13 Recursion

Recursion is a central concept in F# and is used to control flow in loops without the `for` and `while` constructions. Figure 13.1 illustrates the concept of an infinite loop with recursion.

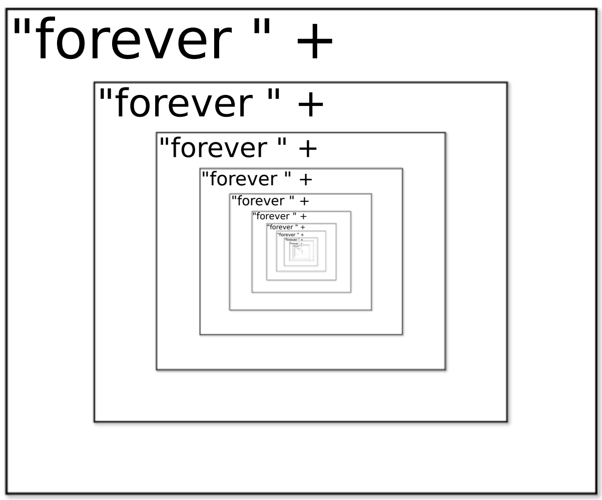

Figure 13.1: An infinitely long string of "forever forever forever...", conceptually calculated by `let rec forever () = "fsharp " + (forever ())`.

13.1 Recursive functions

A *recursive function* is a function, which calls itself, and the syntax for defining recursive functions is an extension of that for regular functions: · recursive function

Listing 13.1 Syntax for defining one or more mutually dependent recursive functions.

```
let rec <ident> = <expr> {and <ident> = <expr>} [in] <expr>
```

From a compiler point of view, the *`rec`* is necessary, since the function is used before the compiler has completed its analysis. If two functions are mutually recursive, then they must be defined jointly using the *`and`* keyword. · `rec` · `and`

An example of a recursive function that counts from 1 to 10 similarly to Listing 8.5 is given in Listing 13.2.

**Listing 13.2 countRecursive.fsx:
Counting to 10 using recursion.**

```
let rec prt a b =
  if a > b then
    printf "\n"
  else
    printf "%d " a
    prt (a + 1) b

prt 1 10
```

```
$ fsharpc --nologo countRecursive.fsx && mono countRecursive.exe
1 2 3 4 5 6 7 8 9 10
```

Here the `prt` function calls itself repeatedly, such that the first call is `prt 1 10`, which calls `prt 2 10`, and so on until the last call `prt 11 10`. Each time `prt` is called, new bindings named `a` and `b` are made to new values. This is illustrated in Figure 13.2. The old values are no longer accessible as indicated by subscript in the figure. E.g., in `prt`$_3$ the scope has access to `a`$_3$ but not `a`$_2$ and `a`$_1$. Thus, in this program, process is similar to a `for` loop, where the counter is `a` and in each loop its value is reduced.

The structure of the function is typical for recursive functions. They very often follow the following pattern.

Listing 13.3 Recursive functions consists of a stopping criterium, a stopping expression, and a recursive step.

```
let rec f a =
  if <stopping condition>
  then <stopping step>
  else <recursion step>
```

The *`match`* – *`with`* are also very common conditional structures. In Listing 13.2 `a > b` is the *stopping condition*, `printfn "\n"` is *stopping step*, and `printfn "\%d " a; prt (a + 1) b` is the *recursion step*.

· `match`
· `with`
· stopping condition
· stopping step
· recursion step

13.2 The call stack and tail recursion

Fibonacci's sequence of numbers is a recursive sequence of numbers with relations to the Golden ratio and structures in biology. Fibonacci's sequence is the sequence of numbers $1, 1, 2, 3, 5, 8, 13, \ldots$. The sequence starts with $1, 1$ and the next number is recursively given as the sum of the two previous. A direct implementation of this is given in Listing 8.7.

```
$ fsharpc countRecursive.fsx && mono countRecursive.exe
```

prt 1 10

$\mathtt{prt}_1$: $\mathtt{a}_1$= 1, $\mathtt{b}_1$= 10

1 > 10: ✗
printf "1 "
prt 2 10

$\mathtt{prt}_2$: $\mathtt{a}_2$= 2, $\mathtt{b}_2$= 10

2 > 10: ✗
printf "2 "
prt 3 10

$\mathtt{prt}_3$: $\mathtt{a}_3$= 3, $\mathtt{b}_3$= 10

3 > 10: ✗
printf "3 "
prt 4 10

$\ddots$

prt 11 10

$\mathtt{prt}_{11}$: $\mathtt{a}_{11}$= 1, $\mathtt{b}_{11}$= 10

11 > 10 ✓
printf "\\n"

()

$\iddots$

()

()

()

()

Figure 13.2: Illustration of the recursion used to write the sequence "1 2 3 ... 10" in line 8 in Listing 13.2. Each frame corresponds to a call to `prt`, where new values overshadow old. All return `unit`.

