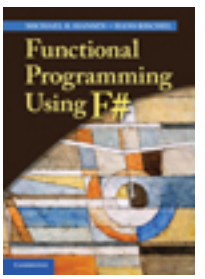


Functional Programming Using F#

Michael R. Hansen, Hans Rischel

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

Online ISBN: 9781139093996

Hardback ISBN: 9781107019027

Paperback ISBN: 9781107684065

Chapter

1 - Getting started pp. 1-20

Chapter DOI: http://dx.doi.org/10.1017/CBO9781139093996.002

Cambridge University Press

1

Getting started

In this chapter we will introduce some of the main concepts of functional programming languages. In particular we will introduce the concepts of value, expression, declaration, recursive function and type. Furthermore, to explain the meaning of programs we will introduce the notions: binding, environment and evaluation of expressions.

The purpose of the chapter is to acquaint the reader with these concepts, in order to address interesting problems from the very beginning. The reader will obtain a thorough knowledge of these concepts and skills in applying them as we elaborate on them throughout this book.

There is support of both compilation of F# programs to executable code and the execution of programs in an interactive mode. The programs in this book are usually illustrated by the use of the interactive mode.

The interface of the interactive F# compiler is very advanced as, for example, structured values like tuples, lists, trees and functions can be communicated directly between the user and the system without any conversions. Thus, it is very easy to experiment with programs and program designs and this allows us to focus on the main structures of programs and program designs, that is, the core of programming, as input and output of structured values can be handled by the F# system.

1.1 Values, types, identifiers and declarations

In this section we illustrate how to use an F# system in interactive mode.

The interactive interface allows the user to enter, for example, an arithmetic expression in a line, followed by two semicolons and terminated by pressing the `return` key:

```
2*3 + 4;;
```

The answer from the system contains the value and the type of the expression:

```
val it : int = 10
```

The system will add some leading characters in the input line to make a distinction between input from the user and output from the system. The dialogue may look as follows:

```
> 2*3 + 4;;
val it : int = 10
>
```

The leading string "> " is output whenever this particular system is awaiting input from the user. It is called the *prompt*, as it "prompts" for input from the user. The input from the user is ended by a double semicolon "`;;`" while the next line contains the answer from the system.

In the following we will distinguish between user input and answer from the system by the use of different type fonts:

```
2*3 + 4;;
val it : int = 10
```

The input from the user is written in `typewriter` font while the answer from the system is written in *`italic typewriter`* font.

The above answer starts with the *reserved word* `val`, which indicates that a value has been computed, while the special *identifier* `it` is a name for the computed value, that is, 10. The *type* of the result is `int`, denoting the subset of the integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$ that can be represented using the system.

The user can give a name to a value by entering a *declaration*, for instance:

```
let price = 125;;
```

where the reserved word `let` starts the declarations. In this case the system answers:

```
val price : int = 125
```

The *identifier* `price` is now a name for the integer value 125. We also say that the identifier `price` is *bound* to 125.

Identifiers which are bound to values can be used in expressions:

```
price * 20;;
val it : int = 2500
```

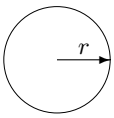
The identifier `it` is now bound to the integer value `2500`, and this identifier can also be used in expressions:

```
it / price = 20;;
val it : bool = true
```

The operator `/` is the quotient operator on integers. The expression `it/price = 20` is a question to the system and the identifier `it` is now bound to the answer `true` of type `bool`, where `bool` is a type denoting the two-element set $\{\text{true}, \text{false}\}$ of truth values. Note that the equality sign in the input is part of an expression of type `bool`, whereas the equality sign in the answer expresses a binding of the identifier `it` to a value.

1.2 Simple function declarations

We now consider the declaration of functions. One can name a *function*, just as one can name an integer constant. As an example, we want to compute the area of a circle with given radius r, using the well known area function: $\text{circleArea}(r) = \pi r^2$.



Circle with radius r and area πr^2.

The constant π is found in the Library under the name `System.Math.PI`:

```
System.Math.PI;;
val it : float = 3.141592654
```

The type `float` denotes the subset of the real numbers that can be represented in the system, and `System.Math.PI` is bound to a value of this type.

We choose the name `circleArea` for the circle area function, and the function is then declared using a `let`-declaration:

```
let circleArea r = System.Math.PI * r * r;;
val circleArea : float -> float
```

The answer says that the identifier `circleArea` now denotes a value, as indicated by the reserved word `val` occurring in the answer. This value is a *function* with the type `float -> float`, where the symbol `->` indicates a function type and the argument as well as the value of the function has type `float`. Thus, the answer says that `circleArea` is bound to a value that is some function of type `float -> float`.

The function `circleArea` can be *applied* to different *arguments*. These arguments must have the type `float`, and the result has type `float` too:

```
circleArea 1.0;;
val it : float = 3.141592654

circleArea (2.0);;
val it : float = 12.56637061
```

Brackets around the argument `1.0` or `(2.0)` are optional, as indicated here.

The identifier `System.Math.PI` is a composite identifier. The identifier `System` denotes a *namespace* where the identifier `Math` is defined, and `System.Math` denotes a namespace where the identifier `PI` is defined. Furthermore, `System` and `System.Math` denote parts of the .NET Library. We encourage the reader to use program libraries whenever appropriate. In Chapter 7 we describe how to make your own program libraries.

