Functional Programming Principles

by

Hifza Khalid

Rubab Zahra Sarfraz

Maira Muneer

Anmol Yousaf

Arooj Perzada

Department of Computer Science and Engineering
UNIVERSITY OF ENGINEERING AND TECHNOLOGY,
Lahore.

March 31, 2014

Chapter 1

Introduction to Functional Programming

1.1 What is Functional Programming?

In computer science, Functional Programming (FP) is a paradigm that models computations as mathematical evaluations and is therefore also known as declarative programming. Programs written in a functional language are immutable and have no side effects. The output of the program depends only on the input and has nothing to do with the state of the program. Thus FP enables concurrency. Higher Order Functions, modularization and recursion are other very important features of FP which will be explained later.

1.2 Programming in Haskell

To explain the ideas used in FP, we need an appropriate language. For this purpose, we will use Haskel because it covers all the concepts of FP and its syntax is inspired from mathematical notation. Besides, it has many other features like lazy evaluation, pattern checking and type system which we find interesting. Above and beyond these considerations, programming in Haskell is fun.

We'll be using GHCi (Glasgow Haskell Compiler Interpreter) throughout this paper. It can compile Haskell scripts (usually with an .hs extension) and also has an interactive mode where we can load these functions from scripts and call them directly to get immediate results.

1.2.1 Infix Notation

The convention of placing the operator between the operands is known as *infix notation.* The purpose of using this notation is that it is similar to the mathematical notation and thus isn't confusing. For example,

```
3 * (-3)
-9
```

1.2.2 Naming Objects

In Haskell, one of the simplest abstraction is *let* which is used to name objects. Typing

```
let a = 1
```

in GHCi causes Haskell to associate the value of a with 1 and thus whenever we use the identifier a in our expressions, it gets evaluated to 1.
Some other examples using "let" are

```
let length = 10
let breadth = 5
let area = length * breadth
let perimeter = 2 * (length + breadth)
```

```
area
```
50

```
perimeter
```
30

1.2.3 Compound Procedures

Thus far we know that a powerful programming language must have the following elements

- Numbers and arithmetic operations as primitive data and procedures.
- A mechanism to associate names with object to provide a limited means of abstraction.

Now, let us introduce procedure definitions. A procedure has an identifier, parameters and a body.

```
add x y = x + y
```

In the procedure defined above, add is the "procedure name", x and y are "parameters" (a procedure can have any number of arguments) and the expression proceeding the equality sign is the "body" of the procedure.

1.2.4 Strong Type System

One of the greatest advantage of Haskell over any other functional language is its powerful type system and type inference. In Haskell, every expression's type is known at compile time. Whenever you use procedures with inappropriate data types, your program gives you compile time errors and hence doesn't crash later on. For example,

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

In the example shown above, we specified the type of the input and output of factorial function. Thus, whenever we give an input that is not *Int* type, compiler yells at us. This feature makes Haskell programs safe and concise. No error check can give you more confidence than Haskell's automatic type system.

1.2.5 Substitution Model in Haskell

An expression in Haskell is evaluated using substitution rule, that is when the function is called with some arguments, its formal parameters get bound to the corresponding arguments and then these formal parameters are replaced by the corresponding arguments in the body of that function. For example, in the evaluation of

```
⇒ square x = x * x
```

```
⇒ square 3
```

`x` gets bound to `3` and then replaced by `3` in the body of `square x` which will yield

```
(3 * 3)
```

1.2.6 Pattern Matching

Pattern matching is used to specify different patterns for a function to which some data should conform and to deconstruct data according to those patterns. This not only makes our program easier to code but also more readable. For example,

```
check :: [a] -> String
check [] = "Empty"
check (x:xs) = "Contains Elements"
```

Here `check` is a function which tells us if the input list is empty or contains elements. When we call this function with a list, it starts checking it against the patterns sequentially from top to bottom. If the data conforms to one of the patterns then it gets evaluated accordingly. Note that the last pattern of the `check` function takes in a list as an input which can be of any data type. This is known as *catchall pattern* because due to this pattern this function works for a list of any length.

1.2.7 Lazy Evaluation

The order of evaluation that Haskell uses is Lazy. We can define it as:

Lazy Evaluation = non-strictness + sharing

where non-strictness simply means that arguments of functions are not evaluated until needed and sharing means that arguments to a function are only evaluated once; the calculated values are "shared" between their users.

