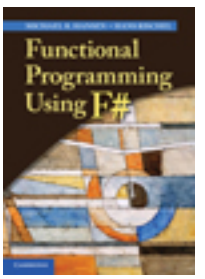


Functional Programming Using F#

Michael R. Hansen, Hans Rischel

Book DOI: http://dx.doi.org/10.1017/CBO9781139093996

Online ISBN: 9781139093996

Hardback ISBN: 9781107019027

Paperback ISBN: 9781107684065

Chapter

7 - Modules pp. 149-174

Chapter DOI: http://dx.doi.org/10.1017/CBO9781139093996.008

Cambridge University Press

7

Modules

Throughout the book we have used programs from the F# core library and from the .NET library, and we have seen that programs from these libraries are reused in many different applications. In this chapter we show how the user can make own libraries by means of modules consisting of signature and implementation files. The implementation file contains the declarations of the entities in the library while the signature file specifies the user's interface to the library.

Overloaded operators are defined by adding augmentations to type definitions. Type augmentations can also be used to customize the equality and comparison operators and the `string` conversion function. Libraries with polymorphic types are obtained by using signatures containing type variables.

These features of the module system are illustrated by small examples: plane geometric vectors and queues of values with arbitrary type. The last part of the chapter illustrates the module system by a larger example of piecewise linear plane curves. The curve library is used to describe the recursively defined family of Hilbert curves, and these curves are shown in a window using the .NET library. The theme of an exercise is a picture library used to describe families of recursively defined pictures like Escher's fishes.

7.1 Abstractions

A key concept in designing a good program library is *abstraction*: the library must provide a service where a user can get a general understanding of *what* a library function is doing without being forced to learn details about *how* this function is implemented. The interface to any standard library, like, for example, the `Set` library, is based on useful abstractions, in this case the (mathematical) concept of a set. Based on a general understanding of this concept you may use the `Set` library while still being able to focus your main attention on other aspects of your program.

Modules are the technical means of dividing a programming problem into smaller parts, but this process must be guided by useful abstractions. Creation of useful abstractions is the basis for obtaining a successful modular program design.

Abstractions are described on the *semantical* level by:

- Description of a collection of entities to be *represented* by an F# type.
- Description of specific values and computations to be *implemented* by F# values and functions.

The above example of the `Set` library comes with a natural language explanation of the concept of sets and of operations on sets plus certain special sets like the empty set and singleton sets (cf. Section 5.2). Throughout the book we follow this style in describing the semantics, as seen from a user, of the library in two parts:

- A general, narrative description of the conceptual framework.
- A precise, but short description of the *intended working* of each type, value and function.

Parts of this description may be included as comments in the signature file.

Abstractions are described on the *syntactical* level by *specifications*. The main forms are:

```
type TypeName ...
val Name : type
```

They are part of an F# program and found in the *signature* file of a module. They specify entities to be *represented* and *implemented* in the library.

A type specification without type expression

```
type TypeName
```

hides the structure of the type from the user and this structure is then only found in the implementation of the library. The user's access to the library is restricted to use values and functions according to the specifications in the signature and the user cannot look into details of the representation of values of this type or make such values "by hand." This feature of the interface to a library is called *data hiding*. It gives the means for protecting the integrity of representations of values such that, for example, invariants are not violated.

7.2 Signature and implementation

An F# *module* consists of F# source files that are compiled together to make a *library*. This library can then later be used by other programs. The first file in the compilation is a *signature* file containing *specifications* of the user's interface to the resulting program library while the other consists of F# declarations to define the *implementation* of this interface. Signature and implementation files are text files edited by the programmer while the library file is produced by the F# compiler when signature and implementation are compiled together.

The compilation of a pair of signature and implementation requires that the implementation *matches* the signature: each *specification* in the signature should be *implemented* by a corresponding *declaration* in the implementation file. A declaration in the implementation may have a more general type than specified in the signature but the type is then specialized to the type in the signature. Declarations of entities that do not appear in the signature are *local* to the implementation and not visible to a user of the library.

Example: vectors in the plane

Consider the example of vectors in the plane in Section 3.3. We want to make a vector library with hidden `Vector` type. Besides the functions in the earlier version of this example we have to provide a function `make` to make values of type `Vector` and a function `coord` to inspect a value of this type because the user would otherwise not be able to make or inspect any such value.

The signature should hence contain a specification of a (hidden) type `Vector` without type expression:

```
type Vector
```

together with specifications of the functions. This gives the following signature of a module with *module name* `Vector`:

```
module Vector                              // Vector signature
type Vector
val ( ~-. ) : Vector -> Vector             // Vector sign change
val ( +. )  : Vector -> Vector -> Vector // Vector sum
val ( -. )  : Vector -> Vector -> Vector // Vector difference
val ( *. )  : float  -> Vector -> Vector // Product wth number
val ( &. )  : Vector -> Vector -> float  // Dot product
val norm    : Vector -> float              // Length of vector
val make    : float * float -> Vector      // Make vector
val coord   : Vector -> float * float      // Get coordinates
```

The implementation must contain a definition of the type `Vector` and declarations of all values specified in the signature. The type `Vector` specified as hidden in the signature must be a tagged type or a record type, so we have to add a tag, say `V`, in the type definition:

```
module Vector                          // Vector implementation
type Vector = V of float * float
let (~-.) (V(x,y))              = V(-x,-y)
let (+.) (V(x1,y1)) (V(x2,y2)) = V(x1+x2,y1+y2)
let (-.)  v1         v2         = v1 +. -. v2
let ( *.) a          (V(x1,y1)) = V(a*x1,a*y1)
let (&.) (V(x1,y1)) (V(x2,y2)) = x1*x2 + y1*y2
let norm  (V(x1,y1))           = sqrt(x1*x1+y1*y1)
let make  (x,y)                = V(x,y)
let coord (V(x,y))             = (x,y)
```

The resulting library can be used as follows:

```
open Vector;;
let a = make(1.0,-2.0);;
val a : Vector

let b = make(3.0,4.0);;
val b : Vector
```

```
let c = 2.0 *. a -. b;;
val c : Vector

coord c;;
val it : float * float = (-1.0, -8.0)

let d = c &. a;;
val d : float = 15.0

let e = norm b;;
val e : float = 5.0
```

Note that the response from the system displays only the type name and no value when an expression of type `Vector` is entered: the structure of the type is hidden from the user.

Remark: The above method of defining infix operators in a library is *not recommended*. One should instead use the syntax described in Section 7.3. The reason is that the operators in this example are only available in a program using the library when the library has been *opened*, in our case by "`open Vector`". One would normally prefer *not* to open the library in order to limit the "pollution of the namespace". Entities in the library are then accessed using composite names like `Vector.make`, and the operators will then only be available in the inconvenient prefix form like `Vector.(+.)` and the elegance of the infix operators is not available. It is even worse if we, for instance, declare an operator + on vectors using the above syntax. This declaration will then override the existing declaration of + when the module is opened, and the + operator on numbers will no longer be available.