**Listing 13.4 fibRecursive.fsx:
The n'th Fibonacci number using recursive.**

```
let rec fib n =
  if n < 1 then
    0
  elif n = 1 then
    1
  else
    fib (n - 1) + fib (n - 2)

for i = 0 to 10 do
  printfn "fib(%d) = %d" i (fib i)
```

```
$ fsharpc --nologo fibRecursive.fsx && mono fibRecursive.exe
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34
fib(10) = 55
```

Here we extended the sequence to $0, 1, 1, 2, 3, 5, \ldots$ and starting sequence $0, 1$ allowing us to define all $\mathrm{fib}(n) = 0$, $n < 1$. Thus, our function is defined for all integers, and for the irrelevant negative arguments, it fails gracefully by returning 0. This is a general advice: **make functions that fail gracefully.**

Advice

A visualization of the calls and the scopes created by `fibRecursive` is shown in Figure 13.3. The figure illustrates that each recursive step results in two calls to the function, thus creating two new scopes. And it gets worse. Figure 13.4 illustrates the tree of calls for `fib 5`. Thus a call to the function `fib` generates a tree of calls that is five levels deep and has `fib(5)` number of nodes. In general for the program in Listing 13.4, a call to `fib(n)` produces a tree with $\texttt{fib(n)} \leq c\alpha^n$ calls to the function for some positive constant c and $\alpha \geq \frac{1+\sqrt{5}}{2} \sim 1.6$[1]. Each call takes time and requires memory, and we have thus created a slow and somewhat memory intensive function. This is a hugely ineffective implementation of calculating entries into Fibonacci's sequence, since many of the calls are identical. E.g., in Figure 13.4 `fib 1` is called five times. Before we examine a faster algorithm, we first need to discuss how F# executes function calls.

When a function is called, then memory is dynamically allocated internally for the function on what is known as the *call stack*. Stacks are used for many things in programming, but typically the call stack is considered special since it is almost always implicitly part of any program execution. Hence, it is often just referred to as *The Stack*. When a function is called, a new *stack frame* is stacked (pushed) on the call stack including its arguments, local storage such as mutable values, and where execution should return to when the function is finished. When the function finishes, the stack frame is unstacked (popped) and in its stead, the return value of the function is stacked. This return value is then unstacked and used by the caller. After unstacking the return value, the call stack is identical to its state prior to the call. Figure 13.5 shows snapshots of the call stack, when calling `fib 5` in Listing 13.4. The call first stacks a frame onto the call stack with everything needed to execute the

· call stack
· The Stack
· stack frame

[1] Jon: https://math.stackexchange.com/questions/674533/prove-upper-bound-big-o-for-fibonaccis-sequence

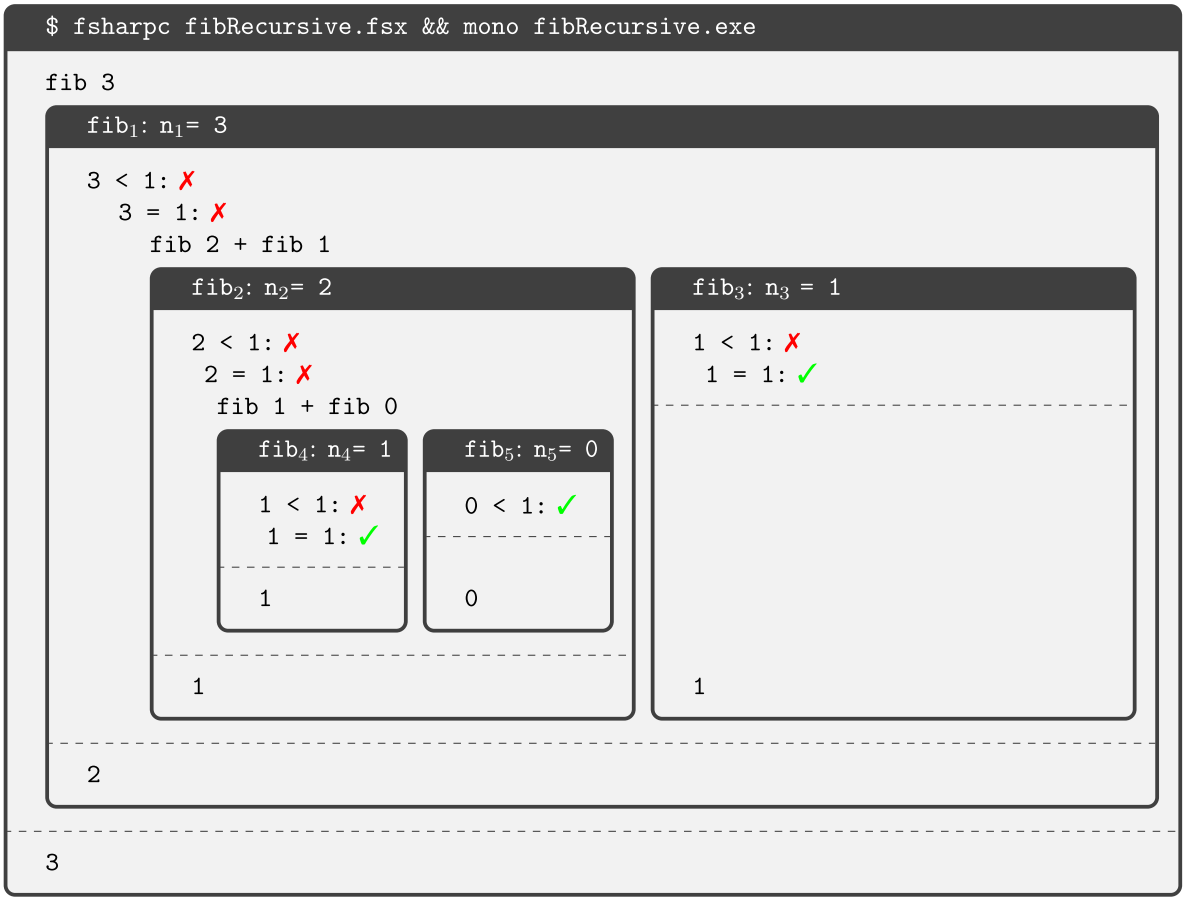

Figure 13.3: Illustration of the recursion used to write the sequence "1 2 3 ... 10" in line 8 in Listing 13.2. Each frame corresponds to a call to `fib`, where new values overshadow old.