Lazy evaluation is also called *Call-by-need*. Its opposite is applicative/eager evaluation, in which the function is applied to the arguments after evaluating each one of them from left to right.

One example of language using eager evaluation technique is Scheme.

Lazy evaluation uses thunks for this purpose. Thunk is simply a piece of code to perform a delayed computation. Haskell creates thunks of arguments to a function which are then evaluated to get the values of arguments on demand.

Advantages of Lazy Evaluation

1. We can deal with infinite lists. Example:

   ```
   take 3 (repeat 5)
   ```

   will give `[5,5,5]`. `(repeat 5)` generates an infinite list but since we require only first three values the program will terminate by giving first three values.

2. Those expressions are not evaluated which are not required. For example,

   ```
   proc a b = if a<b
                then a+b
                else a-b
   ```

   If `a<b` is true, only expression `a+b` will be evaluated. Since `a-b` is not required, it will not get evaluated.

3. Owing to the ability of sharing, each expression is evaluated either `0` or `1` time which makes program efficient to some extent.

Disadvantages of Lazy Evaluation

1. Since thunks are created on heap, extra memory is used even if the expression is to be never used.

1.2.8 Recursion

Recursion is a way of defining a function by calling the same function in its definition until you reach a base case which has to be explicitely defined in the definition of the function. For example,

```
myLast :: [a] -> a
myLast [] = error "Empty List"
myLast (x:[]) = x
myLast (x:xs) = myLast xs
```

The function defined above is to extract the last element of a list. When the list is not a singleton or an empty list, we call the same function with the `tail` of the list. This process continues until the base case for a singleton list is satisfied and the function simply returns that remaining one element in the list.

As you can see here, pattern matching is very useful in defininig recursive functions.

1.2.9 List Comprehensions

List Comprehensions are a way to transform, filter and combine lists. These are quite similar to set comprehensions used in Mathematics. To understand the syntax of a list comprehension, consider typing the following expression in GHCi

```
ghci> [x*2 | x <- [1..10], x*2 >= 12]
```

It gets evaluated to

```
[12,14,16,18,20]}
```

Here, in the part of the list comprehension after the vertical pipe (`|`), `[1..10]` is the list from which each element is to be drawn. `x` is the variable that gets bound to each element of the list `[1..10]` and expression after comma is the predicate. The part before the vertical pipe is the output which shows how each element will be reflected in the output list.

1.2.10 Pattern Guards

Patter Guards are used to check if certain properties are satisfied on the arguments of a function. They are more like an `if` expression but are much more readable and play nicely when we have to check the arguments for a number of properties as compared to using nested `if` expressions. For example,

```
max' :: (Ord a) => a -> a -> a
max' a b
 | a <= b = b
 | otherwise = a
```

This function introduces something new in the type declaration section that is, the symbol =>. Everything before this symbol is known as the *class constraint.* Here, the type declaration can be read as, this function takes in two values that can be of any type but they must be an instance of the *Order* class (a type class that describes a certain behavior for its instances).

The example shown above is the definition of a function using pattern guards. The symbol (`|`) is known as a *guard*. If the the predicate after a guard evaluates to `True` then the function returns the value of the expression after the equality sign, else it drops through to the next guard and the process repeats. Usually the last guard is `otherwise` which catches everything.

1.2.11 Making Our Own Types

In Haskell, we are not restricted to use only the in-built data types. We can make our own data types that make our programs easier to design, write and understand. This also allows us to use the related parts of a data in a more convenient way.

One way to make our own data type is using the keyword `data`. The code shown below is to make a recursive data type that is, a data type that contains an element of the same data type inside.

```
data NestedList a = Elem a | List [NestedList a]
```

Here `NestedList a` is the identifier of the new data type which contains an element of type `a` in its field. The expressions after the equality sign, separated by vertical pipes (`|`) are the value constructors (possible values that the data type can have). Think of these value constructors as functions and their fields as the parameters of the function. An example of a function defined using this new data type is shown below:

```
flatten :: NestedList a -> [a]
flatten (Elem a) = [a]
flatten (List []) = []
flatten (List (x:xs)) = flatten x ++ flatten (List xs)
```

1.2.12 `where` clause in Haskell

Sometimes we have to calculate the same value over and over again in our functions. This not only increases the size of our code but is also inefficient. However, Haskell provides us with a functionality to avoid doing these computations again and again using the keyword `where`. Let us look at an example.

```
bmiTell :: Double -> Double -> String
bmiTell weight height
 | bmi <= 18.5 = "You're skinny"
 | bmi <= 25.0 = "You're supposedly normal"
 | otherwise = "You're fat"
 where bmi = weight / height ^ 2
```

The function defined in the `where` block is visible throughout the `bmiTell` function. We can define multiple variables and functions in the `where` section of a function but these wouldn't be shared across function bodies of different patterns. For that, we'll have to define those variables and functions globally.

