1 Quick-start Guide

Programming is the art of solving problems by writing a program to be executed by a computer. For example, to solve the following problem,

Problem 1.1

What is the sum of 357 and 864?

we have written the program shown in Listing 1.1.

Listing 1.1 quickStartSum.fsx:
A script to add 2 numbers and print the result to the console.

```
1 let a = 357
2 let b = 864
3 let c = a + b
4 do printfn "%A" c
```

```
1 $ fsharpc --nologo quickStartSum.fsx && mono quickStartSum.exe
2 1221
```

In the box the above, we see our program was saved as a script in a file called `quickStartSum.fsx`, and in the console we executed the program by typing the command `fsharpc --nologo quickStartSum.fsx && mono quickStartSum.exe`. The result is then printed in the console to be `1221`. Here, as in the rest of this book, we have used the optional flag `--nologo`, which informs `fsharpc` not to print information about its version etc., thus making the output shorter. The `&&` notation tells the console to first run the command on the left, and if that did not report any errors, then run the command on the right. This could also have been performed as two separate commands to the console, and throughout this book we will use the above shorthand when convenient.

To solve the problem, we made program consisting of several lines, where each line was an *expressions*. The first expression, `let a = 357`, in line 1 used the *`let` keyword* to *bind* the value 357 to the name `a`. This is called a *let-binding* and makes the name synonymous with the value. Another notable point is that F# identifies 357 as an *integer number*, which is F#'s preferred number type, since computations on integers are very efficient, and since integers are very easy to communicate to other programs. In line 2 we bound the value 864 to the name `b`, and to the name `c` we bound the result of evaluating the sum `a + b` in line 3. Line 4 is a *do-binding*, as noted by the keyword *`do`*. The `do`-bindings are also sometimes called *statements*, and the `do` keyword is optional in F#. Here the value of `c` was printed to the console followed by a newline with the *`printfn` function*. A function in F# is an entity that takes zero or more arguments and returns a value. The function `printfn` is very special, since it can take any number of arguments and returns the special value *"()"* which has type *`unit`*. The `do` tells F# to ignore this value. Here `printfn` has been used with 2 arguments: `"%A"` and `c`. Notice that in contrast to many other languages,

· expression
· let@`let`
· keyword
· binding
· let-binding
· integer number
· do-binding
· do@`do`
· statement
· printfn@`printfn`
· function
· ()@`()`
· unit@`unit`

F# does not use parentheses to frame the list of arguments, nor does it use commas to separate them. In general, the `printfn` function always has 1 or more arguments, and the first is a *format string*. A *string* is a sequence of characters starting and ending with double quotation marks. E.g., `let s = "this is a string of characters"` binds the string `"this is..."` to the name `s`. For the `printfn` function, the format string may be any string, but if it contains format character sequences, such as `%A`, then format character sequence are replaced by the arguments to `printfn` which follows the format string. The format string must match the value *type*, that is, here `c` is of type integer, whereas the format string `%A` matches many types.

· format string
· string
· type

Types are a central concept in F#. In the script 1.1 we bound values of integer type to names. There are several different integer types in F#, here we used the one called `int`. The values were not *declared* to have these types, instead the types were *inferred* by F#. Typing these bindings line by line in an interactive session, we see the inferred types as shown in Listing 1.2.

· type declaration
· type inference

Listing 1.2: Inferred types are given as part of the response from the interpreter.

```
> let a = 357;;
val a : int = 357

> let b = 864;;
val b : int = 864

> let c = a + b;;
val c : int = 1221

> do printfn "%A" c;;
1221
val it : unit = ()
```

The interactive session displays the type using the *`val`* keyword followed by the name used in the binding, its type, and its value. Since the value is also responded, the last `printfn` statement is superfluous. However, **it is ill-advised to design programs to be run in an interactive session, since the scripts need to be manually copied every time it is to be run, and since the starting state may be unclear.** Notice that `printfn` is automatically bound to the name *`it`* of type `unit` and value *"()"*. F# insists on binding all statements to values, and in lack of an explicit name, it will use `it`. Rumor has it that `it` is an abbreviation for "irrelevant".

· val@`val`
Advice
· it@`it`
· ()@`()`

The following problem,

Problem 1.2
What is the sum of 357.6 and 863.4?

uses *decimal point* numbers instead of integers. These numbers are called *floating point numbers*, and their internal representation is quite different to integer numbers used previously. Likewise, algorithms used to perform arithmetic are quite different from integers. Now the program would look like Listing 1.3.

· decimal point
· floating point number

**Listing 1.3 quickStartSumFloat.fsx:
Floating point types and arithmetic.**

```
1 let a = 357.6
2 let b = 863.4
3 let c = a + b
4 do printfn "%A" c
```

```
1 $ fsharpc --nologo quickStartSumFloat.fsx && mono
    quickStartSumFloat.exe
2 1221.0
```

On the surface, this could appear as an almost negligible change, but the set of integers and the set of real numbers (floats) require quite different representations in order to be effective on a computer, and as a consequence, the implementation of their operations, such as addition, are very different. Thus, although the response is an integer, it has type `float` which is indicated by `1221.0` and which is not the same as `1221`. F# is very picky about types, and generally does not allow types to be mixed, as demonstrated in the interactive session in Listing 1.4.