The `module`-declaration in signature and implementation can use a *composite* module name, for example:

```
module MyLibrary.Vector
```

A program may then access the library in any of the following ways:

```
open MyLibrary.Vector;;
let a = make(1.0,-2.0);;
```

or

```
open MyLibrary;;
let a = Vector.make(1.0,-2.0);;
```

or

```
let a = Mylibrary.Vector.make(1.0,-2.0);;
```

Files and compilation

The F# module system uses the file types `fsi`, `fs` and `dll` as follows:

FileName`.fsi`	F# signature file	Text file edited by programmer
FileName`.fs`	F# implementation file	Text file edited by programmer
FileName`.dll`	Library file	Binary output file from F# compiler

The *separate compilation* of a module is supported by any of the development platforms for F#. Using the batch compiler `fsc` (cf. Section 1.10) the compilation of a library with signature file *SignatureName*`.fsi` and implementation file *LibraryName*`.fs` is made by the command:

`fsc -a` *Signature*`.fsi` *Library*`.fs`

This compilation produces a library file:

Library`.dll`

The library gets the same file name as the implementation file – with a different file type – while the signature can have a different file name. These file names should not be confused with the module name introduced by the `module`-declarations in signature and implementation files as `Vector` in the above example.

Compilation of a program using the library *Library*`.dll` is made by a command like:

`fsc ... -r` *Library*`.dll ...`

Compilation of library and program should use the *same version* of F# and .NET.

A library *Library*`.dll` may be used as follows from an interactive environment:

1. Move the library file *Library*`.dll` to a special directory, for example:
   `c:\Documents and Settings\FsharpBook\lib`
2. Refer to the library from the interactive environment using an `#r` directive:
   `#r @"c:\Documents and Settings\FsharpBook\lib\`*Library*`.dll";;`

7.3 Type augmentation. Operators in modules

A *type augmentation* adds declarations to the definition of a tagged type or a record type and it allows declaration of (overloaded) operators. The type augmentation uses an OO-flavoured syntax. This is illustrated in the signature and implementation files in Tables 7.1 and 7.2. The operators +, − and ⋆ on numbers are overloaded to denote also operations on values of type `Vector` and the operator ⋆ is even overloaded to denote two different operations on vectors. Resolving the overloading is made using the types of the operands, so the two versions of the operator ⋆ could not have identical operand types.

```
module Vector
[<Sealed>]
type Vector =
    static member ( ~- ) : Vector -> Vector
    static member ( + )  : Vector * Vector -> Vector
    static member ( - )  : Vector * Vector -> Vector
    static member ( * )  : float  * Vector -> Vector
    static member ( * )  : Vector * Vector -> float
val make : float * float -> Vector
val coord: Vector -> float * float
val norm : Vector -> float
```

Table 7.1 *Signature file with type augmentation*

```
module Vector
type Vector =
     | V of float * float
     static member (~-) (V(x,y))           = V(-x,-y)
     static member (+) (V(x1,y1),V(x2,y2)) = V(x1+x2,y1+y2)
     static member (-) (V(x1,y1),V(x2,y2)) = V(x1-x2,y1-y2)
     static member (*) (a, V(x,y))         = V(a*x,a*y)
     static member (*) (V(x1,y1),V(x2,y2)) = x1*x2 + y1*y2
let make(x,y)     = V(x,y)
let coord(V(x,y)) = (x,y)
let norm(V(x,y))  = sqrt(x*x + y*y)
```

Table 7.2 *Implementation file with type augmentation*

The `member` declarations cannot be intermixed with `let` declarations (but local `let` declarations are, of course, allowed inside the expressions in the declarations).

The functions `make`, `coord` and `norm` are specified and implemented as usual F# functions – the OO-features should only be used to obtain an effect (here: operators) that cannot be obtained using normal F# style.

Note that the implementation file in Table 7.2 compiles without signature file if the `module` declaration is commented out. This is often convenient during the implementation and test of a module. Furthermore, the output from the interactive F# compiler in such a compilation can be useful in getting details in the signature correct.

Note the following:

- The *attribute* `[<Sealed>]` is mandatory when a type augmentation is used. It protects the library against unintentional use in the OO-world.
- The heading "`|`" in the type and the declarations in the augmentation must be at the same indentation level.
- The "`member`" specification and declaration of an infix operator (e.g. +) corresponds to a type of form $type_1 * type_2 \text{ -> } type_3$, while the earlier `val` specification and `let` declaration (of e.g. `(+.)`) on Page 151 use a higher-order type of form $type_1 \text{ -> } type_2 \text{ -> } type_3$. The indicated type is required to get the overloading. The resulting prefix function (like e.g. `(+)`) will, nevertheless, get the usual higher-order type.
- The operators +, − and $*$ are available on vectors without opening the library. The operators can still be used on numbers.
- The functions `make`, `coord` and `norm` could have been declared as "static members" using the OO notation. The usual F# form is more succinct and should be used whenever possible.

The following are examples of use of the `Vector` library specified in Table 7.1:

```
let a = Vector.make(1.0,-2.0);;
val a : Vector.Vector

let b = Vector.make(3.0,4.0);;
val b : Vector.Vector
```

```
let c = 2.0 * a - b;;
val c : Vector.Vector

Vector.coord c;;
val it : float * float = (-1.0, -8.0)

let d = c * a;;
val d : float = 15.0

let e = Vector.norm b;;
val e : float = 5.0

let g = (+) a b;;
val g : Vector.Vector

Vector.coord g;;
val it : float * float = (4.0, 2.0)
```

7.4 Type extension

The implementation in Table 7.2 can instead be made using a *type extension*;

```
type ... with ...
```

as shown in Table 7.3. This implementation compiles with the signature in Table 7.1 and has the same effect as the implementation in Table 7.2, but it offers the possibility of inserting usual function declarations between the type definition and the `member` declarations like `make` and `coord` in Table 7.3. Such functions can be used in the `member` declarations and that may sometime allow simplifications. This possibility is used later in the example of plane curves in Section 7.9.

```
module Vector
type Vector = V of float * float
let make(x,y)     = V(x,y)
let coord(V(x,y)) = (x,y)
type Vector with
     static member (~-) (V(x,y))           = V(-x,-y)
     static member (+) (V(x1,y1),V(x2,y2)) = V(x1+x2,y1+y2)
     static member (-) (V(x1,y1),V(x2,y2)) = V(x1-x2,y1-y2)
     static member (*) (a, V(x,y))         = V(a*x,a*y)
     static member (*) (V(x1,y1),V(x2,y2)) = x1*x2 + y1*y2
let norm(V(x,y))  = sqrt(x*x + y*y)
```

Table 7.3 *Implementation module with type extension*

7.5 Classes and objects

There are full features for Object-oriented (OO) programming in F#, but this is not a major theme of this book. We just give a brief introduction to cover the topics that are needed when using the .NET library in an F# program and when making computational expressions as described in a later chapter. The OO-features in F# are only used on a larger scale when implementing applications that can be used from programs made in another .NET language.