1.2.13 `let...in` Expression

`let` expressions are very similar to `where` bindings. However, `let` expressions are...well expressions (i.e. they have a value) whereas `where` bindings are not. Hence we can use `let` expressions anywhere in our code. They take the following form

```
let <bindings> in <expression>
```

We can use `let` expressions in our functions as well but unlike `where` bindings, `let` bindings are not visible throughout the functions. They are very much local and are visible only throughout the `let` expression. For example,

```
cylinder :: Double -> Double -> Double
cylinder r h =
   let sideArea = 2 * pi * r * h
       topArea = pi * r ^ 2
   in sideArea + 2 * topArea
```

1.2.14 Tuples

Tuples are used to store multiple values in a single value. The are quite different from lists. The major differences between the two are listed below:

1. Tuples are heterogeneous whereas lists are not. Thus, tuples can be used whenever we want to store elements of different data types for example, the attributes of an object.

2. Tuples have a fixed length unlike lists. Thus it makes sense to use tuples whenever we know the length of the list ahead of time. Due to this quality of tuples, we can also use them to return multiple values in a function.

Tuples are surrounded by parentheses and their elements are separated by commas. They can be of any length for example, *pair* (a tuple of length 2), *triple* (a tuple of length 3), *4-tuple* and so forth.

Pairs are very commonly used in Haskell, which provides us with two in-built functions to retrieve its values. These functions are `fst` and `snd`. The former returns us the first element of the pair whereas the latter returns us the second element of a pair as shown below.

```
ghci> fst (8, 11)
8

ghci> snd ("Wow", False)
False
```

1.2.15 Ranges

Ranges are basically lists that contain elements in some particular arithmetic sequence. For instance, if we want to build a list of integers from `1` to some specific number `n`, we can simply write it as `[1..n]`. Ranges can also be used to make lists that involve steps i.e. we can generate a list of even number by writing `[2,4..n]`. But we can specify only one step in our range.

Ranges can also be used to produce infinite lists by not giving the upper limit for example, `[1,2..]`.

So ranges are really helpful in different functions as we do not have to fill the required list explicitly. Two examples that use ranges are given below. First one uses infinite range and second one finite range.

```
take 24 [13,26..]
```

It computes first twenty-four multiples of `13`.

```
primeR l u | x == [] = []
        | isPrime (head x) = (head x) : (primeR (l + 1) u)
        | otherwise = (primeR (l + 1) u)
         where x = [l..u]
```

This function uses a range that has its lower limit specified by input `l` and upper limit by `u`.

1.2.16 Multiple Ways to Define a Function

Haskell provides us with a variety of in-built functions which we can use to define our functions in a number of different ways. It totally depends on the user which method goes with his requirements or which way he prefers. Below is the example of a factorial function defined in different ways.

1. **Using Pattern Matching**

```
factorial1 :: Int -> Int
factorial1 0 = 1
factorial1 n = n * factorial1 (n  1)
```

2. **Using Patten Matching and Patten Guards**

```
factorial2 :: Int -> Int
factorial2 n
 | n == 0 = 1
 | otherwise = n * factorial2 (n - 1)
```

3. **Using Patten Matching and Ranges**

```
factorial3 :: Int -> Int
factorial3 n = product [1..n]
```

4. **Using an if-else Expression**

```
factorial4 n = if n == 1
then 1
else (n * factorial4 (n - 1))
```

Chapter 2

Introduction to F#

F# is a powerful "functional first" language where functional programming is the first option for solving complex problems. F# differs from many functional languages in that it embraces imperative and object-oriented (OO) programming where necessary. F# is a multi-paradigm .NET language. The first paradigm is *functional programming*, which is becoming more important for some properties including the fact that functional codes are easier to test. The second paradigm is *object oriented programming* and the third paradigm is *language oriented programming*, a programming style in which programmer solves a problem in DSL *(Domain Specific Language)* or creates a language of one or more specific domains and solves the problem in that domain. A "programming domain" tells us the use of a programming language .

