Functional Languages

Contents

1. Intro

What makes a language functional? The fact that "*functions are first class citizen of the language*". This means that functions can be passed as arguments to other functions, returned as values from other functions, and assigned to variables or data structures.

1.1. Some basics

- **Statements** are single line of code of type `statement`
- **Expressions** are entities that can be evaluated and have a type

Let's describe (not completely) the C grammar in BNF notation:

```
expr ::= constant
      |  (expr)
      |  x
      |  expr++
      |  ++expr
      |  expr--
      |  --expr
      |  expr = expr
      |  expr + expr
      |  expr - expr
      |  expr * expr
      |  expr / expr
      |  expr % expr
      |  expr << expr
      |  expr >> expr
      |  expr & expr
      |  expr | expr
      |  expr ^ expr
      |  ~ expr
      |  - expr
      |  ! expr
      |  expr && expr
      |  expr || expr
      |  expr == expr
      |  expr < expr
      |  expr <= expr
      |  expr > expr
      |  expr >= expr
      |  & expr
      |  * expr
      |  expr ? expr : expr
      |  f(expr1, ..., exprN)
      |  expr[expr]
      |  expr.x
```

```
constant ::= 0 | 1 | 2 | 3 | ...
           |  'a' | 'b' | ...
           |  1.0 | 2.45 | ...
           |  "string..."

block ::= statement ;
        |  statement ; block

cases ::= case constant: block
        |   case constant: block cases

statement ::= return expr
            |  if (expr) { block }
            |  if (expr) { block } else { block }
            |  for (statement ; expr ; expr ) { block }
            |  while (expr) { block }
            |  do { block } while (expr)
            |  switch (expr) { cases }
            |  break
            |  continue
            |  goto x
```

NOTE: this notation is called *Backus-Naur Form*.

It is used to describe context-free grammars and syntax of languages.

1.2. Reduction

Reduction is the evaluation of an expression. With this process every part of the expression is "reduced" to a smaller form until we get to a *ground value*. Let's take $1 + 2 * 4$ as example:

$$\overbrace{\overbrace{\underbrace{1+2}_{3} * 4}^{\text{int}}}^{\text{int}}$$

$$\underbrace{\phantom{\underbrace{1+2}_{3} * 4}}_{12}$$

On the lower brackets we can see the result of the evaluation of the arithmetical operations. From the upper brackets instead we can see that the reduction preserves the types of the expression.

1.3. Declaration, assegnation, binding

In imperative languages there are some core concepts like **declaration** and **assignment** or **initialiation**. So for example in C

```
int x;            // declaration
x = 7;            // assignment
int answer = 42;  // declaration with initialiation
```

In the context of functional languages, following the syntax of mathematics, we use only the last option, declaration with initialization, and we call it *binding*.

Binding answer to 42 in F# will be `let answer = 42`

1.4. Syntax vs Semantics... and types

Let's take for example this code `int x = &42`, a declaration with inizialitation in C.

This respects the grammar of the language because it is **syntactically correct**, in fact this can be described as `Type ID = expr`. We can also expand that expression in `expr = &expr`, which is the address extraction of a variable, where the internal `expr` is the constant expressions `42`. But speaking about **semantics** this is **wrong**!

To understand why this is incorrect we have to analyze the types of the expression. The address extraction operation transform types like that $\&\tau \rightsquigarrow \tau*$, so in our example it transforms `int` to an `int*`.

1.5. Characteristics of functional languages

Functional languages:

- have only expressions
- have variable definition
- have function definition
  - Note: in the C grammar described before is missing function definition

They have to offer the minimum to be Turing-complete, so a way of looping is needed. This structure is missing in this type of languages, but covered by *recursion*.

1.6. Syntactical vs Lexical

Lexicon: collection of words. In programming language is the set of keywords and values that can be used.

So, for example, −7 is different from `-expr` where the expression is 7. In `expr + expr` there are two expression and the operator + which is a keyword in the lexicon of the language.

1.7. Functional vs Imperative

We have to consider that "*to assing*" means to modify. This is why functional languages lacks of assignment operation, meaning anything that can modify data directly.

```
int x = 3;  // declaration with initialiation -> a BINDING in FL
x = 4;      // assignment -> MISSING in FL
```

1.8. Polymorphism

When the type of an expression is not important we can use some form of **polymorphism**. For example it could be **subtyping**, such as the possibility of interchange sub and base classes in OOP.

