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,

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. Here we extract the elements of a pair twice. First by binding to `x` and `y`, and second

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
```

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. 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. Here we used a pattern for the discriminated union cases and a wildcard

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
```

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 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,

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 1.4 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
```

The wildcard pattern is used to disregard the first element in a pair while iterating

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
```

over the complete list. It is good practice to **use wildcard patterns to emphasize unused values.** ⋆

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

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. Here we use an anonymous function ex-

**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
```

pression 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 content has yet to be decided. Thus, mockup functions can be used as loose place-holders while experimenting with program design.

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

1.1 Wildcard Pattern

A *wildcard pattern* is denoted "_" and matches anything, see e.g., Listing 1.10. In

**Listing 1.10 wildcardPattern.fsx:
Constant patterns match 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
```

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.

1.2 Constant and Literal Patterns

A *constant pattern* matches any input pattern with constants, see e.g., Listing 1.11. In this example, the input pattern is queried for a match with `0`, `1`, or the wildcard

Listing 1.11 constPattern.fsx:
Constant patterns match 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
```

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. Note that matching is performed in a lazy manner and stops at 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 match the input.

Constants can also be pre-bound by the `[<Literal>]` attribute for value-bindings. This is demonstrated in Listing 1.12. The attribute 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.

Listing 1.12 literalPattern.fsx:
A variant of constant patterns is 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..."
```

1.3 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 patterns are also allowed in conjunction with with records and arrays. This is demonstrated in Listing 1.13. In this example, the value identifier `n` has the function

Listing 1.13 variablePattern.fsx:
Variable patterns are useful for, e.g., extracting and naming fields

```
let (name, age) = ("Jon", 50)
let getAgeString (age : int) : string =
  match age with
    0 -> "a 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
```

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 1.4.

1.4 Guards

A *guard* is a pattern used together with match-expressions including the *when*-keyword, as shown in Listing 1.5. Here guards are used to iteratively carve out subset of

Listing 1.14 guardPattern.fsx:
Guard expressions can be used with other patterns to restrict matches.

```fsharp
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
```

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.

1.5 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 processing lists, as shown in Listing 1.15 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

Listing 1.15 listPattern.fsx:
Recursively parsing a list with 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
```

be matched explicitly named elements, as demonstrated in the `sumThree` function. The elements to be matched can be any mix of constants and variables.

It is also possible to match on a series of cons-operators. For example `elm0 :: elm1 :: rest` would match a list with at least two elements, where the first will be bound to `elm1`, the second to `elm2`, and the remainder to `rest`.

1.6 Array, Record, and Discriminated Union Patterns

Array, *record*, and *discriminated union patterns* are direct extensions on constant, variable, and wildcard patterns. Listing 1.16 gives examples of array patterns. 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 patterns, we use the field names to specify matching criteria. This is demonstrated in Listing 1.17. 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, the pattern match need not

Listing 1.16 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 1"
    | [|x;1;_|] -> "3 elements, first is " + (string x) +
   " Second 1"
    | 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 1
3 elements, first is 3 Second 1
A general array
```

mention the ignored fields, and the example match is equivalent to `{zip = z} -> z`. The curly brackets are required for record patterns.

Discriminated union patterns are similar. For discriminated unions with arguments, the arguments can be matched as constants, variables, or wildcards. A demonstration is given in Listing 1.18. 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 instead of "`,`" to delimit the fields in the match.

Listing 1.17 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
```

Listing 1.18 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)
```

1.7 Disjunctive and Conjunctive Patterns

Patterns may be combined using the "|" and "&" lexemes. These patterns are called disjunctive and conjunctive patterns, respectively, and work similarly to their logical operator counter parts, "||" and "&&".

Disjunctive patterns require at least one pattern to match, as illustrated in Listing 1.19. Here one or more cases must match for the final case expression, and thus,

Listing 1.19 disjunctivePattern.fsx:
Patterns can be combined logically as 'or' syntax structures.

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

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

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

any vowel results in the value `true`. Everything else is matched with the wildcard pattern.

For *conjunctive patterns*, all patterns must match, which is illustrated in Listing 1.20. 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 used together with active patterns, to be discussed below.

1.8 Active Patterns

The concept of patterns is extendable to functions. Such functions are called *active patterns*, and active patterns come 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 for active patterns is one of,

**Listing 1.20 conjunctivePattern.fsx:
Patterns can be combined logically as 'and' 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]
```

Listing 1.21: Syntax for binding active patterns to expressions.

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

When using the `(|<caseName>|_|])` variants, then the active pattern function must return an option type. `(|<caseName>|<caseName>|...|<caseName>|)` is the multi-case variant and 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 by example how the various patterns are used.

The single case, `(|<caseName>|])`, matches all and is useful for extracting information from complex types, as demonstrated in Listing 1.22. 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 1.5. 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` match a vector record, but they dismantle the contents differently. For an alternative solution using Class types, see **??**.

More complicated behavior is obtainable by supplying additional arguments to the single case. This is demonstrated in Listing 1.23. Here we supply an offset, which

**Listing 1.22 activePattern.fsx:
Single case active pattern for deconstructing complex types.**

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

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

```
$ fsharpc --nologo activePattern.fsx && mono activePattern.exe
{x = 2.0;
 y = 3.0;}:
 Cartesian (2.0, 3.0)
{x = 2.0;
 y = 3.0;}:
 Polar (3.605551275, 0.9827937232)
```

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 1.24. 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 work similarly to discriminated unions without arguments. An example is given in Listing 1.25. 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

**Listing 1.23 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)
```

pattern function to the input expression `i` is `Gold`. In this case, a solution based on discriminated unions would probably be clearer.

Listing 1.24 activeOptionPattern.fsx:
Option type active patterns mismatch 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
```

Listing 1.25 activeMultiCasePattern.fsx:
Multi-case active patterns have a syntactical structure similar to discriminated unions.

```
let (|Gold|Silver|Bronze|) inp =
  if inp = 0 then Gold
  elif inp = 1 then Silver
  else Bronze

let intToMedal (i : int) =
    match i with
      Gold -> printfn "%d: It's gold!" i
      | Silver -> printfn "%d: It's silver." i
      | Bronze -> printfn "%d: It's no more than bronze." i

List.iter intToMedal [0..3]
```

```
$ fsharpc --nologo activeMultiCasePattern.fsx
$ mono activeMultiCasePattern.exe
0: It's gold!
1: It's silver.
2: It's no more than bronze.
3: It's no more than bronze.
```

1.9 Static and Dynamic Type Pattern

Input patterns 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 1.26. Here the head of the list `n` in the list pattern is explicitly matched as

Listing 1.26 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
```

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 runtimes 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 1.27. 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 **??** for more on objects. Dynamic type patterns are often used for analyzing exceptions, which is discussed in **??**. While dynamic type patterns are useful, they imply runtime checking, and **it is almost always better to prefer compile time over runtime type checking.**

Listing 1.27 dynamicTypePattern.fsx:
Dynamic 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
```