A *class* definition looks syntactically like an augmented type definition where the type expression has been removed and replaced by declarations of *constructor* functions. A class in F# determines a type and a value of such a type is called an *object*. An object is obtained by calling a constructor of the class. The call of a constructor is often preceded by the keyword `new`. We illustrate classes and objects by an OO-version of our vector example:

```
type ObjVector(X: float, Y: float) =
    member v.x = X
    member v.y = Y
    member v.coord() = (v.x, v.y)
    member v.norm() = sqrt(v.x * v.x + v.y * v.y)
    static member (~-) (v: ObjVector) = ObjVector(- v.x,- v.y)
    static member (+)  (v1: ObjVector, v2:ObjVector)
        = ObjVector(v1.x + v2.x, v1.y + v2.y)
    static member (-)  (v1: ObjVector, v2:ObjVector)
        = ObjVector(v1.x - v2.x, v1.y - v2.y)
    static member (*) (a,v:ObjVector) = ObjVector(a*v.x,a*v.y)
    static member (*)  (v1: ObjVector, v2:ObjVector)
        = v1.x * v2.x + v1.y * v2.y
```

The constructor `ObjVector` initializes the members `x` and `y` using the parameter values `X` and `Y`. The following show some uses of the class:

```
let a = ObjVector(1.0,-2.0);;
val a : CbjVector

let b = ObjVector(Y = 4.0, X = 3.0);;   // Named arguments
val b : ObjVector

b.coord();;
val it : float * float = (3.0, 4.0)

let c = 2.0 * a - b;;
val c : ObjVector

c.coord();;
val it : float * float = (-1.0, -8.0)

b.x;;
val it : float = 3.0
```

```
let d = c * a;;
val d : float = 15.0

let e = b.norm();;
val e : float = 5.0

let g = (+) a b;;
val g : ObjVector

g.coord();;
val it : float * float = (4.0,2.0)
```

The above examples

```
let b = ObjVector (Y=4.0, X=3.0);;
```

illustrates the use of *named arguments* where arguments in a function call are identified by name instead of position in the argument list. Named arguments can make call of functions from the .NET library more readable as the meaning of each argument is visible from the context, while the meaning of an argument can otherwise only be found by studying the documentation of the function in question.

The example of plane curves uses a similar feature called *optional property setting* (cf. Section 7.9).

Note that members `coord`, `norm` and `x` are written as a suffix to the values `c` and `b`. They are in the OO-world considered as belonging to the values `c` and `b`. Using `fun`-expressions they determine functions

```
fun v -> v.coord()
fun v -> v.norm()
fun v -> v.x
```

where, for example:

```
(fun v -> v.coord()) c = c.coord()
```

Using OO-style constructs is daily life for the F# programmer as the .NET library is 100 percent OO, and the OO-features of F# give a quite streamlined access to this library. An object member is used as argument of a higher-order function by packaging it into a `fun`-expression as shown above.

7.6 Parameterized modules. Type variables in signatures

A module in F# can be *parameterized* by type variables and may thereby implement polymorphic types, values and functions. This is illustrated by the example of a *queue*: A queue is a row of values of the same type. The `put` function inserts a new value at the rear end of the queue while the `get` function gets the front element. An exception should be raised if `get` is attempted on an empty queue. This idea is specified in the signature in Table 7.4.

The implementation uses an interesting data representation due to L.C. Paulson (cf. [10], Chapter 7) where a queue is represented by two lists, a `front` list containing the first queue elements in the order of insertion and a `rear` list containing the remaining queue elements in the reverse order of insertion. The representation of a queue containing values 1, 2, 3 may hence look as follows:

```
front  [1]
rear   [3; 2]
```

Using `put` to insert a value, say 4, will simply "cons" the value onto the `rear` list:

```
front  [1]
rear   [4; 3; 2]
```

while `get` removes the heading element 1 from the `front` list:

```
front  []
rear   [4; 3; 2]
```

A call of `get` in this situation with empty `front` list will *reverse* the `rear` list to get the list `[2; 3; 4]` with the queue elements in the order of insertion. This list is then used as `front` list while the `rear` list becomes empty.

```
front  [3; 4]        (the front element 2 has been removed by get)
rear   []
```

The implementation module in Table 7.5 uses this idea and represents a `Queue` value as a record `{front:'a list; rear:'a list}` containing the two lists. Note that the representation of a queue is not *unique* because *different* pairs of front and rear lists may represent the *same* queue.

```
module Queue
type Queue<'a>
val empty : Queue<'a>
val put   : 'a -> Queue<'a> -> Queue<'a>
val get   : Queue<'a> -> 'a * Queue<'a>
exception EmptyQueue
```

Table 7.4 *Signature of parameterized* `Queue` *module*

```
module Queue
exception EmptyQueue
type Queue<'a> = {front: 'a list; rear: 'a list}
let empty = {front = []; rear = []}
let put y {front = xs; rear = ys} = {front = xs; rear = y::ys}
let rec get = function
              | {front = x::xs; rear = ys} ->
                    (x,{front = xs; rear = ys})
              | {front = []; rear = []} -> raise EmptyQueue
              | {front = []; rear = ys} ->
                    get {front = List.rev ys; rear = []}
```

Table 7.5 *Implementation of parameterized* `Queue` *module*

The Queue library can be used as follows:

```
let q0 = Queue.empty: Queue.Queue<int>;;
val q0 : Queue.Queue<int>

let q1 = Queue.put 1 q0;;
val q1 : Queue.Queue<int>

let q2 = Queue.put 2 q1;;
val q2 : Queue.Queue<int>

let (x,q3) = Queue.get q2;;
val x : int = 1
val q3 : Queue.Queue<int>

let q4 = Queue.put 4 q3;;
val q4 : Queue.Queue<int>

let (x2,q5) = Queue.get q4;;
val x2 : int = 2
val q5 : Queue.Queue<int>

let (x3,q6) = Queue.get q5;;
val x3 : int = 4
val q6 : Queue.Queue<int>
```