Comments

A string enclosed within a matching pair `(*` and `*)` is a *comment* which is ignored by the F# system. Comments can be used to make programs more readable for a human reader by explaining the intention of the program, for example:

```
(* Area of circle with radius r *)
let circleArea r = System.Math.PI * r * r;;
val circleArea : float -> float
```

Two slash characters // can be used for one-line comments:

```
// Area of circle with radius r
let circleArea r = System.Math.PI * r * r;;
val circleArea : float -> float
```

A comment line can also begin with three slash characters ///. The tool XMLDocs can produce program documentation from such comment, but we will not pursue this any further in this book.

Comments can be very useful, especially in large programs, but long comments should be avoided as they tend to make it more difficult for the reader to get an overview of the program.

1.3 Anonymous functions. Function expressions

A function can be created in F# without getting any name. This is done by evaluating a *function expression*, that is an expression where the *value* is a *function*. This section introduces simple function expressions and function expressions with patterns.

A nameless, anonymous function can be defined by a *simple function expression*, also called a *lambda expression*,[1] for example:

```
fun r -> System.Math.PI * r * r;;
val it : float -> float = <fun:clo@10-1>
it 2.0;;
val it : float = 12.56637061
```

The expression `fun r -> System.Math.PI * r * r` denotes the circle-area function and it reads: "the function of r given by $\pi \cdot r^2$". The reserved word `fun` indicates that a function is defined, the identifier `r` occurring to the left of `->` is a pattern for the argument of the function, and `System.Math.PI * r * r` is the expression for the value of the function.

The declaration of the circle-area function could be made as follows:

```
let circleArea = fun r ->  System.Math.PI * r * r;;
val circleArea : float -> float
```

but it is more natural in this case to use a `let`-declaration `let circleArea r = ...` with an argument pattern. We shall later see many uses of anonymous functions.

Function expressions with patterns

It is often convenient to define a function in terms of a number of cases. Consider, for example, a function giving the number of days in a month, where a month is given by its number, that is, an integer between 1 and 12. Suppose that the year of consideration is not a leap year. This function can thus be expressed as:

[1] Lambda calculus was introduced by Alonzo Church in the 1930s. In this calculus an expression of the form $\lambda x.e$ was used to denote the function of x given by the expression e. The `fun`-notation in F# is a direct translation from λ-expressions.

```
function
| 1  -> 31  // January
| 2  -> 28  // February
| 3  -> 31  // March
| 4  -> 30  // April
| 5  -> 31  // May
| 6  -> 30  // June
| 7  -> 31  // July
| 8  -> 31  // August
| 9  -> 30  // September
| 10 -> 31  // October
| 11 -> 30  // November
| 12 -> 31;;// December
  function
  ^
stdin(17,1): warning FS0025: Incomplete pattern matches on
this expression. For example, the value '0' may indicate a
case not covered by the pattern(s).
val it : int -> int = <fun:clo@17-2>
```

The last part of the answer shows that the computed value, named by `it`, is a function with the type `int -> int`, that is, a function from integers to integers. The answer also shows the internal name for that function. The first part of the answer is a warning that the set of patterns used in the `function`-expression is incomplete. The expression enumerates a value for every legal number for a month $(1, 2, \ldots, 12)$. At this moment we do not care about other numbers.

The function can be applied to 2 to find the number of days in February:

```
it 2;;
val it : int = 28
```

This function can be expressed more compactly using a *wildcard pattern* "_":

```
function
| 2  -> 28  // February
| 4  -> 30  // April
| 6  -> 30  // June
| 9  -> 30  // September
| 11 -> 30  // November
| _  -> 31;;// All other months
```

In this case, the function is defined using six clauses. The first clause `2 -> 28` consists of a pattern `2` and a corresponding expression `28`. The next four clauses have a similar explanation, and the last clause contains a wildcard pattern. Applying the function to a value v, the system finds the clause containing the first pattern that matches v, and returns the value of the corresponding expression. In this example there are just two kinds of matches we should know:

- A constant, like 2, matches itself only, and
- the wildcard pattern _ matches any value.

For example, applying the function to 4 gives 30, and applying it to 7 gives 31.

An even more succinct definition can be given using an *or*-pattern:

```
function
| 2         -> 28  // February
| 4|6|9|11  -> 30  // April, June, September, November
| _         -> 31  // All other months
;;
```

The or-pattern `4|6|9|11` matches any of the values $4, 6, 9, 11$, and no other values.

