```
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.

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

Listing 15.8 Syntax for anonymous functions with pattern matching.

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

This is an extension of the syntax discussed in Section 6.2. A typical use for patterns in *fun*-expressions is shown in Listing 15.9. · fun

Listing 15.9 funPattern.fsx:
Patterns may be used in -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 using the wildcard pattern. Some limitations apply to the patterns allowed in `fun`-expressions.[1] The wildcard pattern in `fun`-expressions are often used for *mockup functions*, where the code requires the said function, but its content has yet to be decided. Thus, mockup functions can be used as loose place-holders, while experimenting with program design. · mockup functio

Patterns are also used in exceptions to be discussed in Section 18.1, and in conjunction with the `function`-keyword, an expression we discourage in this book. We will now demonstrate a list of important patterns in F#.

an expression is not a keyword

15.1 Wildcard pattern

A *wildcard pattern* is denoted "`_`" and matches anything, see e.g., Listing 15.10. · wildcard patter · _

[1] Jon: **Remove or elaborate.**

Listing 15.10 wildcardPattern.fsx:
Constant patterns matches to constants.

```
let whatEver (x : int) : string =
  match x with
    _ -> "If you say so"

printfn "%s" (whatEver 42)
```

```
$ fsharpc --nologo wildcardPattern.fsx && mono wildcardPattern.exe
If you say so
```

In this example, anything matches the wildcard pattern, so all cases are covered and the function always returns the same sentence. This is rarely a useful structure on its own, since this could be replaced by a value binding or by a function ignoring its input. However, wildcard patterns are extremely useful, since they act as the final `else` in `if`-expressions.

15.2 Constant and literal patterns

A *constant pattern* matches any input pattern with constants, see e.g., Listing 15.11. · constant patter

Listing 15.11 constPattern.fsx:
Constant patterns matches to constants.

```
type Medal = Gold | Silver | Bronze
let intToMedal (x : int) : Medal =
  match x with
    0 -> Gold
    | 1 -> Silver
    | _ -> Bronze

printfn "%A" (intToMedal 0)
```

```
$ fsharpc --nologo constPattern.fsx && mono constPattern.exe
Gold
```

In this example, the input pattern are queried for a match with `0` and `1` or the wildcard pattern. Any simple literal type constants may be used in the constant pattern such as `8`, `23y`, `1010u`, `1.2`, `"hello world"`, `'c'`, and , `false`. Here we also use the wildcard pattern. Notice, matching is performed in a lazy manner and stops for the first matching case from the top. Thus, although the wildcard pattern matches everything, its case expression is only executed if none of the previous patterns matches the input.

Constants can also be pre-bound by the `[<Literal>]` attribute for value-bindings. This is demonstrated in Listing 15.12.

What about
let 3 = 4
or let (x, 0) = (3, 1) ?

Listing 15.12 literalPattern.fsx:
A variant of constant patterns are literal patterns.

```
[<Literal>]
let TheAnswer = 42
let whatIsTheQuestion (x : int) : string =
  match x with
    TheAnswer -> "We will need to build a bigger machine..."
    | _ -> "Don't know that either"

printfn "%A" (whatIsTheQuestion 42)
```

```
$ fsharpc --nologo literalPattern.fsx && mono literalPattern.exe
"We will need to build a bigger machine..."
```

Here the attributed is used to identify the value-binding `TheAnswer` to be used as if it were a simple literal type. Literal patterns must be either uppercase or module prefixed identifiers.

15.3 Variable patterns

A *variable pattern* is a single lower-case letter identifier. Variable pattern identifiers are assigned the value and type of the input pattern. Combinations of constant and variable patters are also allowed together with records and arrays. This is demonstrated in Listing 15.13.

· variable pattern

Listing 15.13 variablePattern.fsx:
Variable patterns are useful for extracting and naming fields etc.