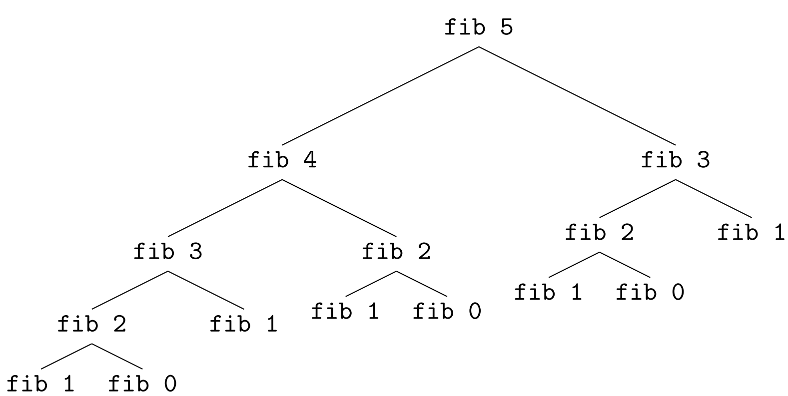

Figure 13.4: The function calls involved in calling `fib 5`.

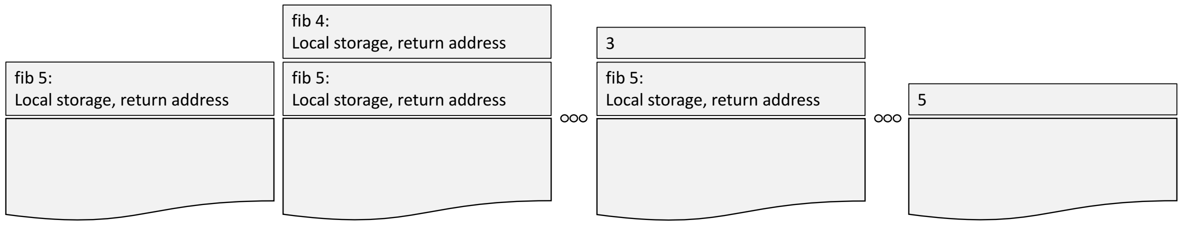

Figure 13.5: A call to `fib 5` in Listing 13.4 starts a sequence of function calls and stack frames on the call stack.

function body plus a reference to where the return to, when the execution is finished. Then the body of `fib` is executed, which includes calling `fib 4` and `fib 3` in turn. The call to `fib 4` stacks a frame onto the call stack, and its body is executed. Once execution is returned from the call to `fib 4`, the result of the function is on top of the stack. It is unstacked, saved and the call to `fib 3` is treated equally. When the end of `fib 5` is reached, its frame is unstacked, and its result is stacked. In this way, the call stack is returned to its original state except for the result of the function, and execution is returned to the point right after the original call to `fib 5`. Thus, for Listing 13.4 $\mathcal{O}(\alpha^n)$, $\alpha = \frac{1+\sqrt{5}}{2}$ stacking operations are performed for a call to `fib n`. The $\mathcal{O}(f(n))$ is the *Landau symbol* used to denote the order of a function, such that if $g(n) = \mathcal{O}(f(n))$ then there exists two real numbers $M > 0$ and a n_0 such that for all $n \geq n_0$, $|g(n)| \leq M|f(n)|$.[2] As indicated by the tree in Figure 13.4, the call tree is at most n high, which corresponds to a maximum of n additional stack frames as compared to the starting point.

· Landau symbol

The implementation of Fibonacci's sequence in Listing 13.4 can be improved to run faster and use less memory. One such algorithm is given in Listing 13.5

Listing 13.5 fibRecursiveAlt.fsx:
A fast, recursive implementation of Fibonacci's numbers. Compare with Listing 13.4.

```
let fib n =
  let rec fibPair n pair =
    if n < 2 then pair
    else fibPair (n - 1) (snd pair, fst pair + snd pair)
  if n < 1 then 0
  elif n = 1 then 1
  else fibPair n (0, 1) |> snd

printfn "fib(10) = %d" (fib 10)
```

```
$ fsharpc --nologo fibRecursiveAlt.fsx && mono fibRecursiveAlt.exe
fib(10) = 55
```

Calculating the 45th Fibonacci number a MacBook Pro, with a 2.9 GHz Intel Core i5 using Listing 13.4 takes about 11.2s while using Listing 13.5 is about 224 times faster and only takes 0.050s. The reason is that `fib` in Listing 13.5 calculates every number in the sequence once and only once by processing the list recursively while maintaining the previous two values needed to calculate the next in the sequence. I.e., the function `helper` transforms the pair `(a,b)` to `(b,a+b)` such that, e.g., the 4th and 5th pair `(3,5)` is transformed into the 5th and the 6th pair `(5,8)` in the sequence. What complicates the algorithm is that besides the transformation, we must keep track of when to stop, which here is done using a counter variable, that is recursively reduced by 1 until our stopping criterium.