Listing 1.4: Mixing types is often not allowed.

```
> let a = 357;;
val a : int = 357

> let b = 863.4;;
val b : float = 863.4

> let c = a + b;;
  let c = a + b;;
  ------------^

stdin(4,13): error FS0001: The type 'float' does not match the
  type 'int'
```

We see that binding a name to a number without a decimal point is inferred to be an integer, while when binding to a number with a decimal point the type is inferred to be a float, and that our attempt of adding an integer and floating point value gives an error. The *error message* contains much information. First, it states that the error is in `stdin(4,13)`, which means that the error was found on standard-input at line 4 and column 13. Since the program was executed using `fsharpi quickStartSumFloat.fsx`, here standard input means the file `quickStartSumFloat.fsx` shown in Listing 1.3. The corresponding line and column are also shown in Listing 1.4. After the file, line, and column number, F# informs us of the error number and a description of the error. Error numbers are an underdeveloped feature in Mono and should be ignored. However, the verbal description often contains useful information for *debugging*. In the example we are informed that there is a type mismatch in the expression, i.e., since `a` is an integer, F# expected `b` to be one too. Debugging is the process of solving errors in programs, and here we can solve the error by either making `a` into a float or `b` into an int. The right solution depends on the application.

· error message

· debugging

F# is a functional first programming language, and one implication of this is that names have a *lexical scope*. A scope is the lines in a program where a binding is valid, and lexical scope means that to find the value of a name, F# looks for the value in the above lines. Furthermore, at the outermost level, rebinding is not allowed. If attempted, then F# will return an error as shown in Listing 1.5.

· lexical scope

Listing 1.5 quickStartRebindError.fsx:
A name cannot be rebound.

```
let a = 357
let a = 864
```

```
$ fsharpc --nologo -a quickStartRebindError.fsx

quickStartRebindError.fsx(2,5): error FS0037: Duplicate
  definition of value 'a'
```

However, if the same code is executed in an interactive session, then rebinding does not cause an error, as shown in Listing 1.6.

Listing 1.6: Names may be reused when separated by the lexeme ";;".

```
> let a = 357;;
val a : int = 357

> let a = 864;;
val a : int = 864
```

The difference is that the *";;" lexeme* is used to specify the end of a *script-fragment*. A lexeme is a letter or word, which F# considers as an atomic unit. Script-fragments may be defined both in scripts and in interactive mode, and rebinding is not allowed at the outermost level in script-fragments. Even with the ";;" lexeme, rebinding is not allowed in compile-mode. In general, **avoid rebinding of names.**

· ;;@;;
· lexeme
· script-fragment
Advice

In F#, *functions* are also values, and we may define a function `sum` as part of the solution to the above program, as shown in Listing 1.7.

· function

Listing 1.7 quickStartSumFct.fsx:
A script to add 2 numbers using a user-defined function.

```
let sum x y = x + y
let c = sum 357 864
do printfn "%A" c
```

```
$ fsharpc --nologo quickStartSumFct.fsx && mono
  quickStartSumFct.exe
1221
```

Functions are useful to *encapsulate* code, such that we can focus on the transformation of data by a function while ignoring the details on how this is done. Functions are also useful for code reuse, i.e., instead of repeating a piece of code in several places, such code

· encapsulation

can be encapsulated in a function and replaced with function calls. This makes debugging and maintenance considerably simpler. Entering the function into an interactive session will illustrate the inferred type the function `sum` has: `val sum : x:int -> y:int -> int`. The "`->`" is the mapping operator in the sense that functions are mappings between sets. The type of the function `sum`, should be read as `val sum : x:int -> (y:int -> int)`, that is, `sum` takes an integer and returns a function, which takes an integer and returns an integer. This is an example of a higher-order function.

Type inference in F# may cause problems, since the type of a function is inferred based on the context in which it is defined. E.g., in an interactive session, defining the sum in one scope on a single line will default the types to integers, F#'s favorite type. Thus, if the next script-fragment uses the function with floats, then we will get an error message as shown in Listing 1.8.

Listing 1.8: Types are inferred in blocks, and F# tends to prefer integers.

```
val sum : x:int -> y:int -> int

> let c = sum 357.6 863.4;;
  let c = sum 357.6 863.4;;
  ------------^^^^^

stdin(3,13): error FS0001: This expression was expected to
   have type
    'int'
but here has type
    'float'
```

A remedy is to define the function in the same script-fragment as it is used, such as shown in Listing 1.9.

Listing 1.9: Type inference is per script-fragment.

```
> let sum x y = x + y
- let c = sum 357.6 863.4;;
val sum : x:float -> y:float -> float
val c : float = 1221.0
```

Alternatively, the types may be explicitly stated as shown in Listing 1.10.

Listing 1.10: Function argument and return types may be stated explicitly.

```
> let sum (x : float) (y : float) : float = x + y;;
val sum : x:float -> y:float -> float

> let c = sum 357.6 863.4;;
val c : float = 1221.0
```

The function `sum` has two arguments and a return type, and in Listing 1.10 we have specified all three. This is done using the "`:`" lexeme, and to resolve confusion, we must use parentheses around the arguments, such as `(y : float)`, otherwise F# would not be able to understand whether the type annotation was for the argument or the return value.

Often it is sufficient to specify just some of the types, since type inference will enforce the remaining types. E.g., in this example, the "`+`" operator is defined for identical types, so specifying the return value of `sum` to be a float implies that the result of the "`+`" operator is a float, and therefore that its arguments must be floats, and finally then that the arguments for `sum` must be floats. However, in this book we advocate the following advice: **specify types unless explicitly working with generic functions.** Advice

In this chapter, we have scratched the surface of learning how to program by concentrating on a number of key programming concepts and how they are expressed in the F# language. In the following chapters, we will expand the description of F# with features used in all programming approaches.