2.1 What is .NET framework?

.Net framework is developed by Microsoft. It is a software that runs on Microsoft Windows. It provides language interoperability i.e. each language can use code written in other languages.

2.2 Getting to know F#

2.2.1 Syntax

There are two types of syntax in F#: *Lightweight* and *verbose.* The default syntax is lightweight. Lightweight is shorter, but uses indentation. Its advantage over verbose syntax is, you don't have to use many extra keywords like "in", "for" etc. A small description of both is given below.

Lightweight syntax

The F# language uses simplified, indentation-aware syntactic constructs known as lightweight syntax.

To enable the lightweight syntax enter the following command

```
#light;;
```

Lightweight syntax applies to all the major constructs of the F# syntax. In the next example, the code is incorrectly aligned. The declaration starts in the first line and continues to the second and subsequent lines, so those lines must be indented to the same column under the first line:

```
let Derivative f x =
                   let p1 = f (x - 0.05)
let p2 = f (x + 0.05)
                  (p2 - p1) / 0.1
```

The following shows the correct alignment:

```
let Derivative f x =
          let p1 = f (x - 0.05)
          let p2 = f (x + 0.05)
          (p2 - p1) / 0.1
```

Verbose Syntax

There is an alternative syntax that does not use indentation; it is called "verbose" syntax.With verbose syntax, you are not required to use indentation, and whitespace is not significant, but the downside is that you are required to use many extra keywords, including things like:

1. "in" keywords after every "let" and "do" binding.
2. "begin"/ "end" keywords for code blocks such as if-then-else.
3. "done" keywords at the end of loops.

Here is an example:

```
#indent off
        let f =
      let  x = 1 in
if x = 2 then
begin a end else begin
b
end
#indent on
```

2.3 Literals

In a programming language, literals are fixed values or text that provides information about a variable. For example, an integer could be represented as hexadecimal, octal or binary. Literals provide us the facility to do so. They represent a constant value. F# has a large set of literals. Some literals are shown in the table below.

Example	F# Type	Description
0x22	Int/int32	An integer as a hexadecimal
0o42	Int/int32	An integer as an octal
0b10010	Int/ int32	An integer as a binary
34y	Sbyte	A signed byte
true, false	Bool	A Boolean
34uy	Byte	An unsigned byte

For example:
If I define an integer as hexadecimal.

```
> let var = 0x22;;
val var : int = 34
```

2.4 Lambda Functions

To define a lambda expression in F#, we use *fun* keyword. The arguments of a function are separated by spaces and body is separated by left arrow (->). For Example:

```
fun x y -> x + y
```

Here function does not have any name. In F# , we call it anonymous function, lambda function or just lambdas.

2.5 Declaring Variables and Functions

The most common keyword in F# is *let.* It is used for the declaration of variables and functions. Both variables and functions use same syntax. For example:

```
let x = 4 ; ;
```

This declares a variable x and assigns value 4 to it. Functions can also be declared similarly. For example:

```
let square x = x * x ; ;
```

Square is the name of a function and it takes one argument `x`. This function can also be defined using fun keyword.

```
let square = fun x -> x*x
```

The syntax for declaring functions or variables is indistinguishable in F#. Functions can also be passed as values .So F# treats both of them similarly.

The above function can be executed as follows:

```
let y = square 4 ; ;
```

This assigns `y` the return value of the function, which in this case is `16`. Unlike many other languages, F# functions do not have an explicit keyword to return a value. Instead, the return value of a function is simply the value of the last statement executed in the function.

In F# we call some functions as *partial* or *curried* functions. Partial or curried functions are those that can be applied partially, which means that incomplete number of arguments can be passed to those functions. Unlike in other languages, it will not give an error rather it will return a function of the remaining number of arguments. Like

```
>let add a b = a + b;;
val add = a: int ->b: int -> int

>add 4;;
val it : (int -> int) = <fun:it@15-1>

>((add 4) 5);;
val it : int = 9
```

2.6 Operators

F# supports infix (in which operator appears in the middle of operands) and prefix operators (in which operator appears before the operands). F# provides us with the facility of operator overloading that is, we can redefine built-in operators. We can also define new operators in F#. For example, we can redefine + operator for concatenation of lists using let keyword.

```
>let (+) a b = a @ b;;
val ( + ) : a:'a list -> b:'a list -> 'a list

>[``re''] + [``defined''];;
val it : string list = [re; defined]
```

The parentheses are placed across the operator while defining it so that, the compiler does not confuse it with actual operator and takes it as a name of the operator. The drawback of redefining a built-in operator is that we'll no longer be able to use it for any other data type. Like in the above example, we can't use + operator for adding two integers now.