[2]Jon: **Introduction of Landau notation needs to be moved earlier, since it used in Collections chapter.**

Listing 13.5 also uses much less memory than Listing 13.4, since its recursive call is the last expression in the function, and since the return value of two recursive calls to `helper` is the same as the return value of the last. In fact, the return value of any number of recursive calls to `helper` is the return value of the last. This structure is called *tail-recursion*. Compilers can easily optimize the call stack usage for tail recursion, since when in this example `helper` calls itself, then its frame is no longer needed, and may be replaced by the new `helper` with the slight modification, that the return point should be to `fib` and not the end of the previous `helper`. Once the recursion reaches the stopping criteria, then instead of popping a long list of calls of `helper` frames, then there is only one, and the return value is equal to the return value of the last call and the return point is to `fib`. Thus, many stack frames in tail recursion are replaced by one. Hence, **prefer tail-recursion whenever possible.**

· tail-recursion

Advice

13.3 Mutual recursive functions

Functions that recursively call each other are called *mutually recursive* functions. F# offers the *`let`* – *`rec`* – *`and`* notation for co-defining mutually recursive functions. As an example, consider the function `even : int -> bool`, which returns true if its argument is even and false otherwise, and the opposite function `odd : int -> bool`. A mutually recursive implementation of these functions can be developed from the following relations: `even 0 = true`, `odd 0 = false`, and for $n > 0$, `even n = odd (n-1)`, which implies that for $n > 0$, `odd n = even (n-1)`:

· mutually recursive
· `let`
· `rec`
· `and`

Listing 13.6 mutuallyRecursive.fsx:
Using mutual recursion to implement even and odd functions.

```
let rec even x =
  if x = 0 then true
  else odd (x - 1)
and odd x =
  if x = 0 then false
  else even (x - 1);;

let w = 5;
printfn "%*s %*s %*s" w "i" w "even" w "odd"
for i = 1 to w do
  printfn "%*d %*b %*b" w i w (even i) w (odd i)
```

```
$ fsharpc --nologo mutuallyRecursive.fsx && mono mutuallyRecursive.exe
    i  even   odd
    1 false  true
    2  true false
    3 false  true
    4  true false
    5 false  true
```

Notice that in the lightweight notation the `and` must be on the same indentation level as the original `let`.

Without the `and` keyword, F# will issue a compile error at the definition of `even`. However, it is possible to implement mutual recursion by using functions as an argument, e.g.,

Listing 13.7 mutuallyRecursiveAlt.fsx:
Mutual recursion without the and keyword needs a helper function.

```
let rec evenHelper (notEven: int -> bool) x =
  if x = 0 then true
  else notEven (x - 1)

let rec odd x =
  if x = 0 then false
  else evenHelper odd (x - 1);;

let even x = evenHelper odd x

let w = 5;
printfn "%*s %*s %*s" w "i" w "Even" w "Odd"
for i = 1 to w do
  printfn "%*d %*b %*b" w i w (even i) w (odd i)
```

```
$ fsharpc --nologo mutuallyRecursiveAlt.fsx
$ mono mutuallyRecursiveAlt.exe
    i  Even   Odd
    1 false  true
    2  true false
    3 false  true
    4  true false
    5 false  true
```

But, Listing 13.6 is clearly to be preferred over Listing 13.7.

In the above, we used the `even` and `odd` function problems to demonstrate mutual recursion. There is, of course, a much simpler solution, which does not use recursion at all:

Listing 13.8 parity.fsx:
A better way to test for parity without recursion.

```
let even x = (x % 2 = 0)
let odd x = not (even x)
```

This is to be preferred anytime as the solution to the problem. [3]

[3] Jon: **Here it would be nice to have an intermezzo, giving examples of how to write a recursive program by thinking the problem has been solved.**

14 | Programming with types

F# is a strongly typed language, meaning that types are known or inferred at compile time. In the previous chapters, we have used *primitive types* such as `float` and `bool`, function types, and compound types implicitly defined by tuples. These types are used for simple programming tasks, and everything that can be programmed can be accomplished using these types. However, larger programs are often easier to read and write when using more complicated type structures. In this chapter, we will discuss type abbreviations, enumerated types, discriminated unions, records, and structs. Class types are discussed in depth in Chapter 20.

· primitive types

14.1 Type abbreviations

F# allows for renaming of types, which is called *type abbreviation* or *type aliasing*. The syntax is

· type abbreviation
· type aliasing

Listing 14.1 Syntax for type abbreviation.

```
type <ident> = <aType>
```

where the identifier is a new name, and the type-name is an existing or a compound of existing types. E.g., in Listing 14.2 several type abbreviations are defined.

Listing 14.2 typeAbbreviation.fsx:
Defining 3 type abbreviations, two of which are compound types.

```
type size = int
type position = float * float
type person = string * int
type intToFloat = int -> float

let sz : size = 3
let pos : position = (2.5, -3.2)
let pers : person = ("Jon", 50)
let conv : intToFloat = fun a -> float a
printfn "%A, %A, %A, %A" sz pos pers (conv 2)
```

```
$ fsharpc --nologo typeAbbreviation.fsx && mono typeAbbreviation.exe
3, (2.5, -3.2), ("Jon", 50), 2.0
```

Here we define the abbreviations `size`, `position`, `person`, and `intToFloat`, and later make bindings enforcing the usage of these abbreviations.

