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
```

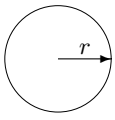
The identifier `it` is now bound to the integer value `2500`, and this identifier can also be used in expressions:

```
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:

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

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:

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

Computation of arguments

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

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

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*.