7.7 Customizing equality, hashing and the `string` function

The F# compiler will automatically generate a default equality operator for the above type `Queue<'a>` whenever the type variable `'a` is instantiated with an equality type. This default equality operator is, however, *not* the wanted operator because it distinguishes values that we want to consider equal. We may for instance get a queue containing the single integer `2` in two ways: as the above queue value `q3` where the integers `1` and `2` are put into the empty queue `q0` followed by a `get` to remove the integer `1`, or as the below queue value `qnew` where we just put the integer `2` into the empty queue `q0`. These two values are considered different by the default equality operator:

```
let qnew = Queue.put 2 q0 ;;
val qnew : Queue.Queue<int>
qnew = q3;;
val it : bool = false
```

The reason is that the queues `qnew` and `q3` are *represented* by *different* values of type `Queue`:

The value of	`qnew`	is represented by	`{front = []; rear = [2]}`
The value of	`q3`	is represented by	`{front = [2]; rear = []}`

and the default equality operator is based on *structural* equality of the representing values. Hence, `qnew` and `q3` are considered different by this operator.

```
module Queue
exception EmptyQueue
[<CustomEquality;NoComparison>]
type Queue<'a when 'a : equality> =
     {front: 'a list; rear: 'a list}
     member q.list() = q.front @ (List.rev q.rear)
     override q1.Equals qobj =
         match qobj with
         | :? Queue<'a> as q2 -> q1.list() = q2.list()
         | _ -> false
     override q.GetHashCode() = hash (q.list())
     override q.ToString() = string (q.list())
```

Declarations of `empty`, `put` and `get` are as in Table 7.5.
In signature: `type Queue<'a when 'a : equality>`

Table 7.6 *Type definition with augmentation for equality, hashing and* `string`

It is possible to *override* the default equality operator using a type augmentation as shown in Table 7.6. The signature in Table 7.4 needs an equality constraint on the type variable `'a` of queue elements as the `Equals` function uses equality for `'a list` values:

```
type Queue<'a when 'a : equality>
```

The signature can otherwise be used unchanged.

The `Equals` function contains the clause:

```
:? Queue<'a> as q2 ->...
```

It expresses a match on *type*. The value of `qobj` matches the pattern if the type of `qobj` matches the type `Queue<'a>` in the pattern, that is, if the type of `qobj` is an instance of this type.The identifier `q2` is then bound to the value of `qobj`.

Note the following:

- The customized equality compares single lists containing all queue elements. This list `q.list()` is obtained from the used representation `{front=xs; rear=ys}` of a queue as the `front` list `q.front` with the reversed of the `rear` list `q.rear` appended.
- The overriding cannot be given in a separate type extension. There are hence no possibility of declaring a local function to be used in the `member`-declarations. The frequently used expression `q.front @ (List.rev q.rear)` is therefore defined as a `member` function `q.list()`.
- The compiler gives a warning if the hash function is not customized because values considered equal should have same hash code. This condition becomes critical if the imperative collections `HashSet` or `Directory` (cf. Section 8.11) are used with elements of type `Queue`.
- Overriding `ToString` gives a reasonable conversion of a queue to a string by using `string` on `q.list()`.

Applying the new `Queue` module with customized comparison and `string` function to the example in Section 7.6 with declarations of `q0`,`q1`,...,`q6` and `s` we now get:

```
qnew = q3;;
val it : bool = true

string q2;;
val it : string = "[1; 2]"
```

7.8 Customizing ordering and indexing

Using a suitable type augmentation one may also customize the *ordering:* $q_1 < q_2$ and *indexing:* `q.[n]` on values of a defined type. The corresponding type augmentation in the queue example is shown in Table 7.7.

The ordering is declared by overriding the `CompareTo` method in the `IComparable` interface. The implemented comparison uses `compare` on the lists of the queue elements in insertion order. The signature must tell that this interface is used:

```
interface System.IComparable
```

The indexing is expressed by the `get` part of an `Item` member function. The implementation uses list indexing in the list of queue elements in insertion order. The signature must contain the corresponding specification:

```
member Item : int -> 'a with get
```

```
[<Sealed>]
type Queue<'a when 'a : comparison> =
     interface System.IComparable
     member Item : int -> 'a with get
```

Signature of Queue with ordering and indexing: type part

```
[<CustomEquality;CustomComparison>]
type Queue<'a when 'a : comparison> =
     {front: 'a list; rear: 'a list}
     member q.list() = q.front @ (List.rev q.rear)
     interface System.IComparable with
       member q1.CompareTo qobj =
         match qobj with
         | :? Queue<'a> as q2 -> compare (q1.list()) (q2.list())
         | _ ->
           invalidArg "qobj"
                      "cannot compare values of different types"
     member q.Item
       with get n = (q.list()).[n]
```

Implementation of Queue with ordering and indexing

Note: Equality and hashing as in Table 7.6 are also needed

Table 7.7 *Type augmentation for ordering and indexing in queue module*

The following illustrates uses of ordering and indexing:

```
let q0 = Queue.empty;;
let q1 = Queue.put 1 q0;;
let q2 = Queue.put 2 q1;;

q2 > q1 ;;
val it : bool = true

q2.[1] ;;
val it : int = 2
```

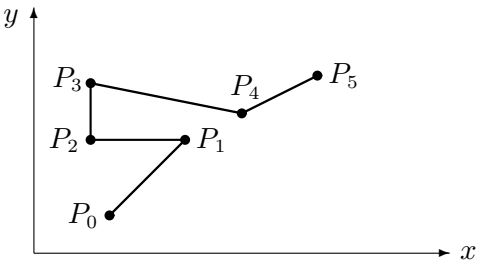
7.9 Example: Piecewise linear plane curves

In this example we consider piecewise linear curves in the plane following an idea due to Fokkinga (cf. [4]). Such a curve consists of a point P_0 and a (possible empty) sequence of line segments $P_0P_1,\ P_1P_2,\ \ldots,\ P_{n-2}P_{n-1}$ where $P_0, P_1, \ldots, P_{n-1}$ are points in the plane. The point P_0 is called the *start point* of the curve while P_{n-1} is called the *end point*. We use usual rectangular, Cartesian coordinates in the plane, so points and vectors in the plane correspond to coordinates that are pairs of `float` numbers.

We want represent a curve by a F# value and to implement the operations on curves shown in Table 7.8. A corresponding signature is shown in Table 7.9. Note that the user of the library can understand and use the functions while thinking purely in geometrical terms, so we have obtained the wanted abstraction.

