Reflections on Workshop 1

Lecture 2

F# Language & Syntax Reference

- Bin the bracket
  - Blocks and sub-blocks indicated by line indentation
    - No block brackets { }
    - Optional statement/expression end ";" end of line indicates end of statement if that makes sense
    - Start of inner block marked by indentation
  - Good tutorial on F# block syntax: http://bit.ly/1rTcXOw
- Finding documentation
  - Google topic e.g. "F# records" to get good references
    - MSDN - best in-depth resource (usually 1st hit in google)
    - fsharpforfunandprofit - good intro to many topics
  - Check library list for many textbooks, useful for in-depth examples
- For language nerds: the complete F# 4.0 language reference
  - http://bit.ly/22fUacC

F# Language Features: Coverage so far

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`**

Programs in F#

- Programs are sequences of expressions and let-bindings
  - The last item in the sequence must be an expression and is the value of the program
- Let-bindings introduce sub-blocks from which inner bindings cannot escape
- Let-bindings are identical to constant declarations in an imperative language

```
let sq x  = x*x
sq 3 + sq 4
```

x is bound to x*x in sq 3 + sq 4

```
let sq x  = x*x
printfn "x=%d" x    1.
sq 3 + sq 4         2.
```

- Expressions are evaluated in sequence
- Last expression is value
- Earlier expressions only relevant for side effects

```
let pythag a b =
    let sq x = x*x
    sq a + sq b
pythag 3 4
```

- Sub-blocks delimit scope of inner lets
- Here sq is local to the code in the sub-block

F# Definition order

- F# enforces "no forward references"
  - Enables precise type inference as you type
  - Factor code as you write it using **parameters** instead of forward references (see examples).
    - The use of a parameter to feed in a dependence is called **dependency injection** in OOP.
    - It prevents (less maintainable) cycles of references by allowing the forward part of the cycle to be replace by a parameter. **More later**.
- Love it or hate it
  - As above it forces better style
  - It can be annoying when functions inside one file must be ordered to prevent backward references

```fsharp
let f1 x =
    f2 x + f2 x // forward reference

let f2 x = x * x

let result = f1 2
```

```fsharp
let f1a f x = // no forward reference.
    f x + f x

let f2a x = x * x

let resulta = f1a f2a 2 // f2a is no longer forward reference
```

Types in F#

- F# makes no distinction between functions and non-functions. Expressions can be either.
- Every expression (and therefore every value) has a static **type**.
  - F# uses the *Hindley-Milner* type system which can infer types of all expressions
  - Type checking catches many **semantic** programming errors at compile-time
  - Good tools check types as you edit. Types *and parameters* of functions can be displayed by hovering mouse:
    - List.collect: mapping: (T -> U list) -> list: T list -> U list
- While editing auto-completion gives possible functions
  - List. (displays all List functions with info)

Example	Type
int, float	scalar types
T1 -> T2	Type of function with parameter T1 returning value of type T2
T1 * T2	Tuple with elements type T1 and T2
T1 list	List with elements of type T1
List.map	(T1->'b) -> 'a list -> 'b list
List.collect	('a -> 'b list) -> 'a list -> 'b list

Use of let definitions

- Use many local value definitions to make complex calculations simpler.
- Give local definitions descriptive names.
- Use pipelines instead of multiple `let` definitions where this is clearer and the intermediate value is immediately used.

<u>Unnecessary local names</u>

```
let deletedPairs = deleteBadpairs pairlist
let triples = List.map makeTriple deletedPairs
printTriple triples
```

<u>Idiomatic F# uses pipeline</u>

```
pairlist |> deleteBadPairs |> List.map makeTriple |> printTriple
```

or

```
pairlist
|> deleteBadPairs
|> List.map makeTriple
|> printTriple
```

How is **`let`** different from assignment (e.g. in C++)?

- Here the line 5 let-binding in lines creates a new copy of a and over-rules the line 2 let-binding for a in lines 5 & 6.
  - This is allowed
  - It is **bad practice**
- **`f1()`** references to the first a binding and therefore returns 1!
- a, `f1`, `f2` are all **immutable**

```
1 let testLet () =
2     let a = 1
3     let f1() = a
4     let a = 2
5     let f2() = a
6     printfn "a=%d, f1()=%d, f2=%d" a (f1()) (f2())
```

Speed

Mostly speed does not matter
Programmers should design correct code - not do low-level speed optimisation

There are exceptions!