Unlike functions, operators are not values and we can't pass them to any function as arguments. But if we put parentheses around an operator, the compiler treats it as a function and thus enables us to pass it as an argument as well as return it as the value of the function. We can treat operators surrounded with parentheses as simple functions. For example,

```
>(+) 2 3;;
val it : int = 5
```

Here (+) is a function called with arguments `2` and `3`.

2.7 Type Inference

Most programming languages require that you specify type information for function declaration. However, F# infers type information. For example, for the function declaration below:

```
let square x = x * x ; ;
```

F# knows that `square` takes a parameter named `x` and returns `x*x`. Many primitive types support multiplication(*) operator (such as byte or double) however, for arithmetic operations, F# infers the type *int* by default.

Although F# can typically infer type information but sometimes we must provide explicit type in F# code. For example:

```
> let strcat (x : string) y = x + y;;
Val strcat : string -> string -> string
```

Because `x` is stated to be of type string, and the only version of the + operator that accepts a left-hand argument of type string also takes a string as the right-hand argument, the F# compiler infers that the parameter `y` must also be a string. Thus, the result of `x+y` is the concatenation of the strings. Without the type annotation, the F# compiler would not have known which version of the + operator was intended and would have assumed int data type by default.

The process of type inference also applies automatic generalization to declaration. For example, code can be used for many different data types.

```
>let temp x y = y;;
val  temp : a -> b -> b

>temp 5 value;;
val it : string = value
>temp throwaway 10.0;;
val it : float = 10.0
>temp 5 30;;
val it : int = 30
```

Note: Here “it” is an identifier. If an expression is left unnamed, by default, it is called “it”. “it” binds to any unnamed last executed expression.

For example:
If we execute “it” in the interactive window, it will give us the value `30` because value of last executed unnamed expression was `30`.

```
>it;;
val it : int = 30
```

2.8 Lists

In F# lists are defined by enclosing data in square brackets. Elements are separated by semicolon (`;`). In F# “cons” is represented by the symbol (`::`). Empty list is represented by square brackets (`[]`) . A few definitions of lists are shown below For example:

```
let lst = ``one'' :: []
let lst2 = ``two'' :: ``three'' :: ``four'' :: []
```

This way of defining lists is verbose. The simple and better way to define lists is to use semicolons. For example,

```
let lst = [``one'']
let lst1 = [ ``two''; ``three''; ``four'']
```

List concatenation is done by (`@`) operator

```
let lst3 = lst @ lst2
>lst3;;
val it : string list = [``one''; ``two'' ; ``three'' ; ``four'']
```

There are many built-in functions in F# for lists like `map`, `rev` etc.

```
let lst_rev = List.rev lst3
>lst_rev;;
val it : string list = [``four'' ; ``three'' ; ``two'' ; ``one'']
```

2.9 Pattern Matching

Pattern matching allows us to look at a value, test it against series of conditions and perform some computations if the condition is met. It is similar to `if...else` conditions in other languages and `cond` in scheme. One of the pattern used in F# is `match` expression, in which the pattern is what follows the pipe symbol.

```
match expression with
        | pattern1 -> result1
        | pattern2 -> result2
        | pattern3 -> result3
        | _ -> defaultResult
```

Each `|` defines a condition. The symbol `->` means if the pattern matches with the expression then the result corresponding to it will be executed. If the match is not found then the next pattern is tested. The "_" is a default pattern, which means it matches anything. If none of the above pattern matches the expression then the default result will be executed.
Here is an example of a recursive procedure using pattern matching.

```
Let rec fib n =
           match n with
                 | 0 -> 0
                 | 1 -> 1
                 | _ -> fib (n  1) + fib (n  2)
```

F# has shorthand syntax for writing pattern matching functions using function keyword.

```
Let getPrice = function
                 | ``apple'' -> 50
                 | ``orange'' -> 40
                 | ``banana'' -> 70
```

Although it appears that function does not take any parameter, but it has a type `string -> int` which means it take a single parameter of type string and returns an int.
F# can automatically bind values to identifiers if they match certain patterns. For example, in the function declaration below:

```
Let rec factorial = function
         | 0 | 1 -> 0
         | n -> n * factorial (n-1)
```

If we call `factorial` with `3` then `0` and `1` patterns will not match, but last pattern will match and it will bind the value `3` with `n`.