We shall make extensive use of such a case splitting in the definition of functions, also when declaring named functions:

```
let daysOfMonth = function
    | 2         -> 28  // February
    | 4|6|9|11  -> 30  // April, June, September, November
    | _         -> 31  // All other months
;;
val daysOfMonth : int -> int

daysOfMonth 3;;
val it : int = 31

daysOfMonth 9;;
val it : int = 30
```

1.4 Recursion

This section introduces the concept of recursion formula and recursive declaration of functions by an example: the factorial function $n!$. It is defined by:

$$\begin{aligned} 0! &= 1 \\ n! &= 1 \cdot 2 \cdot \ldots \cdot n \quad \text{for } n > 0 \end{aligned}$$

where n is a non-negative integer. The dots $\cdots$ indicate that all integers from 1 to n should be multiplied. For example:

$$4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$$

Recursion formula

The underbraced part of the below expression for $n!$ is the expression for $(n-1)!$:

$$n! = \underbrace{1 \cdot 2 \cdot \ldots \cdot (n-1)}_{(n-1)!} \cdot n \qquad \text{for } n > 1$$

so we get the formula:

$$n! = n \cdot (n-1)! \qquad \text{for } n > 1$$

This formula is actually correct also for $n = 1$ as:

$$0! = 1 \qquad \text{and} \qquad 1 \cdot (1-1)! = 1 \cdot 0! = 1 \cdot 1 = 1$$

so we get:

$$\begin{array}{lcll} 0! & = & 1 & \text{(Clause 1)} \\ n! & = & n \cdot (n-1)! \quad \text{for } n > 0 & \text{(Clause 2)} \end{array}$$

This formula is called a *recursion formula* for the factorial function (_!) as it expresses the value of the function for some argument n in terms of the value of the function for some other argument (here: $n - 1$).

Computations

This definition has a form that can be used in the computation of values of the function. For example:

$$\begin{array}{rl} & 4! \\ = & 4 \cdot (4-1)! \\ = & 4 \cdot 3! \\ = & 4 \cdot (3 \cdot (3-1)!) \\ = & 4 \cdot (3 \cdot 2!) \\ = & 4 \cdot (3 \cdot (2 \cdot (2-1)!)) \\ = & 4 \cdot (3 \cdot (2 \cdot 1!)) \\ = & 4 \cdot (3 \cdot (2 \cdot (1 \cdot (1-1)!))) \\ = & 4 \cdot (3 \cdot (2 \cdot (1 \cdot 0!))) \\ = & 4 \cdot (3 \cdot (2 \cdot (1 \cdot 1))) \\ = & 24 \end{array}$$

The clauses of the definition of the factorial function are applied in a purely "mechanical" way in the above computation of $4!$. We will now take a closer look at this mechanical process as the system will compute function values in a similar manner:

Substitution in clauses

The first step is obtained from Clause 2, by *substituting* 4 for n. The condition for using the second clause is satisfied as $4 > 0$. This step can be written in more detail as:

$$\begin{array}{rll} & 4! & \\ = & 4 \cdot (4-1)! & \text{(Clause 2, } n = 4\text{)} \end{array}$$

Computation of arguments

The new argument $(4 - 1)$ of the factorial function in the expression $(4 - 1)!$ is computed in the next step:

$$\begin{array}{rll} & 4 \cdot (4-1)! & \\ = & 4 \cdot 3! & \text{(Compute argument of !)} \end{array}$$

Thus, the principles used in the first two steps of the computation of $4!$ are:

- Substitute a value for n in Clause 2.
- Compute argument.

These are the only principles used in the above computation until we arrive at the expression:

$$4 \cdot (3 \cdot (2 \cdot (1 \cdot 0!)))$$

The next computation step is obtained by using Clause 1 to obtain a value of $0!$:

$$\begin{array}{rl} & 4 \cdot (3 \cdot (2 \cdot (1 \cdot 0!))) \\ = & 4 \cdot (3 \cdot (2 \cdot (1 \cdot 1))) \qquad \text{(Clause 1)} \end{array}$$

and the multiplications are then performed in the last step:

$$\begin{array}{rl} & 4 \cdot (3 \cdot (2 \cdot (1 \cdot 1))) \\ = & 24 \end{array}$$

This recursion formula for the factorial function is an example of a general pattern that will appear over and over again throughout the book. It contains a clause for a *base case* "0!", and it contains a clause where a more general case "$n!$" is reduced to an expression "$n \cdot (n-1)!$" involving a "smaller" instance "$(n-1)!$" of the function being characterized. For such recursion formulas, the computation process will terminate, that is, the computation of $n!$ will terminate for all $n \geq 0$.

Recursive declaration

We name the factorial function `fact`, and this function is then declared as follows:

```
let rec fact = function
    | 0 -> 1
    | n -> n * fact(n-1);;
val fact : int -> int
```

This declaration corresponds to the recursion formula for $n!$. The reserved word `rec` occurring in the `let`-declaration allows the identifier being declared (`fact` in this case) to occur in the defining expression.

This declaration consists of two clauses

`0 -> 1` and `n -> n * fact(n-1)`

each initiated by a vertical bar. The *pattern* of the first clause is the constant `0`, while the pattern of the second clause is the identifier `n`.

The patterns are *matched* with integer arguments during the *evaluation* of function values as we shall see below. The only value matching the pattern `0` is `0`. On the other hand, every value matches the pattern `n`, as an identifier can name any value.

Evaluation

The system uses the declaration of `fact` to evaluate function values in a way that resembles the above computation of $4!$.

Substitution in clauses

To evaluate `fact 4`, the system searches for a clause in the declaration of `fact`, where `4` matches the pattern of the clause.

The system starts with the first clause of the declaration: `0 -> 1`. This clause is skipped as the value `4` does not match the pattern `0` of this clause.