Type abbreviations are used as short abbreviations of longer types, and they add semantic content to the program text, thus making programs shorter and easier to read. Type abbreviations allow the programmer to focus on the intended structure at a higher level by, e.g., programming in terms of a type `position` rather than `float * float`. Thus, they often result in programs with fewer errors. Type abbreviations also make maintenance easier. For instance, if we at a later stage decide that positions only can have integer values, then we only need to change the definition of the type abbreviation, not every place, a value of type `position` is used.

14.2 Enumerations

Enumerations or *enums* for short are types with named values. Names in enums are assigned to a subset of integer or char values. Their syntax is as follows:

· enumerations
· enums

Listing 14.3 Syntax for enumerations.

```
type <ident> =
  [ | ] <ident> = <integerOrChar>
  | <ident> = <integerOrChar>
  | <ident> = <integerOrChar>
  ...
```

An example of using enumerations is given in Listing 14.4.

Listing 14.4 enum.fsx:
An enum type acts as a typed alias to a set of integers or chars.

```
type medal =
  Gold = 0
  | Silver = 1
  | Bronze = 2

let aMedal = medal.Gold
printfn "%A has value %d" aMedal (int aMedal)
```

```
$ fsharpc --nologo enum.fsx && mono enum.exe
Gold has value 0
```

In the example, we define an enumerated type for medals, which allows us to work with the names rather than the values. Since the values most often are arbitrary, we can program using semantically meaningful names instead. Being able to refer to an underlying integer type allows us to interface with other – typically low-level – programs that require integers, and to perform arithmetic. E.g., for the medal example, we can typecast the enumerated types to integers and calculate an average medal harvest.

14.3 Discriminated Unions

A discriminated union is a union of a set of named cases. These cases can further be of specified types. The syntax for defining a discriminated union is as follows:

Listing 14.5 Syntax for type abbreviation.

```
[<attributes>]
type <ident> =
  [| ]<ident> [of [<ident> :] <aType> [* [<ident> :] <aType> ...]]
  | <ident> [of [<ident> :] <aType> [* [<ident> :] <aType> ...]]
  ...
```

Discriminated unions are reference types, i.e., their content is stored on *The Heap*, see Section 6.8 for a discussion on reference types. Since they are immutable, there is no risk of side-effects. As reference types, when used as arguments to and returned from a function, then only a reference is passed. This is in contrast to value types, which transport a complete copy of the data structure. Discriminated unions are thus effective for large data structures. However, there is a slight overhead, since working with the content of reference types is indirect through their reference. Discriminated unions can also be represented as structures using the `[<Struct>]` attribute, in which case they are value types. See Section 14.5 for a discussion on structs.

· The Heap

An example just using the named cases but no further specification of types is given in Listing 14.6.

Listing 14.6 discriminatedUnions.fsx:
A discriminated union of medals. Compare with Listing 14.4.

```
type medal =
  Gold
  | Silver
  | Bronze

let aMedal = medal.Gold
printfn "%A" aMedal
```

```
$ fsharpc --nologo discriminatedUnions.fsx
$ mono discriminatedUnions.exe
Gold
```

Here we define a discriminated union as three named cases signifying three different types of medals. Comparing with the enumerated type in Listing 14.4, we see that the only difference is that the cases of the discriminated unions have no value. A commonly used discriminated union is the *option type*, see Section 18.2 for more detail.

· option type

Discriminated unions may also be used to store data. Where the names in enumerated types are aliases of single values, the names used in discriminated unions can hold any value specified at the time of creation. An example is given in Listing 14.7.

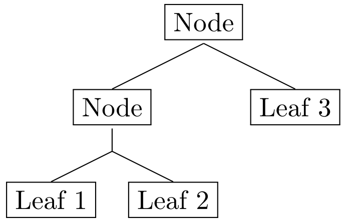

Figure 14.1: The tree with 3 leaves.

Listing 14.7 discriminatedUnionsOf.fsx:
A discriminated union using explicit subtypes.

```
type vector =
  Vec2D of float * float
  | Vec3D of x : float * y : float * z : float

let v2 = Vec2D (1.0, -1.2)
let v3 = Vec3D (x = 1.0, z = -1.2, y = 0.9)
printfn "%A and %A" v2 v3
```

```
$ fsharpc --nologo discriminatedUnionsOf.fsx
$ mono discriminatedUnionsOf.exe
Vec2D (1.0,-1.2) and Vec3D (1.0,0.9,-1.2)
```

In this case, we define a discriminated union of two and three-dimensional vectors. Values of these types are created using their names followed by a tuple of their arguments. As can be seen, the arguments may be given field names, and if they are, then the names may be used when creating values of this type. As also demonstrated, the field names can be used to specify the field values in arbitrary order. However, values for all fields must be given.

Discriminated unions can be defined recursively. This feature is demonstrated in Listing 14.8.

Listing 14.8 discriminatedUnionTree.fsx:
A discriminated union modelling binary trees.

```
type Tree =
  Leaf of int
  | Node of Tree * Tree

let one = Leaf 1
let two = Leaf 2
let three = Leaf 3
let tree = Node (Node (one, two), three)
printfn "%A" tree
```

```
$ fsharpc --nologo discriminatedUnionTree.fsx
$ mono discriminatedUnionTree.exe
Node (Node (Leaf 1,Leaf 2),Leaf 3)
```

In this example we define a tree as depicted in Figure 14.1.