```
let (name, age) = ("Jon", 50)
let getAgeString (age : int) : string =
  match age with
    0 -> "newborn"
    | 1 -> "1 year old"
    | n -> (string n) + " years old"

printfn "%s is %s" name (getAgeString age)
```

```
$ fsharpc --nologo variablePattern.fsx && mono variablePattern.exe
Jon is 50 years old
```

In this example, the use of the value identifier `n` has the function of a named wildcard pattern. Hence, the case could as well have been `| _ -> (string age) + "years old"`, since `age` is already defined in this scope. However, variable patterns syntactically act as an argument to an anonymous function, and thus act to isolate the dependencies. They are also very useful together with guards, see Section 15.4.

15.4 Guards

A *guard* is a pattern used together with `match`-expressions including the *when*-keyword, as shown in Listing 15.3.

· guard
· when

Listing 15.14 guardPattern.fsx:
Guard expressions can be used with other patterns to restrict matches.

```
let getAgeString (age : int) : string =
  match age with
    n when n < 1 -> "infant"
    | n when n < 13 -> "child"
    | n when n < 20 -> "teen"
    | _ -> "adult"

printfn "A person aged %d is a/an %s" 50 (getAgeString 50)
```

```
$ fsharpc --nologo guardPattern.fsx && mono guardPattern.exe
A person aged 50 is a/an adult
```

Here guards are used to iteratively carve out subset of integers to assign different strings to each set. The guard expression in `<pat> when <guardExpr> -> <caseExpr>` is any expression evaluating to a Boolean, and the case expression is only executed for the matching case.

15.5 List patterns

Lists have a concatenation pattern associated with them. The *"::"* cons-operator is used to to match the head and the rest of a list, and *"[]"* is used to match an empty list, which is also sometimes called the nil-case. This is very useful, when recursively parsing lists as shown in Listing 15.15

- list pattern
- ::
- []

Listing 15.15 listPattern.fsx:
Recursively parsing a list using list patterns.

```
let rec sumList (lst : int list) : int =
  match lst with
    n :: rest -> n + (sumList rest)
    | [] -> 0

let rec sumThree (lst : int list) : int =
  match lst with
    [a; b; c] -> a + b + c
    | _ -> sumList lst

let aList = [1; 2; 3]
printfn "The sum of %A is %d, %d" aList (sumList aList) (sumThree aList)
```

```
$ fsharpc --nologo listPattern.fsx && mono listPattern.exe
The sum of [1; 2; 3] is 6, 6
```

In the example, the function `sumList` uses the cons operator to match the head of the list with `n` and the tail with `rest`. The pattern `n :: tail` also matches `3 :: []`, and in that case `tail` would be assigned the value `[]`. When `lst` is empty, then it matches with "`[]`". List patterns can also be matched explicitly named elements as demonstrated in the `sumThree` function. The elements to be matched can be any mix of constants and variables.

15.6 Array, record, and discriminated union patterns

Array, *record*, and *discriminated union patterns* are direct extensions on constant, variable, and wildcard patterns. In Listing 15.16 is given examples of array patterns.

- array pattern
- record pattern
- discriminated u patterns

Listing 15.16 arrayPattern.fsx:
Using variable patterns to match on size and content of arrays.

```
let arrayToString (x : int []) : string =
  match x with
    [|1;_;_|] -> "3 elements, first of is a one"
    | [|x;1;_|] -> "3 elements, first is " + (string x) + "Second is one"
    | x -> "A general array"

printfn "%s" (arrayToString [|1; 1; 1|])
printfn "%s" (arrayToString [|3; 1; 1|])
printfn "%s" (arrayToString [|1|])
```

```
$ fsharpc --nologo arrayPattern.fsx && mono arrayPattern.exe
3 elements, first of is a one
3 elements, first is 3Second is one
A general array
```

In the function `arrayToString` the first case matches arrays of 3 elements, where the first is the integer 1, the second case matches arrays of 3 elements, where the second is a 1 and names the first `x`, and the final case matches all arrays and works as a default match case. As demonstrated, the cases are treated from first to last, and only the expression of the first case that matches is executed.

For record pattern, we use the field names to specify matching criteria. This is demonstrated in Listing 15.17.

Listing 15.17 recordPattern.fsx:
Variable patterns for records to match on field values.