Then, the second clause: `n -> n * fact(n-1)` is investigated. The value `4` matches the pattern of this clause, that is, the identifier `n`. The value `4` is bound to `n` and then substituted for `n` in the right-hand side of this clause thereby obtaining the expression: `4 * fact(4-1)`.

We say that the expression `fact 4` *evaluates to* `4 * fact(4-1)` and this evaluation is written as:

```
    fact 4
⇝   4 * fact(4-1)
```

where we use the symbol $\rightsquigarrow$ for a step in the evaluation of an expression. Note that the symbol $\rightsquigarrow$ is not part of any program, but a symbol used in explaining the evaluation of expressions.

Evaluation of arguments

The next step in the evaluation is to evaluate the argument `4-1` of `fact`:

```
    4 * fact(4-1)
⇝   4 * fact 3
```

The evaluation of the expression `fact 4` proceeds until a value is reached:

```
    fact 4
⇝   4 * fact(4-1)                      (1)
⇝   4 * fact 3                         (2)
⇝   4 * (3 * fact(3-1))                (3)
⇝   4 * (3 * fact 2)                   (4)
⇝   4 * (3 * (2 * fact(2-1)))          (5)
⇝   4 * (3 * (2 * fact 1))             (6)
⇝   4 * (3 * (2 * (1 * fact(1-1))))    (7)
⇝   4 * (3 * (2 * (1 * fact 0)))       (8)
⇝   4 * (3 * (2 * (1 * 1)))            (9)
⇝   4 * (3 * (2 * 1))                 (10)
⇝   4 * (3 * 2)                       (11)
⇝   4 * 6                             (12)
⇝   24                                (13)
```

The argument values 4, 3, 2 and 1 do not match the pattern `0` in the first clause of the declaration of `fact`, but they match the second pattern `n`. Thus, the second clause is chosen for further evaluation in the evaluation steps (1), (3), (5) and (7).

The argument value 0 does, however, match the pattern `0`, so the first clause is chosen for further evaluation in step (9). The steps (2), (4), (6) and (8) evaluate argument values to `fact`, while the last steps (10) - (13) reduce the expression built in the previous steps.

Unsuccessful evaluations

The evaluation of `fact` n may not evaluate to a value, because

- the system will run out of memory due to long expressions,
- the evaluation may involve bigger integers than the system can handle, or
- the evaluation of an expression may not terminate.[2]

For example, applying `fact` to a negative integer leads to an *infinite evaluation*:

```
    fact -1
~>  -1 * fact(-1 - 1)
~>  -1 * fact -2
~>  -1 * (-2 * fact(-2 - 1))
~>  -1 * (-2 * fact -3)
~>  ...
```

A remark on recursion formulas

The above recursive function declaration was motivated by the recursion formula:

$$\begin{aligned} 0! &= 1 \\ n! &= n \cdot (n-1)! \quad \text{for } n > 0 \end{aligned}$$

which gives a unique characterization of the factorial function.

The factorial function may, however, be characterized by other recursion formulas, for example:

$$\begin{aligned} 0! &= 1 \\ n! &= \frac{(n+1)!}{n+1} \quad \text{for } n \geq 0 \end{aligned}$$

This formula is *not* well-suited for computations of values, because the corresponding function declaration based on this formula (where `/` denotes integer division):

```
let rec f = function
    | 0 -> 1
    | n -> f(n+1)/(n+1);;
val f : int -> int
```

gives an infinite evaluation of `f` k when $k > 0$. For example:

```
    f 2
~>  f(2+1)/(2+1)
~>  f(3)/3
~>  f(3+1)/(3+1)
~>  ...
```

[2] Note that a text like `fact` n is *not* part of F#. It is a *schema* where one can obtain a program piece by replacing the *meta symbol* n with a suitable F# entity. In the following we will often use such schemas containing meta symbols in *italic font*.

Thus, in finding a declaration of a function, one has to look for a *suitable* recursion formula expressing the computation of function values. This declaration of `f` contains a base case "`f 0`". However, the second clause does not reduce the general case "`f`(n)" to an instance which is closer to the base case, and the evaluation of `f`(n) will not terminate when $n > 0$.

1.5 Pairs

Consider the function:

$$x^n = x \cdot x \cdot \ldots \cdot x \qquad n \text{ occurrences of } x, \text{ where } n \geq 0$$

where x is a real number and n is a natural number.

The under-braced part of the expression below for x^n is the expression for x^{n-1}:

$$x^n = x \cdot \underbrace{x \cdot \ldots \cdot x}_{x^{n-1}} \qquad n \text{ occurrences of } x, \text{ where } n > 0$$

Using the convention: $x^0 = 1$, the function can be characterized by the recursion formula:

$$\begin{array}{lcl} x^0 & = & 1 \\ x^n & = & x \cdot x^{n-1} \quad \text{for } n > 0 \end{array}$$

In mathematics x^n is a function of two variables x and n, but it is treated differently in F# using the concept of a *pair*:

If a_1 and a_2 are values of types τ_1 and τ_2 then (a_1,a_2) is a value of type $\tau_1 * \tau_2$

For example:

```
let a = (2.0,3);;
val a = (2.0, 3) : float * int
```

Furthermore, given patterns pat_1 and pat_2 there is a *composite* pattern `(`pat_1`,`pat_2`)`. It matches a pair `(`a_1`,`a_2`)` exactly when pat_1 matches a_1 and pat_2 matches a_2, for example:

```
let (x,y) = a;;
val y : int = 3
val x : float = 2.0
```

The concept of a pair is a special case of tuples that are treated in Section 3.1.