Pattern matching must be used in order to define functions on values of a discriminated union. E.g., in Listing 14.9 we define a function that traverses a tree and prints the content of the nodes.[1]

Listing 14.9 discriminatedUnionPatternMatching.fsx:
A discriminated union modelling binary trees.

```
type Tree = Leaf of int | Node of Tree * Tree
let rec traverse (t : Tree) : string =
    match t with
      Leaf(v) -> string v
      | Node(left, right) -> (traverse left) + ", " + (traverse right)

let tree = Node (Node (Leaf 1, Leaf 2), Leaf 3)
printfn "%A: %s" tree (traverse tree)
```

```
$ fsharpc --nologo discriminatedUnionPatternMatching.fsx
$ mono discriminatedUnionPatternMatching.exe
Node (Node (Leaf 1,Leaf 2),Leaf 3): 1, 2, 3
```

Discriminated unions are very powerful and can often be used instead of class hierarchies. Class hierarchies are discussed in Section 21.1.

14.4 Records

A record is a compound of named values, and a record type is defined as follows:

Listing 14.10 Syntax for defining record types.

```
[ <attributes> ]
type <ident> = {
  [ mutable ] <label1> : <type1>
  [ mutable ] <label2> : <type2>
  ...
}
```

Records are collections of named variables and values of possibly different types. They are reference types, and thus their content is stored on *The Heap*, see Section 6.8 for a discussion on reference types. Records can also be *struct records* using the [<Struct>] attribute, in which case they are value types. See Section 14.5 for a discussion on structs. An example of using records is given in Listing 14.11. The values of individual members of a record are obtained using the "." notation

· The Heap
· struct records
· .

[1] Jon: **Example uses pattern matching, which has yet to be introduced.**

Listing 14.11 records.fsx:
A record is defined for holding information about a person.

```
type person = {
    name : string
    age : int
    height : float
}

let author = {name = "Jon"; age = 50; height = 1.75}
printfn "%A\nname = %s" author author.name
```

```
$ fsharpc --nologo records.fsx && mono records.exe
{name = "Jon";
 age = 50;
 height = 1.75;}
name = Jon
```

The examples illustrate a how record type is defined to store varied data about a person, and how a value is created by a record expression defining its field values.

If two record types are defined with the same label set, then the latter dominates the former, and the compiler will at a binding infer that later. This is demonstrated in Listing 14.12.

Listing 14.12 recordsDominance.fsx:
Redefined types dominates old record types, but earlier definitions are still accessible using explicit or implicit specification for bindings.

```
type person = { name : string; age : int; height : float }
type teacher = { name : string; age : int; height : float }

let lecturer = {name = "Jon"; age = 50; height = 1.75}
printfn "%A : %A" lecturer (lecturer.GetType())
let author : person = {name = "Jon"; age = 50; height = 1.75}
printfn "%A : %A" author (author.GetType())
let father = {person.name = "Jon"; age = 50; height = 1.75}
printfn "%A : %A" author (author.GetType())
```

```
$ fsharpc --nologo recordsDominance.fsx && mono recordsDominance.exe
{name = "Jon";
 age = 50;
 height = 1.75;} : RecordsDominance+teacher
{name = "Jon";
 age = 50;
 height = 1.75;} : RecordsDominance+person
{name = "Jon";
 age = 50;
 height = 1.75;} : RecordsDominance+person
```

In the example, two identical record types are defined, and we use the built-in `GetType()` method to inspect the type of bindings. We see that `lecturer` is of `RecordsDominance+teacher` type, since `teacher` dominates the identical `author` type definition. However, we may enforce the `person` type by either specifying it for the name as in `let author : person = ...` or by fully or partially specifying it in the record expression following the "=" sign. In both cases, they are therefore of `RecordsDominance+author` type. The built-in `GetType()` method is inherited from the base class for all types, see Chapter 20 for a discussion on classes and inheritance.

Note that when creating a record, you must supply a value to all fields, and you cannot refer to other fields of the same record, e.g., `{name = "Jon"; age = height * 3; height = 1.75}` is illegal.

Since records are per default reference types, binding creates aliases not copies. This matters for mutable members, in which case when copying, we must explicitly create a new record with the old data. Copying can be done either by using referencing to the individual members of the source or using the short-hand *with* notation. This is demonstrated in Listing 14.13. · `with`

Listing 14.13 recordCopy.fsx:
Bindings are references. To copy and not make an alias, explicit copying must be performed.

```
1  type person = {
2    name : string;
3    mutable age : int;
4  }
5
6  let author = {name = "Jon"; age = 50}
7  let authorAlias = author
8  let authorCopy = {name = author.name; age = author.age}
9  let authorCopyAlt = {author with name = "Noj"}
10 author.age <- 51
11 printfn "author : %A" author
12 printfn "authorAlias : %A" authorAlias
13 printfn "authorCopy : %A" authorCopy
14 printfn "authorCopyAlt : %A" authorCopyAlt
```

```
1 $ fsharpc --nologo recordCopy.fsx && mono recordCopy.exe
2 author : {name = "Jon";
3  age = 51;}
4 authorAlias : {name = "Jon";
5  age = 51;}
6 authorCopy : {name = "Jon";
7  age = 50;}
8 authorCopyAlt : {name = "Noj";
9  age = 50;}
```

Here `age` is defined as a mutable value, and can be changed using the usual "`<-`" assignment operator. The example demonstrates two different ways to create records. Note that when the mutable value `author.age` is changed in line 10, then `authorAlias` also changes, since it is an alias of `author`, but neither `authorCopy` nor `authorCopyAlt` changes, since they are copies. As illustrated, copying using `with` allows for easy copying and partial updates of another record value.