```
type Address = {street : string; zip : int; country : string}
let contact : Address = {
    street = "Universitetsparken 1";
    zip = 2100;
    country = "Denmark"}
let getZip (adr : Address) : int =
  match adr with
    {street = _; zip = z; country = _} -> z

printfn "The zip-code is: %d" (getZip contact)
```

```
$ fsharpc --nologo recordPattern.fsx && mono recordPattern.exe
The zip-code is: 2100
```

Here, the record type `Address` is created, and in the function `getZip`, a variable pattern `z` is created for naming zip values, and the remaining fields are ignored. Since the fields are named, then the pattern match needs not mention the ignored fields, and the example match is equivalent to `{zip = z} -> z`. The curly brackets are required for record patterns.

↑ is there no way of expressing that one intends to match all fields?

Discriminated union patterns are similar. For discriminated unions with arguments, the arguments can be match as constants, variables, or wildcards. A demonstration is given in Listing 15.18.

Listing 15.18 unionPattern.fsx:
Matching on discriminated union types.

```
type vector =
  Vec2D of float * float
  | Vec3D of float * float * float

let project (vec : vector) : vector =
  match vec with
    Vec3D (a, b, _) -> Vec2D (a, b)
    | v -> v

let v = Vec3D (1.0, -1.2, 0.9)
printfn "%A -> %A" v (project v)
```

```
$ fsharpc --nologo unionPattern.fsx && mono unionPattern.exe
Vec3D (1.0,-1.2,0.9) -> Vec2D (1.0,-1.2)
```

In the `project`-function, three dimensional vectors are projected to two dimensions by removing the third element. Two dimensional vectors are unchanged. The example uses the wildcard pattern to emphasize, that the third element of three dimensional vectors is ignored. Named arguments can also be matched, in which case ";" is used to delimit the fields in the match instead of ",".

15.7 Disjunctive and conjunctive patterns

Patterns may be combined logically ~~by the~~ disjunctively using the *"|"* lexeme and conjunctively using the *"&"* lexeme. *Disjunctive patterns* combine as fall-through as illustrated in Listing 15.19.

· |
· &
· disjunctive patt

Listing 15.19 disjunctivePattern.fsx:
Patterns can be combined logically as 'or' syntax structures.

```
let wovel (c : char) : bool =
  match c with
    'a' | 'e' | 'i' | 'o' | 'u' | 'y' -> true
    | _ -> false

String.iter (fun c -> printf "%A " (wovel c)) "abcdefg"
```

```
$ fsharpc --nologo disjunctivePattern.fsx && mono disjunctivePattern.exe
true false false false true false false
```

Here one or more cases must match for the final case expression, and thus, any vowel results in the same case expression `true`. All else is matched with the wildcard pattern.

For *conjunctive patterns* all patterns must match, which is illustrated in Listing 15.20.

· conjunctive patt

Listing 15.20 conjunctivePattern.fsx:
Patterns can be combined logically as 'or' syntax structures.

```
let is11 (v : int * int) : bool =
  match v with
    (1,_) & (_,1) -> true
    | _ -> false

printfn "%A" (List.map is11 [(0,0); (0,1); (1,0); (1,1)])
```

```
$ fsharpc --nologo conjunctivePattern.fsx && mono conjunctivePattern.exe
[false; false; false; true]
```

In this case, we separately check the elements of a pair for the constant value 1, and return true only when both elements are 1. In many cases, conjunctive patters can be replaced by more elegant matches, e.g., using tuples, and in the above example a single case `(1,1) -> true` would have been simpler. Nevertheless, conjunctive patterns are useful together with active patterns, to be discussed below.

15.8 Active Pattern

The concept of patterns is extendable to functions. Such functions are called *active patterns*, and active patterns comes in two flavors: regular and option types. The active pattern cases are constructed as function bindings, but using a special notation. They all take the pattern input as last argument, and may take further preceding arguments. The syntax one of the two following

· active patterns

rephrase

Listing 15.21 Syntax for binding active patterns to expressions.