Using these concepts we represent x^n as a function `power` with a pair `(x,n)` as the argument. The following declaration is based on the above recursion formula, using composite patterns `(x,0)` and `(x,n)`:

```
let rec power = function
    | (x,0) -> 1.0                  //  (1)
    | (x,n) -> x * power(x,n-1);;   //  (2)
val power : float * int -> float
```

The type of `power` is `float * int -> float`. The argument of `power` is therefore a pair of type `float * int` while the value of the function is of type `float`.

The `power` function can be applied to pairs of type `float * int`:

```
power a;;
val it : float = 8.0

power(4.0,2);;
val it : float = 16.0
```

A function in F# has *one* argument and *one* value. In this case the argument is a pair (u, i) of type `float * int`, while the value of the function is of type `float`.

The system evaluates the expression `power(4.0,2)` as follows:

```
   power(4.0,2)
⇝  4.0 * power(4.0,2-1)          (Clause 2, x is 4.0, n is 2)
⇝  4.0 * power(4.0,1)
⇝  4.0 * (4.0 * power(4.0,1-1))  (Clause 2, x is 4.0, n is 1)
⇝  4.0 * (4.0 * power(4.0,0))
⇝  4.0 * (4.0 * 1.0)             (Clause 1, x is 4.0)
⇝  16.0
```

Notes on pattern matching

Note that the *order* of the *clauses* in the declaration of `power` is significant. The following declaration will *not* work:

```
let rec powerNo = function
    | (x, n) -> x * powerNo(x,n-1)   // This does NOT work
    | (x, 0) -> 1.0
;;
```

The first pattern `(x,n)` will match any pair of form (u, i) and the second clause will consequently never come into use. The F# compiler actually discovers this and issues a warning:

```
     | (x, 0) -> 1.0
    ----^^^^^^
... warning FS0026: This rule will never be matched
```

The function can be applied to an argument (despite the warning), but that would give an infinite evaluation since the base case `(x, 0) -> 1.0` is never reached.

A similar remark on the order of clauses applies to the declaration of `fact`.

One should also note that a prior binding of an identifier used in a pattern has no effect on the pattern matching.[3] Hence, the following will also *not* work:

```
let zero = 0;;

let rec powerNo = function
    | (x,zero) -> 1.0                // This does NOT work
    | (x,n)    -> x * powerNo(x,n-1)
;;
```

The first pattern `(x,zero)` will match any pair of form (u, i), binding `x` to u and `zero` to i so the second clause will again never come into use. The F# compiler issues a warning like in the previous example.

1.6 Types and type checking

The examples in the previous sections show that types like `float * int -> float` or `int` form an integral part of the responses from the system.

In fact, F# will try to *infer* a *type* for each value, expression and declaration entered. If the system can infer a type for the input, then the input is accepted by the system. Otherwise the system will reject the input with an error message.

For example, the expression `circleArea 2.0` is accepted, because

- `circleArea` has the type `float -> float`, and
- `2.0` has the type `float`.

Furthermore, the result of evaluating `circleArea 2.0`, that is `12.5663706144`, has type `float`.

On the other hand, the system will reject the expression `circleArea 2` with an error message since `2` has type `int` while the argument for `circleArea` must be of type `float`:

```
circleArea 2;;
  circleArea 2;;
  -----------^

stdin(95,12): error FS0001: This expression was expected to
have type
    float
but here has type
    int
```

The above type consideration for *function application* $f(e)$ is a special case of the general type rule for function application:

if f has type τ_1 -> τ_2 and e has type τ_1 then $f(e)$ has type τ_2.

[3] Identifiers that are *constructors* are, however, treated in a special way (cf. Section 3.8).

Using the notation $e : \tau$ to assert that the expression e has type τ, this rule can be presented more succinctly as follows:

if $f : \tau_1$ -> τ_2 and $e : \tau_1$
then $f(e) : \tau_2$.

Consider, for example, the function `power` with type `float * int -> float`. In this case, τ_1 is `float * int` and τ_2 is `float`. Furthermore, the pair `(4.0,2)` has type `float * int` (which is τ_1). According to the above rule, the expression `power(4.0,2)` hence has type `float` (which is τ_2).

1.7 Bindings and environments

In the previous sections we have seen that identifiers can be bound to denote an integer, a floating-point value, a pair or a function. The notions of *binding* and *environment* are used to explain that entities are bound by identifiers.

The *execution* of a declaration, say `let` $x = e$, causes the identifier x to be bound to the value of the expression e. For example, the execution of the declaration:

```
let a = 3;;
val a : int = 3
```

causes the identifier `a` to be bound to 3. This binding is denoted by `a` $\mapsto 3$.