Another way to implement polymorphism is by **parametric polymorphism**, with techniques like Java Generics and C++ Templates (this is the way F# follows).

1.9. Language lazyness

A language is defined **lazy** if it defines things that are not already computer (Haskell), instead it's **strict** if evaluation occurs immediately.

1.10. Units

Unit `()` are special values that represent nothing. They are different from `void` or `null`. They are a type with only this value.

2. Functions

2.1. Function applications

The line in the C grammar that allow us to do function calls is $f(e_1, ..., e_n)$. In functional languages we use to call this *application* of the function and usually it is called just by writing the name of the function followed by the arguments.

The "functional grammar" for the application is `expr ::= expr expr`.

Take for example the function `(fun x -> x) 7`. This is the identity function, so from something it returns the exactly same thing. In this case types are not explicit, so the compiler create some "generic" *anonymous types*.

`(fun x -> x) 7`	F# function
$f : \text{'}a \to \text{'}a$	Anonymous type generated by the compiler
$f : \text{int} \to \text{int}$	Types by inferred the compiler using the type of the argument

Note: in application the left part must be an arrow (a function). We can say that lambdas create and applications remove arrows.

After the application, the type is inferred by the compiler. It understands that we are passing a integer, so the function becomes "*monomorphic at time of application*" over the type `int`.

2.1.1. Binding of parameters

```
let succ x = x + 1  // Syntactic sugar for this: let succ = fun x -> x + 1

let seven = f 6     // 6 is bounded to x then the function is computed
```

That 6 in the function applications replaces every occurency of `x` in the scope of the parameter.

2.1.2. Errors accepted by the grammar

If we look at the grammar for function application application (but not only in that) we can see that it is possible to create something that is syntactically correct but that will produce an error. For this we need to use the power of **types** and **type check** what we create with our grammar.

2.2. Currying

Currying is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument.

Let's see an example with integer addition. The uncurried form of this operation is

```
// uncurry_add : int * int -> int
// Syntactic sugar for uncurry_add = fun (x, y) -> x + y
let uncurry_add (x, y) = x + y

let result = uncurry_add (7, 8)
```

It accepts only pair of integers as arguments and after the application it returns the result.

In its curried form instead we have

```
// curry_add : int -> (int -> int)
// Syntactic sugar for curry_add = fun (x, y) -> x + y
let curry_add (x, y) = x + y

// partial_application : int -> int
```

```
let partial_application = curry_add 7

// final_application : int
let final_application = z 1

// total_application : int -> int -> int
let total_application = curry_add 7 1
```

Currying is in fact using more arrows in a function definition. With this form we can pass single integers as arguments and make "partial" applications of the function. It is automatically applied associativity on the left.

2.2.1. Function transformer

We can define functions that transform from uncurried version to the curried one and viceversa.

```
// curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
let curry f x y = f (x, y)

// uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
let uncurry f (x, y) = f x y
```

`curry` converts an uncurried function to a curried function. `uncurry` converts a curried function to a function on pairs. Let's see an example with the addition:

```
>>> curry uncurry_add 40 2
>>> 42

>>> uncurry curry_add (2, 40)
>>> 42
```

2.3. Shadowing and overloading

Overloading is a type of polymorphism, where different functions with the same name are invoked based on the data types of the parameters passed. This is not supported in F#!

Instead shadowing occurs when something declared within a certain scope has the same name as a variable declared in an outer scope. Let's see an example:

```
let a = 3
let b = a + 1
let a = "string"  // This shadows the int version
let c = a + 2     // This won't work because of types don't match
```

Every new *let bind* create shadows. *To rebind* is different from *to reassign* but in some cases it can be useful, for example to block the usage of an old value of the bind.

Another example:

```
let f x =               // First f
  let f x = x + 1       // This is a new f
  let f x = f (x + 1)   // It applies the last defined f
  x
```

2.3.1. Method dispatching

In OOP there is *dynamic dispatching* on method calls (Runtime Choose Methods). Methods are deferenced pointers of the virtual table of the object, in fact the call `object.method` emits a pointer and then jumps to it.

Overloading is instead *static dispatching*, because it uses different prototypes created in a way that allows the compiler to choose the right method based on the arguments at compile time.