Syntax	Function
`point`(x, y)	The curve consisting of the single point with coordinates (x, y)
c_1 + c_2	The curve consisting of the curve c_1, the segment from the end point of c_1 to the start point of c_2 and the curve c_2.
a * c	The curve obtained from c by multiplication with factor a from the start point of c
c |^ a	The curve obtained by rotating c the angle a (in degrees) around its start point
c --> (x, y)	The curve obtained from c by the parallel translation in the plane moving the start point of c to the point with coordinates (x, y)
c >< a	The curve obtained from c by horizontal reflection in the vertical line with equation $x = a$
`verticRefl` c b	The curve obtained from c by vertical reflection in the horizontal line with equation $y = b$
`boundingBox` c	The pair $((x_{min}, y_{min}), (x_{max}, y_{max}))$ of coordinates of lower left and upper right corner of the bounding box of the curve c
`width` c	The width of the bounding box of c
`height` c	The height of the bounding box of c
`toList` c	The list `[`(x_0, y_0)`;` (x_1, y_1)`;`$\ldots(x_{n-1}, y_{n-1})$`]` of coordinates of the curve points $P_0, P_1; \ldots; P_{n-1}$

Table 7.8 *Operations on curves*



Use of the infix operators `|` `^` for the rotate function is overloaded to also allow integer angle values. The infix operators allow `Curve` expressions to be written using a minimum of parentheses.

```
module Curve
[<Sealed>]
type Curve =
    static member ( + ) : Curve * Curve -> Curve
    static member ( * ) : float * Curve -> Curve
    static member ( |^) : Curve * float -> Curve
    static member ( |^) : Curve * int   -> Curve
    static member (-->) : Curve * (float * float) -> Curve
    static member (><)  : Curve * float -> Curve
val point       : float * float -> Curve
val verticRefl  : Curve -> float -> Curve
val boundingBox : Curve -> (float * float) * (float * float)
val width       : Curve -> float
val height      : Curve -> float
val toList      : Curve -> (float * float) list
```

Table 7.9 *Signature of* `Curve` *library*

We present an application of the `Curve` library before presenting its implementation.

Example: Hilbert curves

The Hilbert curves $h_0, h_1, h_2, ...$ form a system of curves, where h_0 consists of the point with coordinates $(0, 0)$ while each curve h_{n+1} is obtained by joining four curves c_1, c_2, c_3, c_4 obtained from h_n by transformations composed of reflections, rotations and translations. The Hilbert curves h_0, h_1, h_2 and h_3 are shown in Figure 7.1. All Hilbert curves start in the origin and the connecting segments (dotted lines in the figure) are of length 1.

We want to declare a function

```
hilbert: Curve.Curve -> Curve.Curve
```

such that

$$h_{n+1} = \texttt{hilbert}\ h_n \quad \text{for } n = 0, 1, 2, \ldots$$

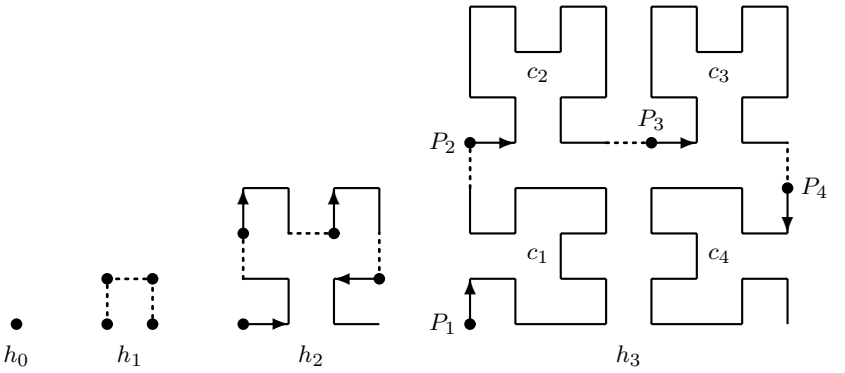
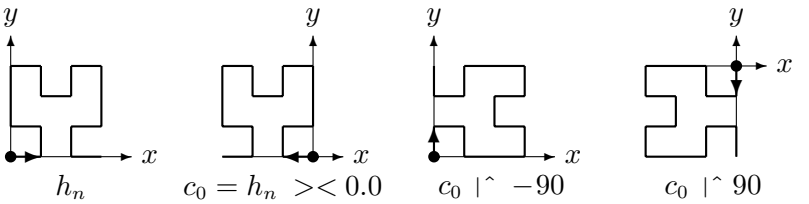


Figure 7.1 Hilbert curves

Studying Figure 7.1 we note that c_2 and c_3 can be obtained from h_n by parallel translations while c_1 and c_4 must be obtained from a mirror image of h_n. The following figure shows the curve c_0 obtained by horizontal reflection of h_n in the vertical line through the start point $(0.0, 0.0)$ and the curves obtained from c_0 by rotations through $-90°$ and $90°$:



Using the width and height of h_n:

```
w = Curve.width hn
h = Curve.height hn
```

we can express the coordinates of the start points P_1, P_2, P_3, P_4 of c_1, c_2, c_3, c_4:

```
P1 : (0.0, 0.0)
P2 : (0.0, w + 1.0)
P3 : (h + 1.0, w + 1.0)
P4 : (h + h + 1.0, w)
```

Note that the height and width of c_1 and c_4 are the width and height, respectively, of h_n. The height of h_n is actually equal to its width.

These considerations leads to the wanted declaration:

```
let h0 = Curve.point (0.0,0.0);;
val h0 : Curve.Curve
```

```
let hilbert hn =
    let w  = Curve.width hn
    let h  = Curve.height hn
    let c0 = hn >< 0.0
    let c1 = c0 |^ -90
    let c2 = hn --> (0.0, w + 1.0)
    let c3 = hn --> (h + 1.0, w + 1.0)
    let c4 = (c0 |^ 90) --> (h + h + 1.0, w)
    c1 + c2 + c3 + c4;;
val hilbert : Curve.Curve -> Curve.Curve
```

Note that the programming of the `hilbert` function has been done using geometric concepts only. We do not need any knowledge about the implementation of the `Curve` library.

Displaying curves

We want to make a function to display a curve in a window using the .NET library. Before getting to the programming we have to make some geometric considerations.