Execution of further declarations gives extra bindings. For example, execution of

```
let b = 7.0;;
val b : float = 7.0
```

gives a further binding `b` $\mapsto 7.0$.

A collection of bindings is called an *environment*, and the environment env_1 obtained from execution of the above two declarations is denoted by:

$$env_1 = \begin{bmatrix} \texttt{a} & \mapsto & 3 \\ \texttt{b} & \mapsto & 7.0 \end{bmatrix}$$

Note that this notation is *not* part of any program. Bindings and environments are mathematical objects used to explain the meaning of programs.

The execution of an additional declaration causes an extension of env_1. For example

```
let c = (2, 8);;
val c : int * int = (2, 8)

let circleArea r = System.Math.PI * r * r;;
val circleArea : float -> float
```

adds bindings of the identifiers `c` and `circleArea` to the environment env_1 giving the environment env_2:

$$env_2 = \begin{bmatrix} \texttt{a} & \mapsto & 3 \\ \texttt{b} & \mapsto & 7.0 \\ \texttt{c} & \mapsto & (2, 8) \\ \texttt{circleArea} & \mapsto & \text{“the circle area function”} \end{bmatrix}$$

The value of an expression is always evaluated in the *actual environment*, that contains the bindings of identifiers that are valid at evaluation time. When the F# system is activated, the actual environment is the *Basis Environment* that gives meanings to `/`, `+`, `-`, `sqrt`, for example. When using environments we will usually not show bindings from the Basis Environment. We will usually also omit bindings of identifiers like `System.Math.PI` from the Library.

1.8 Euclid's algorithm

This section presents the famous algorithm of Euclid for computing the greatest common divisor of two natural numbers.

For a given integer n, an integer d is called a *divisor* of n (written $d|n$) if there exists an integer q such that $n = q \cdot d$. Hence, the number 1 is a divisor of any integer. Any integer $n \neq 0$ has a finite number of divisors as each divisor has absolute value $\leq |n|$, while 0 has infinitely many divisors as any integer is a divisor of 0. Thus, integers m, n have at least one common divisor (namely 1), and if either $m \neq 0$ or $n \neq 0$, then the set of common divisors of m and n is finite.

The *GCD theorem of Euclid* states that for any integers m, n there exists an integer $\gcd(m, n)$ such that $\gcd(m, n) \geq 0$, and such that the *common divisors* of m and n are precisely the *divisors* of $\gcd(m, n)$.

Note that if $m \neq 0$ or $n \neq 0$ then $\gcd(m, n)$ is the *greatest* common divisor of m and n. For $m = 0$ and $n = 0$ we have $\gcd(0, 0) = 0$, as the common divisors for 0 and 0 are precisely the divisors of 0, but 0 and 0 have no *greatest* common divisor as any number is a divisor of 0.

Euclid gave an algorithm for computing $\gcd(m, n)$ for arbitrary integers m and n and this algorithm gives at the same time a proof of the theorem.

Division with remainder. The / and % operators

Euclid's algorithm is based on the concept of *integer division with remainder*. Let m and n be integers with $m \neq 0$. An identity with integers q and r of the form:

$$n = q \cdot m + r$$

is then called a division with quotient q and remainder r. There are infinite many possible remainders (corresponding to different quotients q):

$$\ldots, n - 3 \cdot |m|,\ n - 2 \cdot |m|,\ n - |m|,\ n,\ n + |m|,\ n + 2 \cdot |m|,\ n + 3 \cdot |m|,\ \ldots$$

It follows that there are two possibilities concerning remainders r with $-|m| < r < |m|$:

1. The integer 0 is a remainder and any other remainder r satisfies $|r| \geq |m|$.
2. There are two remainders r_{neg} and r_{pos} such that $-|m| < r_{neg} < 0 < r_{pos} < |m|$.

The F# operators / and % (quotient and remainder) are defined (for $m \neq 0$) such that:

$$n = (n \ / \ m) \cdot m + (n \ \% \ m) \tag{1.1}$$

$$|n \ \% \ m| < |m| \tag{1.2}$$

$$n \ \% \ m \geq 0 \text{ when } n \geq 0 \tag{1.3}$$

$$n \ \% \ m \leq 0 \text{ when } n < 0 \tag{1.4}$$

so $n \ \% \ m = 0$ when m is a divisor of n, otherwise r_{pos} is used if $n > 0$ and r_{neg} if $n < 0$.

Note that the corresponding operators in other programming languages may use different conventions for negative integers.

Euclid's algorithm in F#

Euclid's algorithm is now expressed in the following declaration

```
let rec gcd = function
    | (0,n) -> n
    | (m,n) -> gcd(n % m,m);;
val gcd : int * int -> int
```

For example:

```
gcd(12,27);;
val it : int = 3

gcd(36, 116);;
val it : int = 4
```

Termination of Euclid's algorithm

It is not obvious that the evaluation of $\texttt{gcd}(m, n)$ will terminate with a result for all integers m and n. We will now prove that the second clause in the declaration is in fact used at most $|m|$ times in the evaluation of $\texttt{gcd}(m, n)$. It follows that the evaluation always terminates.