2.4. Recursion

At the syntax level recursion is a function that calls itself, at the semantic level is having the symbol of the function itself inside the scope. In F# to enable recursiveness of a function we have to add ¶ec in the declaration.

```
let rec fact n =
  if n > 1
  then n * fact (n-1)
  else 1
```

2.4.1. Pattern matching

We can redefine our recursive function using *pattern matching*

```
let rec fact n =
  match n with
  | 0 | 1 -> 1
  | n     -> n * fact (n-1)
```

We use the syntax

```
match expr with
| Pattern -> expr
| Pattern -> expr
| ...
```

Following the Chomksy hierarchy Patterns are at the level of regular expressions.

All the branches must return the same type.

2.5. Higher order functions

We call *higher order functions* a function that takes functions as arguments.

2.6. Predicates

We call *predicates* a function where codomain is boolean, so something that returns only true or false.

2.7. η conversion

An *η conversion* is adding or dropping abstraction over a function.

From the first to the second is an *η reduction*, viceversa it is called *η expansion*

```
iter (fun x -> printf("%d") x) [1 .. 10]

iter (printf("%d") x) [1 .. 10]
```

2.7.1. Folding

In functional programming, **fold** (or *reduce*) is a family of higher order functions that process a data structure in some order and build a return value. For example summing all the elements of a list can be obtained from folding it.

An opposed family of function is the *unfold* family which create a data structure starting from a value and applying a function to it.

3. Types

In computer science we define **record** an abstract data type with only field. In languages we have implementation of records, for example structs in C or classes in OOP. Records are also referred as **product types**.

Unions instead are a type in which values may have different rappresentation or formats withing the same position in memory. They are also called **sum types**. They represent a choice. The most famous union type is *boolean*, that is either true or false.

In C we have `typedef` for creating an alias to an already existing type and `enum` to give constant names to integers. Then there are `struct` and `union`.

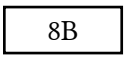
```c
struct S{
  int a;
  double b;
  char* c;
}
```

In memory this struct is

For every element of the struct a memory space is reserved.

```c
union S{
  int a;
  double b;
  char* c;
}
```

In memory this union is



It treats the same memory location in multiple ways.

In F# we have records and unions. Records are defined with the `type` keyword and unions with the `type` keyword and the `|` symbol.

```fsharp
// Record type
type Person = {
  name    : string
  surname : string
  age     : int
}
```

```fsharp
// Union type
type Color = Black | White | Yellow |
Blue

// Typedef (an alias)
type alias = int
```

4. Lists

We could define list in the classic recursive method `(node, next)`.

The classic `Struct` approach in C style is based on record. We define a list using a union type.

```
// Union type list of integers
type myList = Empty | NonEmpty of int*myList
```

In this way we define a list that is either an empty list or a non empty list. We use that `Empty` to mimic the classic `nullptr` terminator. Let's see the standard library polymorphic definition:

```
type `a list = [] | (::) of `a*`a list
```

With this definition `[]` and `::` are just names for the empty and non empty list. Just note that using parenthesis makes the `::` an infix operator, to define things in a more convenient way:

```
myList = NonEmpty(1, NonEmpty(2, Empty))

list = 1 :: 2 :: [] // Inferred type: int
```

4.1. Functions over lists

4.1.1. Length

We recursively calculate the lenght by add one until the last element.

```
let rec length list =
  match list with
  | []            -> 0
  | head :: tail  -> 1 + length tail
```

In this version we define `head` but we never use it. To "define" a placeholder for something we won't use there is _, so the second pattern becomes `_ :: tail -> 1 + length tail`

4.1.2. Insertion

To insert in the head we can just use the list constructor

```
insert_head = 0 :: int_list // OK
// int  :: int list
// `a   :: `a list
```

To insert at the end we need to define a function

```
let rec insert_tail list elem =
  match list with                               // Returns
  | []            -> [elem]                     // `a list
  | head :: tail  -> head :: insert tail elem   // `a list
```

4.1.2.1. Syntactic sugar

```
[e1; e2] // e1 :: e2 :: []

int_list = [1; 2; 3] // 1 :: 2 :: 3 :: []

insert_head = [0; int_list] // ERROR
// 0    :: [1; 2; 3]  ::  []
// int  :: int list   ::  ??

insert_head = [[0], int_list] // OK
// [0]      :: [1; 2; 3]  :: []
// int list :: int list   :: int list
```

That `insert_head` won't work becaue it is appending 0 to a list of integer and then to the empty list. That two elements have different types, because 0 is an `int` and the list is an `int list`, making the result an eterogeneous list.