14.5 Structures

Structures or *structs* for short have much in common with records. They specify a compound type · structures
with named fields, but they are value types, and they allow for some customization of what is to · structs
happen when a value of its type is created. Since they are value types, then they are best used for small amount of data. The syntax for defining struct types are:

Listing 14.14 Syntax for type abbreviation.

```
[ <attributes> ]
[<Struct>]
type <ident> =
  val [ mutable ] <label1> : <type1>
  val [ mutable ] <label2> : <type2>
  ...
  [new (<arg1>, <arg2>, ...) = {<label1> = <arg1>; <label1> = <arg2>; ...}
  [new (<arg1>, <arg2>, ...) = {<label1> = <arg1>; <label1> = <arg2>; ...}
  ...
```

The syntax makes use of the *`val`* and `new` keywords. Keyword `val` like `let` binds a name to a value, but unlike `let` the value is always the type's default value. The `new` keyword denotes the function used to fill values into the fields at time of creation. This function is called the *constructor*. No `let` nor `do` bindings are allowed in structure definitions. Fields are accessed using the "." notation. An example is given in Listing 14.15.

· `val`
·
· `new`
· constructor
· .

Listing 14.15 struct.fsx:
Defining a struct type and creating a value of it.

```
[<Struct>]
type position =
  val x : float
  val y : float
  new (a : float, b : float) = {x = a; y = b}

let p = position (3.0, 4.2)
printfn "%A: x = %A, y = %A" p p.x p.y
```

```
$ fsharpc --nologo struct.fsx && mono struct.exe
Struct+position: x = 3.0, y = 4.2
```

Structs are small versions of classes and allows, e.g., for overloading of the `new` constructor and for overriding of the inherited *`ToString()`* function. This is demonstrated in Listing 14.16.

· overload
· override
· `ToString()`

Listing 14.16 structOverloadNOverride.fsx:
Overloading the `new` constructor and overriding the default `ToString()` function.

```
[<Struct>]
type position =
  val x : float
  val y : float
  new (a : float, b : float) = {x = a; y = b}
  new (a : int, b : int) = {x = float a; y = float b}
  override this.ToString() =
    "(" + (string this.x) + ", " + (string this.y) + ")"

let pFloat = position (3.0, 4.2)
let pInt = position (3, 4)
printfn "%A and %A" pFloat pInt
```

```
$ fsharpc --nologo structOverloadNOverride.fsx
$ mono structOverloadNOverride.exe
(3, 4.2) and (3, 4)
```

We defer further discussion of these concepts to Chapter 20.

The use of structs are generally discouraged, and instead, it is recommended to use enums, records, and discriminated unions possibly with the `[<Struct>]` attribute for the last two in order to make them value types.

14.6 Variable types

An advanced topic in F# is *variable types*. There are three different versions of variable types in F#: *runtime resolved*, which has the syntax `'<ident>`, *anonymous*, which are written as "_", and *statically resolved*, which have the syntax `^<ident>`. Variable types are particularly useful for functions that work for many types. An example of a generic function and its use is given in Listing 14.17.

· variable types
· runtime resolved variable type
· anonymous variable type
· _
· statically resolved variable type

Listing 14.17 variableType.fsx:
A function `apply` with runtime resolved types.

```
let apply (f : 'a -> 'a -> 'a) (x : 'a) (y : 'a) : 'a = f x y
let intPlus (x : int) (y : int) : int = x + y
let floatPlus (x : float) (y : float) : float = x + y

printfn "%A %A" (apply intPlus 1 2) (apply floatPlus 1.0 2.0)
```

```
$ fsharpc --nologo variableType.fsx && mono variableType.exe
3 3.0
```

In this example, the function `apply` has runtime resolved variable type, and it accepts three parameters `f`, `x`, and `y`. The function will work as long as the parameters for `f` is a function of two parameters of identical type, and `x` and `y` are values of the same type. Thus, in the `printfn` statement, we are able to use `apply` for both an integer and a float variant.

The example in Listing 14.17 illustrates a very complicated way to add two numbers. And the "+" operator works for both types out of the box, so why not something simpler like relying on the F# type inference system by not explicitly specifying types as attempted in Listing 14.18.

Listing 14.18 variableTypeError.fsx:
Even though the "+" operator is defined for both integers and floats, the type inference is static and infers `plus : int -> int`.

```
let plus x y = x + y

printfn "%A %A" (plus 1 2) (plus 1.0 2.0)
```

```
$ fsharpc --nologo variableTypeError.fsx && mono variableTypeError.exe

variableTypeError.fsx(3,34): error FS0001: This expression was expected
   to have type
     'int'
but here has type
     'float'

variableTypeError.fsx(3,38): error FS0001: This expression was expected
   to have type
     'int'
but here has type
     'float'
```

Unfortunately, the example fails to compile, since the type inference is performed at compile time, and by `plus 1 2`, it is inferred that `plus : int -> int`. Hence, calling `plus 1.0 2.0` is a type error. Function bindings allow for the use of the *`inline`* keyword, and adding this successfully reuses the definition of `plus` for both types as shown in Listing 14.19.