Consider an evaluation with at least k (> 0) steps using the second clause:

$$\begin{array}{lll} & \texttt{gcd}(m, n) & m \neq 0 \\ \rightsquigarrow & \texttt{gcd}(m_1, n_1) & m_1 \neq 0 \\ \rightsquigarrow & \texttt{gcd}(m_2, n_2) & m_2 \neq 0 \\ \cdots & & \\ \rightsquigarrow & \texttt{gcd}(m_{k-1}, n_{k-1}) & m_{k-1} \neq 0 \\ \rightsquigarrow & \texttt{gcd}(m_k, n_k) & \\ \rightsquigarrow & \cdots & \end{array}$$

The right-hand side of the second clause gives the identities:

$$\begin{array}{lcllcl} m_1 & = & n \ \% \ m & n_1 & = & m \\ m_2 & = & n_1 \ \% \ m_1 & n_2 & = & m_1 \\ & \cdots & & & \cdots & \\ m_k & = & n_{k-1} \ \% \ m_{k-1} & n_k & = & m_{k-1} \end{array}$$

Using $|n \,\%\, m| < |m|$ when $m \neq 0$, we get:

$$|m| > |m_1| > \cdots > |m_k| \geq 0$$

It follows that

$$k \leq |m|$$

because $|m_1|, |m_2|, \ldots, |m_k|$ are k mutually different integers among the $|m|$ integers $|m|-1, |m|-2, \ldots, 1, 0$.

The evaluation of `gcd`(m, n) will hence involve at most $|m|$ uses of the second clause.

Proof of Euclid's theorem

The key to prove Euclid's theorem is that the following holds when $m \neq 0$:

The integers $n \,\%\, m$ and m have the same common divisors as the integers n and m

This follows from the identities:

$$n \,\%\, m + q \cdot m = n \quad \text{and} \quad n - q \cdot m = n \,\%\, m \qquad \text{(with integer } q = n/m\text{)}$$

which show that any common divisor of $(n \,\%\, m)$ and m is also a divisor of n, and hence a common divisor of n and m – and conversely – any common divisor of n and m is also a divisor of $(n \,\%\, m)$, and hence a common divisor of $(n \,\%\, m)$ and m.

Using the above integers $n_1, n_2, \ldots$ and $m_1, m_2, \ldots$ we hence get:

m_1 and n_1	have same common divisors as	m and n
m_2 and n_2	have same common divisors as	m_1 and n_1
$\ldots$	$\ldots$	$\ldots$
m_p and n_p	have same common divisors as	m_{p-1} and n_{p-1}

where the evaluation terminates with an index p where $m_p = 0$ and n_p = `gcd(`m`,`n`)`.

The common divisors of 0 and n_p are, however, exactly the divisors of n_p = `gcd(`m`,`n`)` as any integer is a divisor of 0. It follows by induction that the common divisors of m and n are exactly the divisors of `gcd(`m`,`n`)`.

1.9 Evaluations with environments

During the evaluation of expressions the system may create and use temporary bindings of identifiers. This is, for example, the case for function applications like `gcd(36,116)` where the function `gcd` is applied to the argument `(36,116)`. We will study such bindings as it gives insight into how recursive functions are evaluated.

The declaration:

```
let rec gcd = function
    | (0,n) -> n
    | (m,n) -> gcd(n % m,m);;
val gcd : int * int -> int
```

contains two clauses: One with pattern `(0,n)` and expression `n` and another with pattern `(m,n)` and expression `gcd(n % m,m)`. There are hence two cases in the evaluation of an expression $\texttt{gcd}(x, y)$ corresponding to the two clauses:

1. $\texttt{gcd}(0, y)$: The argument $(0, y)$ matches the pattern `(0,n)` in the first clause giving the binding $\texttt{n} \mapsto y$, and the system will evaluate the corresponding right-hand side expression `n` using this binding:
$$\texttt{gcd}(0, y) \rightsquigarrow (\texttt{n}, [\texttt{n} \mapsto y]) \rightsquigarrow y$$
2. $\texttt{gcd}(x, y)$ with $x \neq 0$: The argument (x, y) does not match the pattern `(0,n)` in the first clause but it matches the pattern `(m,n)` in the second clause giving the bindings $\texttt{m} \mapsto x$, $\texttt{n} \mapsto y$, and the system will evaluate the corresponding right-hand side expression `gcd(n % m,m)` using these bindings:
$$\texttt{gcd}(x, y) \rightsquigarrow (\texttt{gcd(n \% m, m)}, [\texttt{m} \mapsto x, \texttt{n} \mapsto y]) \rightsquigarrow \ldots$$

Consider, for example, the expression `gcd(36,116)`. The value `(36,116)` does not match the pattern `(0,n)`, so the first evaluation step is based on the second clause:

$$\begin{array}{ll} & \texttt{gcd(36,116)} \\ \rightsquigarrow & (\texttt{gcd(n \% m, m)}, [\texttt{m} \mapsto 36, \texttt{n} \mapsto 116]) \end{array}$$

The expression `gcd(n % m, m)` will then be further evaluated using the bindings for `m` and `n`. The next evaluation steps evaluate the argument expression `(n % m, m)` using the bindings:

$$\begin{array}{ll} & (\texttt{gcd(n \% m, m)}, [\texttt{m} \mapsto 36, \texttt{n} \mapsto 116]) \\ \rightsquigarrow & \texttt{gcd(116 \% 36, 36)} \\ \rightsquigarrow & \texttt{gcd(8,36)}, \end{array}$$