2.10 When Guard

There are some constraints in pattern matching. Suppose we want to write a function which works for positive integers only, using pattern matching. Then, if we try to write a function like this

```
>let temp n =
    match n with
    | 0 -> 1
    | n < 0 -> dosomething()
```

compiler will give the following error here.

```
stdin(1,1): error FS0010: Unexpected symbol '>' in interaction
```

when guard provides the facility to do this and helps us to get rid of these constraints. Rewriting above code

```
>let temp n =
    match n with
    | _ when n < 0 -> failwith something
    | 0 -> 1
    | _ -> 2
```

Now this function will give an error if we pass it a negative number as an argument.

Let's take another example, which explains advantages of "when guard".

Suppose we are writing a function which takes a pair and a number and checks whether the number is equal to the first element of the pair. If we try to write it without using "when guard"

```
>let equal pair num =
    match pair with
    |(num, _) -> true
    |_ -> false
```

Apparently, it should return `false` if we call it with `(1,2)` and `3`.

```
>equal (1,2) 3;;
val it : bool = true
```

But it will return `true` in every situation because F# does not care that `num` is already bound with `3` and binds `num` with `1`. `equal` takes `num` as a variable in pair and binds it with any value passed as a first element of the pair.

Rewriting above code using "when guard"

```
>let equal pair num =
    match pair with
    |(a, _) when a = num -> true
    |_ -> false

>equal (1,2) 3;;
val it : bool = false
```

2.11 Pattern Matching and Lists

In F# we work with lists using pattern matching and recursion because we can't separate head and tail of a list using `ifthenelse`. By using pattern matching, it is easy to separate head and tail of a list.

```
>let rec map func list =
    match list with
    | head :: rest -> func head :: map func rest
    | [] -> []
```

Another example,

```
>let rec findSequence l =
    match l with
    | [x; y; z] -> printfn ``Last 3 numbers in the list were %i %i %i''
                             x y z
    | head :: tail -> findSequence tail
    | [] -> ()
```

2.12 Tuples

A tuple is collection of values separated by comma. For example, (8 , "hello") is a tuple with the type `int*string`. Other examples are

Tuple of two integers.
`(1, 2)`

Triple of strings.
`(``one'', ``two'', ``three'')`

Tuple of unknown types.
`(a, b)`

Tuple that has mixed types.
`(``one'', 1, 2.0)`

Tuple of integer expressions.
`(a + 1, b + 1)`

We can use tuple patterns in `let` binding. For example,

```
Let (a , b) = (6 , 8);;
```

This binds values `a` and `b` at the same time. If we need only one element of the tuple, the underscore (_) can be used to avoid creating a new name for a variable that we do not need.

```
let (a, \_) = (1, 2);;
```

To access the first and second elements of tuples function fst ant snd can be used,

```
let c = fst (1, 2);;
let d = snd (1, 2);;
```

There is no built-in function in F# to access the third element of tuple. But we can easily write it as

```
let third( _,_,a)=a;;
```

Different elements of tuple can also be accessed by using pattern matching. Pattern matching on tuples is easy, because it uses the same syntax as used to declare tuple types.

```
let print tuple1 =
   match tuple1 with
    | (a, b) -> printfn "Pair %A %A" a b;;
```

Tuples provide a way to return multiple values from a function, as shown in the following example. This example performs integer division and returns the rounded result of the operation as a first member of a tuple and the remainder as a second member of the tuple.

```
let divRem a b =
   let x = a / b
   let y = a % b
   (x, y);;
```

Important points about Tuples

- A particular tuple type is a single object. When using them in functions they count as a single parameter.
- The order of the multiplication is important i-e `int*string` is not the same tuple type as `string*int`.
- The comma is the critical symbol that defines tuples, not the parenthesis. You can define tuples without the parentheses, although it can sometimes be confusing. In F#, if you see a comma, it is probably part of a tuple.