Reflections on Worksheet 2

Lecture 3

F# Language Features: Coverage so far, or in last worksheet

Types

- `int` , `float` , `string` , `char`
- `bool`
  - `true` , `false`
- `unit`
  - `()`
- `T1 * T2` *(tuple or product type)*
  - `t1 , t2`
- `T1 -> T2` *(function type)*
- *polymorphic type variable*
  - `'a 'T1`
- *range (int or float)*
  - `3..10` , `0.5..2.5`
  - `1..2..9` , `1..-3..-5`
- *discriminated union* (sum type)
  - `| Age of int*int | Retired`
- **Option**
  - `Some 22` , `None`
- `{Person:string ; Age:int}` *(record type)*
  - `{Person = "Me" ; Age = 10}`
  - `{rec1 with Age = 12}`

Collections

- `List`
  - `[a ; b]` , `[]` , `::` , `[1..10]`
  - `lst.[10]` , `lst.[1..10]`
- `Array`
  - `[|a ; b|]` `x.[n]` `x.[a..b]`
- `Map`
  - `{"cat",1; "dog",10}`
  - `m.[k]`
- **Set**
- **Seq**

Language Constructs (Basic)

- *type definition*
  - `type ...`
  - `type ... and ...`
- let definition
  - `let`
  - `let rec ...`
  - `let rec ... and ...`
  - `let mutable`
- `fun ->`
- `if then else`
  - `&&` , `||` , `not`
  - `=` , `<>` , `>` , `<` , `>=` , `<=`
- `e1 ; e2`
  - `ignore`
- `x <- 1` *(assignment)*

Language Constructs

- *pipeline*: `|>`
  - Composition: `>>`
  - Backward pipeline: `<|`
- `match ... with | p1 -> e1`
  - **`when`** (guard expression)
  - List, tuple patterns
  - Active pattern
  - Partial active pattern
- *operators*
  - functionise: `(op)`
  - Custom definition
  - **`inline`**
- *exceptions*
  - **`failwithf`**
  - **`try with`**
  - **`try with finally`**
  - **`raise`**
  - **`reraise`**
- *computation expression*
- *seq comprehension*
- **`module`**
- **`interface`**
- **`class`**

Thinking functionally...

- Lists!
  - If not Lists then Maps!
  - Also Arrays, Sets...
  - Why are lists often used for sets of items?
- Use built-in collections functions to eliminate user recursion
  - Better practice
  - Faster
  - More compact
- Write lots of functions
  - Small functions are easy to understand and test
- Use pipelines
- Avoid use of mutable values
  - Use arrays without mutating elements where possible
  - Exceptions prove the rule...

Fold versus (tail) recursion

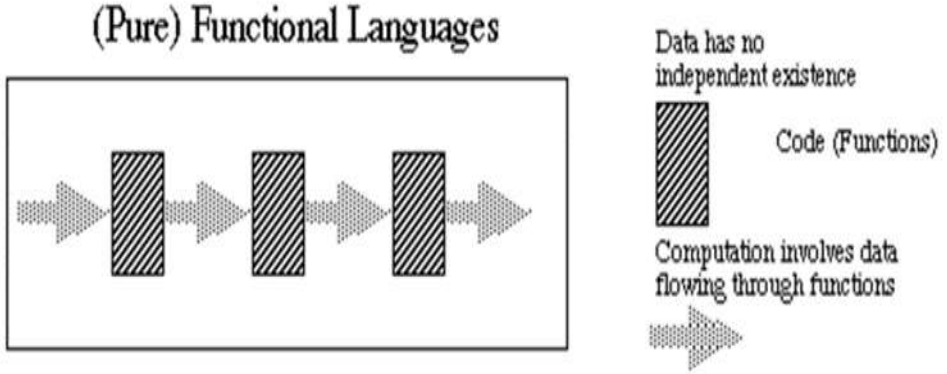
- List.fold is a helper function that implements one common type of recursion on lists
- List.fold will always be tail recursive, and implemented efficiently
- Simple uses can have folder function anonymous
- If List.fold is neater than tail recursion because the outer function is not needed
- Is it more readable?

Use pipelines

- What is a pipeline?
- Expression:
  - `let x = f( g( h(data)))`
- Think of this as an initial value (data) **transformed** by functions h, g, f in sequence.
- Pipelines are written in F# with syntactic sugar as:

```
let x = data |> h |> g |> f
```

- Here `a |> b` ≡ `(b a)`
  - **NB** `|>` associates to left



- Each function transforms its input data to make output
- Computation is represented as multiple functional transformations
- Note that nothing is over-written, transformations are best described as mathematical functions
- Use lots of simple steps
  - => use lots of functions

Worksheet 3

- Types as sets
  - **Algebraic data types: Sum** and **product** data type constructors
  - **Record** versus **Tuple** types for product
  - `list` as a recursive type definition
  - Defining arbitrary recursive types
- Mapping problem domains into data types
  - Key design issue
  - Choose types that promote program correctness
- Assertion-based Testing
  - Framework to write tests
  - Automatically generate test data
- .NET and OOP (object-oriented programming)
  - Need to know how to interface with .NET ecosystem libraries, which use OOP
  - Can write fully compatible .NET objects in F#

The Billion-dollar mistake

- `Null` is used in many HLLs (C++, Java, C#) as a reference value pointing to... nothing
- It is often (conveniently) used as a "no result" valid alternative return
- Unchecked Null returns cause run-time errors - difficult to debug
- Billion-dollar mistake

Tony Hoare (Qcom 2009):

I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

How to solve the 'Null' problem?

```
type Option<'a> = None | Some of 'a

let f (x: Option<int>) =
    match x with
    | None -> printfn "No value"
    | Some x -> printfn "value is %d" x
```

- Errors happen because the type system does not exactly model the valid data.
- `Option<int>` encodes in type system that **either** None **or** n∈int is an expected data value
  - Compiler can ensure that this is not forgotten by programmer
  - Must match an option value to unpack its underlying type.
- Generalise:
  - Type system must model problem domain data
  - Compiler can help ensure data processing is correct
- Disjoint Union (D.U.) type

Testing

- Testing is a hot topic in programming
- **Test Driven Development** - write tests **before** you write code
  - Advantage - conceptualising tests helps you to work out code properties
  - Advantage - tests document code
  - Advantage - at least you have some tests!
  - Disadvantage: "Enterprise Developer From Hell"
    - Passing tests is not success
    - Ad-hoc tests are not a code specification
    - No set of tests can safely ensure that all code works
    - https://fsharpforfunandprofit.com/posts/property-based-testing/
- Functional design + FsCheck
  - Randomised assertion-based test framework
  - Automatic generation of exhaustive tests? (sometimes, when data permits)
  - Works well in functional languages because **side effects** don't exist

```
int -> unit
let testExhaustive p =
    [2..(p |> float |> sqrt |> System.Math.Ceiling |> int)]
    |> List.iter (fun n -> assert (p = n || p % n <> 0))

int -> unit
let testRandom p =
    let numTests = 100
    let testBound =
        p |> float |> sqrt |> System.Math.Ceiling |> int
    let gen = System.Random()
    [1..numTests]
    |> List.iter (
            fun i -> gen.Next( 2,testBound)
            >> fun n -> assert (p = n || p % n <> 0)
    )
```