Functional languages

Contents

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.

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
      |  ! 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.

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:

$$\overbrace{\overbrace{\underbrace{1 + 2}_{3} * 4}^{\text{int}}}^{\text{int}}$$
$$12$$

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.

Declaration, assegnation, binding

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

```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`

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*`.

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*.

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.

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
```

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).