4.1.3. Map

`Map` is a function that applies a function `f` to all elements of a list and returns the list result of the application.

```
// map : `a -> `b -> `a list -> `b list
let rec map f l =
  match l with
  | []      -> []
  | h :: t  -> f h :: map f t
```

4.1.4. Filter

`Filter` is a function that filter all elements of a list over a condition `p` and returns a new list with only the valid elements.

```
// filter : (`a -> bool) -> `a list -> `a list
let rec filter p l =
  match l with
  | []      -> []
  | h :: t  -> if p h then p :: filter p t else filter p t
```

We have that `filter p t` duplicated, so we can define a variable to reuse that value.

```
  | h :: t  ->  let filter_tail = filter p t
                if p h then p :: filter_tail else filter_tail
```

This is not only a syntactic difference: because of F# is a *strict language* everything is evaluated and executed one time.

4.1.5. Iter

`Iter` is a function that applies a function `f` to a list all elements without returning the result of the applications.

```
// iter : (`a -> unit) -> `a list -> unit
let rec iter f l =
  match l with
  | []     -> ()              // () unit represent nothing
  | h :: t -> f h; iter f t   // ;  binary operator : (unit * `a) -> `a
```

Take the application of `iter`

```
// Full
iter ( fun x -> printf("%d") x ) [ 1 .. 10 ]

// eta-reduced
iter (printf("%d")) [1 .. 10]
```

4.1.6. Sum elements in a list

Let's define a first monomorphic version over integers

```
// sum_mono : int list -> int
let rec sum_mono list =
  match list with
  | []     -> 0
  | h :: t -> h + sum_mono t
```

Now the polymorphic version, using the infix operator + in order to make it more easy and readable. We have to pass a function that manage the addition between our parametric type and the representation of the zero in that operation.

```
// sum : (`a -> `b -> `b) -> `b -> `a list -> `b
let rec sum (+) zero list =
  match list with
  | []     -> zero
  | h :: t -> h + sum (+) zero t
```

4.1.7. Fold

The function `fold_back` recur until the last element and them accumulate starting from the end.

```
// fold_back : (`a -> `b -> `b) -> `b -> `a list -> `b
let rec fold_back f acc list =
  match list with
  | []     -> acc
  | h :: t -> f h (fold_back f acc t)
```

The function that works in the other direction is `fold`

```
// fold : (`a -> `b -> `b) -> `b -> `a list -> `b
let rec fold f acc list =
  match list with
  | []     -> acc
  | h :: t -> fold f (f h acc) t
```

4.1.7.1. Using folding with other functions

NOTE: `@` is not a constructor, it is just a function in infix form that take two lists and combine them together.

```
let filter_by_fold p l =
  fold (fun x acc -> if p x then acc @ [] else acc) [] l
```

```
let filter_by_fold_back p l =
  fold (fun x acc -> if p x then x :: acc else acc) [] l
```

```
let map_by_fold f l =
  fold (fun x acc -> acc @ [ f x ]) [] l
```

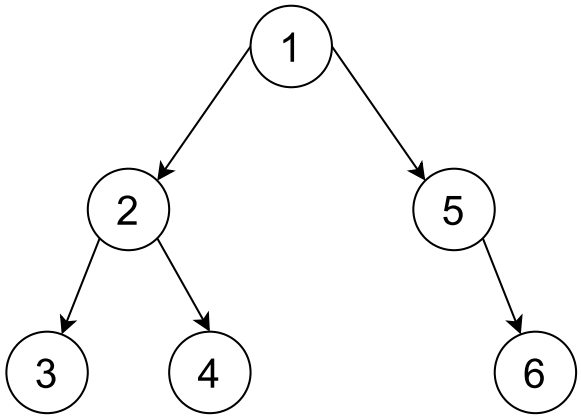
5. Trees

We start by defining a binary tree made of `Node` tuple of (data, left tree, right tree) and `Leaf`.

```
type `a bintree = Leaf | Node of `a * `a bintree * `a bintree
```

With this implementation create a simple tree is long:

```
let tree = bintree
(
  1,
  bintree(
    2,
    bintree(3, Leaf, Leaf),
    bintree(4, Leaf, Leaf)
  ),
  bintree(
    5,
    Leaf,
    bintree(6, Leaf, Leaf)
  )
)
```



5.1. Printing a tree

Just an example of a printing function that prints the tree going depth first on the left.

NOTE: in F# there is a special parameter specifier that is polymorphic, which is `"%0"`

```
let rec print_bintree tree =
  match tree with
  | Leaf ->
  | Node(data, left, right) ->
      printf "%0" data;
      print_bintree left;
      print_bintree right
```