The evaluation continues evaluating the expression `gcd(8,36)` and this proceeds in the same way, but with different values bound to `m` and `n`:

$$\begin{array}{ll} & \texttt{gcd(8,36)} \\ \rightsquigarrow & (\texttt{gcd(n \% m, m)}, [\texttt{m} \mapsto 8, \texttt{n} \mapsto 36]) \\ \rightsquigarrow & \texttt{gcd(36 \% 8, 8)} \\ \rightsquigarrow & \texttt{gcd(4,8)} \end{array}$$

The evaluation will in the same way reduce the expression `gcd(4,8)` to `gcd(0,4)`, but the evaluation of `gcd(0,4)` will use the first clause in the declaration of `gcd`, and the evaluation terminates with result `4`:

$$\begin{array}{ll} & \texttt{gcd(4,8)} \\ \rightsquigarrow & \cdots \\ \rightsquigarrow & \texttt{gcd(0,4)} \\ \rightsquigarrow & (\texttt{n}, [\texttt{n} \mapsto 4]) \\ \rightsquigarrow & 4 \end{array}$$

Note that different bindings for `m` and `n` occur in this evaluation and that all these bindings have disappeared when the result of the evaluation (that is, `4`) is reached.

1.10 Free-standing programs

A free-standing program contains a *main* function of type:

```
string[] -> int
```

preceded by the *entry point attribute*:

```
...
[<EntryPoint>]
let main (param: string[]) =
...
```

The type `string[]` is an array type (cf. Section 8.10) and the argument `param` consists of k strings (cf. Section 2.3):

$$param\,\texttt{.[0]},\ param\,\texttt{.[1]},\ \ldots,param\,\texttt{.[}k-1\texttt{]}$$

The following is a simple, free-standing "hello world" program:

```
open System;;
[<EntryPoint>]
let main(param: string[]) =
  printf "Hello %s\n" param.[0]
  0;;
```

It uses the `printf` function (cf. Section 10.7) to make some output. The zero result signals normal termination of the program. The program source file `Hello.fsx` compiles to an `exe`-file using the F# batch compiler:

```
fsc Hello.fsx -o Hello.exe
```

and the program can now be called from a command prompt:

```
>Hello Peter
Hello Peter

>Hello "Sue and Allan"
Hello Sue and Allan
```

Using the `fsc` command requires that the directory path of the F# compiler (with file name `fsc.exe` or `Fsc.exe`) is included in the `PATH` environment variable.

Summary

The main purpose of this chapter is to familiarize the reader with some of the main concepts of F# to an extent where she/he can start experimenting with the system. To this end, we have introduced the F# notions of values, expressions, types and declarations, including recursive function declarations.

The main concepts needed to explain the meaning of these notions are: integers and floating-point numbers, bindings and environments, and step by step evaluation of expressions.

Exercises

1.1 Declare a function `g: int -> int`, where $\texttt{g}(n) = n + 4$.

1.2 Declare a function `h: float * float -> float`, where $\texttt{h}(x, y) = \sqrt{x^2 + y^2}$. Hint: Use the function `System.Math.Sqrt`.

1.3 Write function expressions corresponding to the functions `g` and `h` in the exercises 1.1 and 1.2.

1.4 Declare a recursive function `f: int -> int`, where

$$\texttt{f}(n) = 1 + 2 + \cdots + (n-1) + n$$

for $n \geq 0$. (Hint: use two clauses with `0` and `n` as patterns.)
State the recursion formula corresponding to the declaration.
Give an evaluation for $\texttt{f}(4)$.

1.5 The sequence $\mathrm{F}_0, \mathrm{F}_1, \mathrm{F}_2, \ldots$ of Fibonacci numbers is defined by:

$$\begin{aligned} \mathrm{F}_0 &= 0 \\ \mathrm{F}_1 &= 1 \\ \mathrm{F}_n &= \mathrm{F}_{n-1} + \mathrm{F}_{n-2} \end{aligned}$$

Thus, the first members of the sequence are $0, 1, 1, 2, 3, 5, 8, 13, \ldots$.
Declare an F# function to compute F_n. Use a declaration with three clauses, where the patterns correspond to the three cases of the above definition.
Give an evaluations for F_4.

1.6 Declare a recursive function `sum: int * int -> int`, where

$$\texttt{sum}(m, n) = m + (m+1) + (m+2) + \cdots + (m + (n-1)) + (m+n)$$

for $m \geq 0$ and $n \geq 0$. (Hint: use two clauses with `(m,0)` and `(m,n)` as patterns.)
Give the recursion formula corresponding to the declaration.

1.7 Determine a type for each of the expressions:

```
(System.Math.PI, fact -1)
fact(fact 4)
power(System.Math.PI, fact 2)
(power, fact)
```

1.8 Consider the declarations:

```
let a = 5;;
let f a = a + 1;;
let g b = (f b) + a;;
```

Find the environment obtained from these declarations and write the evaluations of the expressions `f 3` and `g 3`.