The display is made in a *panel* belonging to a window. The panel uses Cartesian coordinates where the y-axis points *downwards* and the upper left corner of the panel has panel coordinates $(0, 0)$. The situation is depicted in Figure 7.2. The thick box is the panel with width `pw` and height `ph`. The picture shows that a curve point with coordinates (x, y) has panel coordinates:

$$\begin{aligned} x_{panel} &= x \\ y_{panel} &= ph - y \end{aligned} \qquad (*)$$

The program uses two libraries:

`System.Windows.Forms` containing facilities to set up a *window* with scroll-bars and underlying *panel* to contain the drawing

`System.Drawing` containing facilities to *draw* the curve in the panel

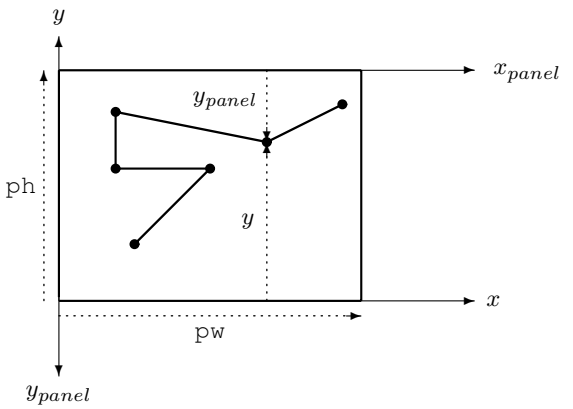


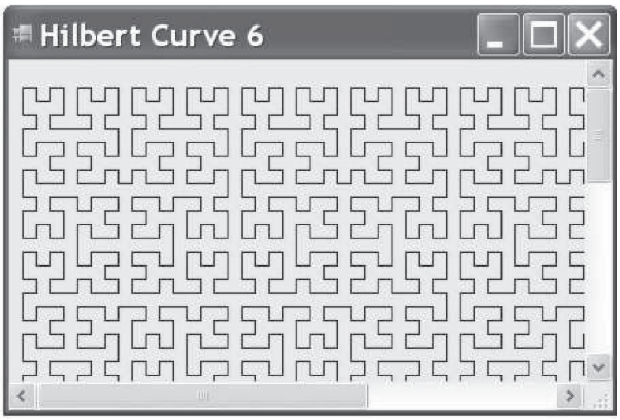
Figure 7.2 Panel coordinates

```
open System.Drawing
open System.Windows.Forms
Application.EnableVisualStyles();;

let winSize = Size(450,300);; // Initial window size in pixels

let display(title: string,(c: Curve.Curve,pw: int,ph: int)) =
    let f(x,y) = Point(int(round x), ph - int(round y))
    let clst = Curve.toList c
    let Ptlst = List.map f clst
    let pArr = Array.ofList Ptlst

    let pen = new Pen(Color.Black)
    let draw(g:Graphics) = g.DrawLines(pen,pArr)

    let panel = new Panel(Dock=DockStyle.Fill)
    panel.Paint.Add(fun e -> draw(e.Graphics))

    let win = new Form(Text=title,Size=winSize,AutoScroll=true,
                       AutoScrollMinSize=Size(pw,ph))
    win.Controls.Add(panel)
    win.Show();;
val display : string * (Curve.Curve * int * int) -> unit
```

Table 7.10 *The* `display` *function*

The function `display` declared in Table 7.10 has two parameters: the `title` to be written on top of the window and a triple comprising the `Curve` to be displayed plus width `pw` and height `ph` of the panel. The function consists of five parts:

1. The function `f` converts a set of coordinates (x, y) to a `Point` object containing the corresponding panel coordinates. The panel coordinates are integers and the conversion from `float` to `int` consists of a `round` followed by an `int` conversion. The formula $(*)$ on Page 165 is used in converting to panel coordinates.
   The list `clst` of coordinates of points on the curve is extracted and the function `f` is applied to each element to get the corresponding list `Ptlst` of `Point` objects. Finally the corresponding array `pArr` of `Point` objects is made. It is ready to be used by the `Graphics` member function `DrawLines`.
2. A `Pen` object `pen` is created and a function `draw` drawing the curve on a `Graphics` object is declared. It calls `DrawLines` using `pen` and the array `pArr` of curve point coordinates.
3. The `Panel` object is created and configured to fill all of the window (`DockStyle`). The draw function is added to the panel's collection of `Paint` objects
4. The window (`Form`) is created using the specified `title`. The size is set and scrolling is enabled. The value of `AutoScrollMinSize` is set to allow the window to scroll to any part of the panel, and scrolling is activated. Finally, the panel is added to the collection of `Controls` of the window.
5. The window is shown (`win.Show()`).



A window is a live object that handles a number of events. The actual window has only events corresponding to manipulation of the window like: resizing of window, use of scrollbars, window comes in to foreground. The part of the panel inside the window is then redrawn using the function in the `Paint` collection of the panel. The parameter `e` is actually an `Event` object.

Some of the above calls of constructors use *optional property setting* like the argument `Dock=DockStyle.Fill` in the argument list of constructor `Panel`. This constructor has actually no `Dock` argument. The specified value `DockStyle.Fill` is instead used as initial value of the `Dock` property of the created `Panel` object.

A curve requires some adjustment before the `display` function can be used: the curve must be suitable scaled to get a proper size of the details of the curve, and the curve must be moved away from the boundary of the panel as boundary points are invisible. This job is done by the function `adjust`. It multiplies the curve `c` by the factor `a` and makes a parallel translation of the curve to leave a blank band of 10 pixels in the panel around the curve:

```
let adjust(c:Curve.Curve, a: float) =
    let c1 = a * c --> (10.0, 10.0)
    let (_,(maxX,maxY)) = Curve.boundingBox c1
    let pw = int(round maxX) + 20
    let ph = int(round maxY) + 20
    (c1,pw,ph);;
```

The value of `adjust` can be used directly as second parameter in the `display` function.

Displaying Hilbert Curves

The `display` function can be used to display Hilbert curves (using the above declarations of value `h0` and function `hilbert`):

```
let h1 = hilbert h0;;
let h2 = hilbert h1;;
let h3 = hilbert h2;;
let h4 = hilbert h3;;
```

```
let h5 = hilbert h4;;
let h6 = hilbert h5;;
display("Hilbert Curve 6", adjust(h6, 10.0));;
```

The displayed curve has been scaled by a factor `10.0` to get a reasonable drawing.

Implementation of the `Curve` *library*

The F# implementation of the `Curve` library is given in Tables 7.11 and 7.12.

A curve is represented by a value of tagged type

```
C of (float * float) * ((float * float) list)
```

containing the coordinates of the start point of the curve plus the (possibly empty) list of coordinates of the remaining points. Any value of this type represents a curve (there is no invariant) and the functions can hence be implemented without any error case where an exception should be raised.

```
module Curve
type Curve =  C of (float*float) * ((float*float) list)

let map  f (C(p0,ps)) = C(f p0,List.map f ps)
let mapP g (C(p0,ps)) = C(p0,List.map (g p0) ps)

type Curve with
  static member(+) (c1:Curve, c2:Curve) =
    match (c1,c2) with
    | (C(p1,ps1),C(p2,ps2)) -> C(p1,ps1@(p2::ps2))
  static member (*) (a: float, c: Curve) =
    let multA  (x0,y0) (x,y) =
      (x0 + a * (x - x0), y0 + a * (y - y0))
    mapP multA c
  static member (|^) (c:Curve, ang: float) =
    let piFact = System.Math.PI / 180.0
    let cs = cos (piFact * ang)
    let sn = sin (piFact * ang)
    let rot (x0,y0) (x,y) =
      let (dx,dy) = (x - x0,y - y0)
      (x0 + cs * dx - sn * dy, y0 + sn * dx + cs * dy)
    mapP rot c
  static member (|^) (c:Curve, ang: int) = c |^ (float ang)
  static member (-->) (c: Curve, (x1,y1): float*float) =
    match c with
    | C((x0,y0),_) -> map (fun (x,y) -> (x-x0+x1, y-y0+y1)) c
  static member (><) (c:Curve, a: float) =
    map (fun (x,y) -> (2.0 * a - x, y)) c
```