· `inline`

Listing 14.19 variableTypeInline.fsx:
The keyword `inline` forces static and independent inference each place the function is used. Compare to the error case in Listing 14.18.

```
let inline plus x y = x + y

printfn "%A %A" (plus 1 2) (plus 1.0 2.0)
```

```
$ fsharpc --nologo variableTypeInline.fsx && mono variableTypeInline.exe
3 3.0
```

In the example, adding the `inline` does two things: Firstly, it copies the code to be performed to each place, the function is used, and secondly, it forces statically resolved variable type checking independently in each place. The type annotations inferred as a result of the `inline`-keyword may be written explicitly as shown in Listing 14.20.

Listing 14.20 compiletimeVariableType.fsx:
Explicitely spelling out of the statically resolved type variables from Listing 14.18.

```
let inline plus (x : ^a) (y : ^a) : ^a when ^a : (static member ( + ) :
   ^a * ^a -> ^a) = x + y

printfn "%A %A" (plus 1 2) (plus 1.0 2.0)
```

```
$ fsharpc --nologo compiletimeVariableType.fsx
$ mono compiletimeVariableType.exe
3 3.0
```

The example in Listing 14.20 demonstrates the statically resolved variable type syntax, `^<ident>`, as well as the use of *type constraints* using the keyword *`when`*. Type constraints have a rich syntax, but will not be discussed further in this book.[2] In the example, the type constraint `when ^a : (static member ( + ) :  ^a * ^a -> ^a)` is given using the object oriented properties of the type variable `^a`, meaning that the only acceptable type values are those, which have a member function `(+)` taking a tuple and giving a value all of identical type, and which where the type can be inferred at compile time. See Chapter 20 for details on member functions.

· type constraints
· `when`

The `inline` construction is useful when generating generic functions and still profiting from static type checking. However, explicit copying of functions is often something better left to the compiler to optimize over. An alternative seems to be using runtime resolved variable types with the `'<ident>` syntax. Unfortunately, this is not possible in case of most operators, since they have been defined in the `FSharp.Core` namespace to be statically resolved variable type. E.g., the "+" operator has type `( + ) : ^T1 -> ^T2 -> ^T3 (requires ^T1 with static member (+) and ^T2 with static member (+))`.

[2] Jon: **Should I extend on type constraints? Perhaps it is better left for a specialize chapter on generic functions.**

15 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 Section 6.1, its extension with pattern matching is given as,

· let

Listing 15.1 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 15.2.

Listing 15.2 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 alternative to `if` – `then` – `else` expressions particularly when parsing input for a function. Consider the example in Listing 15.3.

Listing 15.3 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 can 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, is a discriminated union and a function defined. The funciton converts each case to a supporting statement using an `if`-expression. The same can be done with the *match* − *with* expression and patterns as is demonstrated in Listing 15.4.

· `match`
· `with`

Listing 15.4 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 15.5 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# report an error, when not the complete domain of the input pattern is covered by cases in `match`-expressions.

· input pattern

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

Listing 15.6 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 15.7.

Listing 15.7 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 *anonymous functions*. Patterns for anonymous functions have the syntax, · anonymous functions

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 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 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 functions

Patterns are also used in exceptions to be discussed in Section 18.1, and in conjunction with the

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

function-keyword, a keyword we discourage in this book. We will now demonstrate a list of important patterns in F#.

15.1 Wildcard pattern

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

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 pattern

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 is 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.

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..."
```

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 patterns 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 15.5.

· 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 processing 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 pattern
· record pattern
· discriminated union patterns

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

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 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 disjunctively using the "|" lexeme and conjunctively using the "&" lexeme. *Disjunctive patterns* combine as fall-through as illustrated in Listing 15.19.

· |
· &
· disjunctive pattern

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 patterns

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 used 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 for active patterns is one of,

· active patterns

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
```

```
$ fsharpc --nologo activePattern.fsx && mono activePattern.exe
{x = 2.0;
 y = 3.0;}:
 Cartesian (2.0, 3.0)
{x = 2.0;
 y = 3.0;}:
 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 15.5. 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` 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.

```
1  type vec = {x : float; y : float}
2  let (|Polar|) (o : vec) (v : vec) =
3    let x = v.x - o.x
4    let y = v.y - o.y
5    (sqrt(x*x + y * y), atan2 y x)
6  let printPolar (o : vec) (p : vec) : unit =
7      match p with
8        Polar o (a, d) -> printfn "%A:\n Cartesian (%A, %A)" p a d
9
10 let v = {x = 2.0; y = 3.0}
11 let offset = {x = 1.0; y = 1.0}
12 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.

```
1  let (|Div|_|) (e,d) = if d <> 0.0 then Some (e/d) else None
2
3  let safeDiv (p : float * float) =
4      match p with
5        | (0.0, 0.0) -> printfn "Div %A = undefined" p
6        | Div res -> printfn "Div %A = %A" p res
7        | _ -> printfn "Div %A = infinity" p
8
9  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 work similarly to discriminated unions without arguments.[2] 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]
```

```
$ fsharpc --nologo activeMultiCasePattern.fsx
$ mono activeMultiCasePattern.exe
0: Its gold!
1: Its silver.
2: Its no more than bronze.
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 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 15.26.

· static type pattern
· :

[2] Jon: **This maybe too advanced for this book.**

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 pattern
· :?

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 analyzing 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