```
let (|<caseName>|[_| ]) [ <arg> [<arg> ... ]] <inputArgument> = <expr>
let (|<caseName>|<caseName>|...|<caseName>|) <inputArgumetn> = <expr>
```

When using the `(|<caseName>|_|])` variants, then the active pattern function must return an option type. The multi-case variant `(|<caseName>|<caseName>|...|<caseName>|)` must return a `Fsharp.Core.Choice` type. All other variants can return any type. There are no restrictions on arguments `<arg>`, and `<inputArgumetn>` is the input pattern to be matched. Notice in particular that the multi-case variant only takes one argument and cannot be combined with the option-type syntax. Below we will demonstrate how the various patterns are used by example.

The single case, `(|<caseName>|])` matches all, and is useful for extracting information from complex types, as demonstrated in Listing 15.22.

Listing 15.22 activePattern.fsx:
Single case active pattern for deconstructing complex types.

```
1  type vec = {x : float; y : float}
2  let (|Cartesian|) (v : vec) = (v.x, v.y)
3  let (|Polar|) (v : vec) = (sqrt(v.x*v.x + v.y * v.y), atan2 v.y v.x)
4  let printCartesian (p : vec) : unit =
5      match p with
6        Cartesian (x, y) -> printfn "%A:\n Cartesian (%A, %A)" p x y
7  let printPolar (p : vec) : unit =
8      match p with
9        Polar (a, d) -> printfn "%A:\n Cartesian (%A, %A)" p a d
10
11 let v = {x = 2.0; y = 3.0}
12 printCartesian v
13 printPolar v
```

```
1  $ fsharpc --nologo activePattern.fsx && mono activePattern.exe
2  {x = 2.0;
3   y = 3.0;}:
4   Cartesian (2.0, 3.0)
5  {x = 2.0;
6   y = 3.0;}:
7   Cartesian (3.605551275, 0.9827937232)
```

Here we define a record to represent two-dimensional vectors and two different single case active patterns. Note that in the binding of the active pattern functions in line 2 and 3, the argument is the input expression `match <inputExpr> with ...;` see Listing 15.3. However, the argument for the cases in line 6 and 9 are names bound to the output of the active pattern function.

Both `Cartesian` and `Polar` matches a vector record, but they dismantle the contents differently. For an alternative solution using Class types, see Section 20.1.

More complicated behavior is obtainable by supplying additional arguments to the single case. This is demonstrated in Listing 15.23.

Listing 15.23 activeArgumentsPattern.fsx:
All but the multi-case active pattern may take additional arguments.

```
type vec = {x : float; y : float}
let (|Polar|) (o : vec) (v : vec) =
  let x = v.x - o.x
  let y = v.y - o.y
  (sqrt(x*x + y * y), atan2 y x)
let printPolar (o : vec) (p : vec) : unit =
    match p with
      Polar o (a, d) -> printfn "%A:\n Cartesian (%A, %A)" p a d

let v = {x = 2.0; y = 3.0}
let offset = {x = 1.0; y = 1.0}
printPolar offset v
```

```
$ fsharpc --nologo activeArgumentsPattern.fsx
$ mono activeArgumentsPattern.exe
{x = 2.0;
 y = 3.0;}:
 Cartesian (2.236067977, 1.107148718)
```

Here we supply an offset, which should be subtracted prior to calculating lengths and angles. Notice in line 8, that the argument is given prior to the result binding.

Active pattern functions return option types are called *partial pattern functions*. The option type allows for specifying mismatches as illustrated in Listing 15.24.

· partial pattern functions

Listing 15.24 activeOptionPattern.fsx:
Option type active patterns mismatches on None results.