Table 7.11 *First part of the implementation of* `Curve` *library*

```
let point (p: float*float) = C (p,[])
let verticRefl (c:Curve) (b:float) =
    map (fun (x,y) -> (x, 2.0*b - y)) c
let boundingBox (C((x0,y0),ps)) =
    let minmax ((minX,minY),(maxX,maxY)) ((x,y):float*float) =
        ((min minX x, min minY y),(max maxX x, max maxY y))
    List.fold minmax ((x0,y0),(x0,y0)) ps
let width (c:Curve)  = let ((minX,_),(maxX,_)) = boundingBox c
                       maxX - minX
let height (c:Curve) = let ((_,minY),(_,maxY)) = boundingBox c
                       maxY - minY
let toList (C(p,ps))  = p :: ps
```

Table 7.12 *Last part of the implementation of* `Curve` *library*

A simplification is obtained by introducing two higher-order local functions. The first function `map` applies a function `f` to the coordinates of each curve point including the start point. It is used in declaring functions like parallel translation `-->` and reflection `><` where the same transformation is applied to all curve points.

The second function `mapP` leaves the start point `p0` unchanged and applies a partially evaluated function `g p0` to the coordinates of the remaining curve points. It is used in declaring functions like multiplication `*` or rotation `|^`.

The combined curve `c1+c2` is obtained as the start point of `c1` together with the list of remaining points of `c1` with all points of `c2` appended.

The multiplication with factor `a` from the point $P_0 : (x_0, y_0)$ maps a point $P : (x, y)$ to the point $P' : (x', y')$ where $\overrightarrow{P_0P'} = \texttt{a}\ \overrightarrow{P_0P}$, that is;

$$(x' - x_0, y' - y_0) = (\texttt{a}\ (x - x_0),\ \texttt{a}\ (y - y_0))$$

and the function `multA` is declared accordingly.

The declaration of `rot` is based on the fact that the rotation with angle v around P_0 maps a point P in to the point P' where $\overrightarrow{P_0P'}$ is obtained from $\overrightarrow{P_0, P}$ by a rotation with angle v.

The function `minmax` extends a (bounding) box with lower left corner $(\texttt{minX}, \texttt{minY})$ and upper right corner $(\texttt{maxX}, \texttt{maxY})$ to contain also the point $(\texttt{x}, \texttt{y})$. The bounding box of the curve is then obtained starting with the one-point box containing the start point and folding the `minmax` function over the remaining points of the curve.

Summary

We have introduced the notions of module, signature and implementation – concepts that are needed when a programmer makes his own libraries. Moreover, we have introduced the notion of type augmentation and shown how it can be used to declare overloaded operators and to customize the equality and comparison operations and the string conversion.

Exercises

7.1 Make an implementation file of the vector example in this section using a record type:

```
type Vector = {x: float; y: float}
```

while using the same signature file.

7.2 Make signature and implementation files for a library of complex numbers with overloaded arithmetic operators (cf. Exercise 3.3).

7.3 Make signature and implementation files for a library of multi-sets of integers represented by weakly ascending lists (cf. Exercise 4.11).

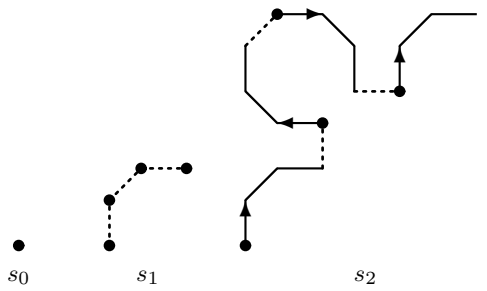
7.4 Make signature and implementation files for a library of polynomials with integer coefficients (cf. Exercise 4.22).

7.5 Customize the `string` function in the library of polynomials in Exercise 7.4.

7.6 Make an indexing in the library of multi-sets of integers in Exercise 7.3 such that the value of s`.[n]` is the number of occurences of `n` in the multi-set s.

7.7 Make an indexing in the library of polynomials in Exercise 7.4 such that p`.[n]` is the coffecient to x^n in the polynomium p.

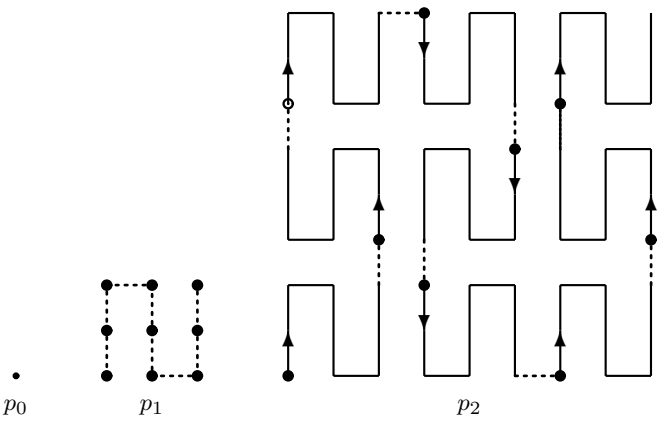
7.8 The Sierpinski curves $s_0, s_1, s_2, ...$ are a system of curves, where the curve s_{n+1} is obtained by joining four curves which are obtained from the curve s_n by transformations composed of reflections, rotations and translations.



The figure shows the Sierpinski curves s_0, s_1 and s_2 and how each of the curves s_1 and s_2 is obtained by joining four curves. All vertical and horizontally segments in a Sierpinski curve have length 1 and all curves s_0, s_1, ... start in the origin. Use the `Curve` library to declare the function `sierpinski` that computes the curve s_{n+1} from the curve s_n for any $n = 0, 1, ...$. Use this function to display the curve s_4 in a window.

7.9 The Peano curves $p_0, p_1, p_2, ...$ are a system of curves, where the curve p_{n+1} is obtained by joining 9 curves which are obtained from the curve p_n by transformations composed of reflections, rotations and translations.

The figure shows the Peano curves p_0, p_1 og p_2 and how each of the curves p_1 and p_2 is obtained by joining 9 curves. All Peano curves start in the origin and the joining segments (thin lines in



the Figure) are of length 1. Use the `Curve` library to declare the function `peano` that computes the curve p_{n+1} from the curve p_n for any $n = 0, 1, \ldots$ (in getting from p_n to p_{n+1} it may be convenient to group the 9 curves into 3 groups each consisting of 3 curves and first build the curve for each of these 3 groups). Use the `peano` function to make a program display the curve p_4 in a window.

7.10 Add a minus operator of type `Curve -> Curve` to the `Curve` library. It should compute the reversed curve, that is, $-c$ should contain the same point as c but taken in the opposite order.

7.11 Make a library for manipulation of pictures (following ideas due to Henderson, cf. [6]). A picture is a set of segments together with a rectangular, upright bounding box in the plane. The bounding box is not shown when drawing a picture but it is used when defining operations on pictures. We use usual rectangular, Cartesian coordinates in the plane, so points in the plane are represented by coordinates which are pairs (x, y) of float numbers. The point with coordinates `(0.0,0.0)` is called the *origin* of the plane. A picture is normally placed in the coordinate system such that the bounding box is situated in the lower left corner of the first quadrant.
If c is a float number with $c > 0.0$ then a picture can be scaled by factor c by mapping each point (x, y) to the point $(c{\star}x, c{\star}y)$. The scaled picture will have width $c{\star}a$ and height $c{\star}b$ where a and b are width and height of the original picture. Scaling is used in some of the below operations in order to adjust the width or the height of a picture.
The library should contain the following functions on pictures:

Grid: Computes a picture directly from width and height of the bounding box and the coordinates of the pairs of end-points of the segments in the picture. The function should be declared such that all the numbers in the input are integers (the function must convert to float numbers as used in the value representing a picture).

Rotate: Computes the picture p' obtained from the picture p by first rotating 90° in the positive (counter-clockwise) direction around the origin and then translating the resulting picture to the right to get its lower left-hand corner into the origin. The height of p' will be the width of p and the width of p' will be the height of p.

Flip: Computes the picture obtained from a picture by horizontal reflection around the vertical line through the middle of the bounding box.

Beside: Computes the picture obtained from two pictures p_1 and p_2 by uniting p_1 with a version of p_2 that has been placed to the right of p_1 and scaled to the same height.

Above: Computes the picture obtained from two pictures p_1 and p_2 by uniting p_2 with a version of p_1 that has been placed on top of p_2 and scaled to the same width.

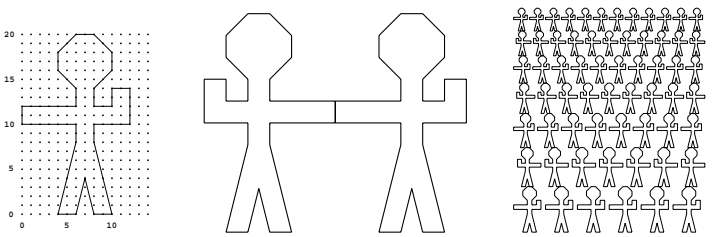
Row: Computes the picture obtained by placing n copies of a picture p beside each other.

Column: Computes the picture obtained by placing n copies of a picture p on top of each other.

Coordinates: Computes the pair $((width, height), segmentList)$ where *width* and *height* are width and height of a picture while *segmentList* is a list of coordinate pairs $((x, y), (x', y'))$ of end-points of the segments in the picture.

You should chose your own names of the functions and use operators whenever appropriate. Furthermore, you should implement a function to display a picture in a window. (Hint: `DrawLine(`*Pen*`,` *Point*$_1$`,` *Point*$_2$`)` draws a segment.)
The library should be used to construct pictures of persons and Escher's fishes – as described in the following.

Persons



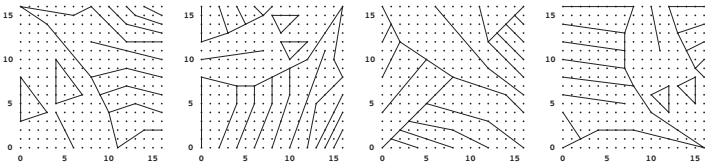
(a) `man` (b) `couple` (c) `crowd`

Figure 7.3 The `man` picture and derived pictures

The starting point is the picture `man` shown in Figure 7.3. It has width 14 and heigh 20. Using the functions on pictures you should now make programs to construct the pictures `couple` and `crowd` shown in Figure 7.3.

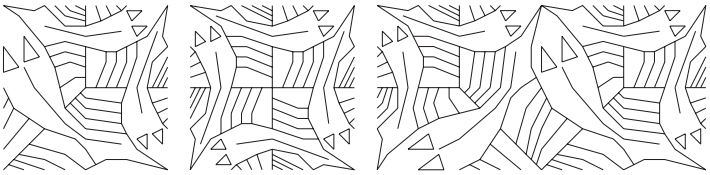
Escher's fishes

The starting point of Escher's fishes is the four (16×16) pictures `p`, `q`, `r`, and `s` shown in Figure 7.4. By combining these four pictures we get the picture `t` in Figure 7.5, while the picture `a` is obtained by combining suitably rotated copies of `q`. Finally the picture `b1` is obtained by combining two suitably rotated copies of `t`.
The Escher fish pictures `e0`, `e1` and `e2` are now obtained by combining the pictures in Figure 7.5 as shown in Figure 7.6. The pictures `b2`, `b3` and `b4` are obtained from `b1` by successive rotations. The transition from an Escher picture to the next adds a border around the picture consisting of a picture `a` in each corner, a row of `b1`'s at the top, a column of `b2`'s at the left, a row of `b3`'s at the bottom, and a column of `b4`'s at the right. In this border there will be one `b1` on top of an `a` and two `b1`'s on top of a `b1`, one `b2` to the left of an `a` and two `b2`'s to the left of a `b2`, etc.
You should make a program to generate the Escher fish pictures `e0`, `e1` and `e2`.



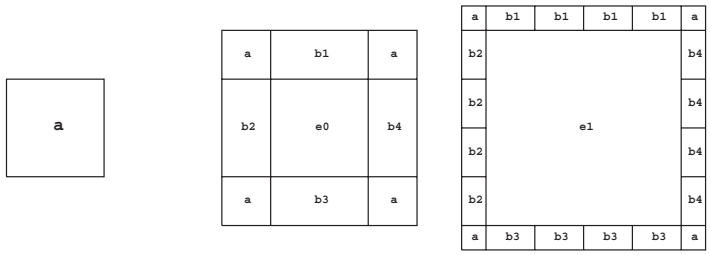
(a) Picture `p` (b) Picture `q` (c) Picture `r` (d) Picture `s`

Figure 7.4 Basic fish pictures



(a) Picture `t` (b) Picture `a` (c) Picture `b1`

Figure 7.5 Escher fish building blocks



(a) Picture `e0` (b) Picture `e1` (c) Picture `e2`

Figure 7.6 Building Escher fish pictures