```
let (|Div|_|) (e,d) = if d <> 0.0 then Some (e/d) else None

let safeDiv (p : float * float) =
    match p with
      | (0.0, 0.0) -> printfn "Div %A = undefined" p
      | Div res -> printfn "Div %A = %A" p res
      | _ -> printfn "Div %A = infinity" p

List.iter safeDiv [(1.0,1.0); (0.0,1.0); (1.0,0.0); (0.0,0.0)]
```

```
$ fsharpc --nologo activeOptionPattern.fsx
$ mono activeOptionPattern.exe
Div (1.0, 1.0) = 1.0
Div (0.0, 1.0) = 0.0
Div (1.0, 0.0) = infinity
Div (0.0, 0.0) = undefined
```

In the example, we use the (|<caseName>|_|]) variant to indicate, that the active pattern returns an option type. Nevertheless, the result binding `res` in line 6 uses the underlying value of `Some`. And in contrast to the two previous examples of single case patterns, the value `None` results in a mismatch, thus in this case, if the denominator is `0.0`, ~~then~~ `Div res` does not match but the wildcard pattern does.

Multicase active patterns works similarly to discriminated unions without arguments. An example is given in Listing 15.25.

· multicase active patterns

Listing 15.25 activeMultiCasePattern.fsx:
Multi-case active patterns have a syntactical structure similar to discriminated unions.

```
1  let (|Gold|Silver|Bronze|) inp =
2    if inp = 0 then Gold
3    elif inp = 1 then Silver
4    else Bronze
5
6  let intToMedal (i : int) =
7      match i with
8        Gold -> printfn "%d: Its gold!" i
9        | Silver -> printfn "%d: Its silver." i
10       | Bronze -> printfn "%d: Its no more than bronze." i
11
12 List.iter intToMedal [0..3]
```

```
1  $ fsharpc --nologo activeMultiCasePattern.fsx
2  $ mono activeMultiCasePattern.exe
3  0: Its gold!
4  1: Its silver.
5  2: Its no more than bronze.
6  3: Its no more than bronze.
```

In this example, we define three cases in line 1. The result of the active pattern function must be one of these cases. For the `match`-expression, the match is based on the output of the active pattern function, hence in line 8, the case expression is executed, when the result of applying the active pattern function to the input expression `i` is `Gold`. In this case, a solution based on discriminated unions would probably be clearer.

15.9 Static and dynamic type pattern

Input pattern can also be matched on type. For *static type matching* the matching is performed at compile time, and indicated using the *":"* lexeme followed by the type name to be matched. Static type matching is further used as input to the type inference performed at compile time to infer non-specified types as illustrated in Listing 15.26.

· static type patt
· :

I think we should resist teaching the students these quite ad-hoc F# features.

Listing 15.26 staticTypePattern.fsx:
Static matching on type binds the type of other values by type inference.

```
let rec sum lst =
    match lst with
      (n : int) :: rest -> n + (sum rest)
      | [] -> 0

printfn "The sum is %d" (sum [0..3])
```

```
$ fsharpc --nologo staticTypePattern.fsx && mono staticTypePattern.exe
The sum is 6
```

Here the head of the list `n` in the list pattern is explicitly matched as an integer, and the type inference system thus concludes that `lst` must be a list of integers.

In contrast to static type matching, *dynamic type matching* is performed at runtime, and indicated using the "`:?`" lexeme followed by a type name. Dynamic type patterns allow for matching generic values at runtime. This is an advanced topic, which is included here for completeness. An example is given in Listing 15.27.

· dynamic type p
· :?

Listing 15.27 dynamicTypePattern.fsx:
Static matching on type binds the type of other values by type inference.

```
let isString (x : obj) : bool =
    match x with
      :? string -> true
      | _ -> false

let a = "hej"
printfn "Is %A a string? %b" a (isString a)
let b = 3
printfn "Is %A a string? %b" b (isString b)
```

```
$ fsharpc --nologo dynamicTypePattern.fsx && mono dynamicTypePattern.exe
Is "hej" a string? true
Is 3 a string? false
```

In F# all types are also objects, whose type is denoted `obj`. Thus, the example uses the generic type when defining the argument to `isString`, and then dynamic type pattern matching for further processing. See Chapter 20 for more on objects. Dynamic type patterns are often used for analysing exceptions, which is discussed in Section 18.1. While dynamic type patterns are useful, they imply runtime checking, and **it is almost always better to prefer compile time than runtime type checking.**

Advice