- Work out whether efficiency is required
  - Usually with modern CPUs it is not!
- Functional solutions are inefficient when the same computation gets repeated
- If efficiency is needed, for the W1 deliverable each non-zero term of the sine series can be generated efficiently from the *previous term* in a loop:
  - $T_n(x) = -\frac{x^2 T_{n-1}(x)}{2n(2n+1)}$
  - $T_1(x) = x$
  - $\sin(x) \cong \sum_{i=1}^{i=N} T_n(x)$
- You can implement this using recursion, or List.fold
  - Some types of recursion will automatically be turned into equivalent loops!
  - See Worksheet 2

Worksheet 2

- Concepts
  - High order functions and currying
  - Type polymorphism
  - Recursion and Tail Recursion
- F# Practice
  - Collections: List, Array, (Set), Map

Currying

- In C function has fixed number of parameters
- In F#, technically, every function has just one parameter!
- Two different F# ways to get multiple parameters
  - Tuple
  - Currying
- Big advantage of currying
  - Can use partial application
  - e.g. with List.map
- Use curried functions in idiomatic F#

Tuples

```
let addT (x,y) = x+y
```

// (x,y) is a 2-tuple

// addT has a single parameter of type tuple

```
addT: int*int -> int
addT(1,2) // usage
```

Currying

let addC x y = x+y // curried function

Same as:

```
let addC x = (fun y -> x+y)
let addC = (fun x -> (fun y -> x+y))

addC 1 2 // usage parses as: (addC 1) 2
```

When first parameter is applied it returns a function that accepts the second parameter to return the result. As a function type:

```
addC: int -> (int -> int) // brackets not needed
                          // -> is right associative

let add5 = addC 5
```

// add5 is now a function which adds 5

High order functions and currying

```
List.filter pred lst :
  ('T -> Bool) -> 'T list -> 'T list
```

`pred: ('T -> bool)` *a specific predicate*

a specific List filter:

`(List.filter pred): 'T list -> 'T list`

- Use **currying** to generate *specific* filter functions by **partial application** of `List.filter`

```
let retainPositive =
   List.filter (fun x -> x >= 0)
```

- Functional combinators operate on <u>functions</u> as data
  - Complex impedance is represented as a function:

    `freq: float -> Complex`

  - Series, parallel connection implemented as **high order functions**

```
open System.Numerics
let pi = System.Math.PI
let R x = fun f -> Complex(x, 0.0)
let C x = fun f -> Complex(0.0, -1.0/(2.0*pi*x*f))
let L x = fun f -> Complex(0.0, 2.0*pi*x*f)

let seriesConnect c1 c2 f = (c1 f) + (c2 f)

let parallelConnect c1 c2 f = // note series, parallel are
  let x1 = c1 f               // curried functions
  let x2 = c2 f
  x1*x2 / (x1+x2)

let ( ++ ) = seriesConnect
let ( ||| ) = parallelConnect

let network = (R 0.1 ++ L 2.0) ||| C 3.0
```

let (++) = f
// defines ++ as infix
// operator equivalent of f
a ++ b ≡ f a b

Function application: the hidden operator

- Function application: •
  - f(x) = f • x
- Implied between any two adjacent items
- Left associative
  - Why is this necessary?
- Binds tighter than other operators

```
f a b c
```
F# source

((f • a) • b) • c

Parsed as

```
add 1 3 - 4
```
F# source

((add • 1) • 3) - 4

Parsed as

Unusual syntax found in ML-family functional languages
Makes complex expressions with curried functions easy

Type Polymorphism: 'a or 'T

- List.map: what is its type?
  - Problem is that the type of `List.map` depends on the context in which it is used.
  - Any fixed type is too specific
  - `let res = List.map f lst` we want the type of `f` to match the type of `lst` and `res`.
- Solution: ('a -> 'b) -> 'a list -> 'b list
  - 'a and 'b are called *polymorphic* or *generic* type variables
  - 'a and 'b can be changed as needed to fit a function call:

    ```
    [1 ; 2 ; 3 ] |> List.map (fun n -> 0.5 * (float n))
    ```

    In this example: 'a = int 'b = float
  - 'a and 'b are implicitly *universally quantified*:

    ∀'a.∀'b. ('a -> 'b) -> 'a list -> 'b list
  - This says for any List.map function call 'a, 'b can be freely chosen but the two 'a parameters and the two 'b parameters must have definite values for each function call.

Type Polymorphism (2)

- Type polymorphism is a very hot topic we will return to later
- Polymorphic types make type inference <u>more difficult</u>
  - To infer a function type each polymorphic type variable must be instantiated in the most general way to fit.
  - Hindley-Milner type systems (e.g. F#) have a fast computable algorithm to do this.
  - More complex type systems may not have decidable type inference or checking
- In Java and C# polymorphic type variables are called "generics". Polymorphism is more complex because of the *OOP type hierarchy*.

Tail recursion

- A function call is tail recursive if its return value is also the return value of the calling function
  - Nothing needs be done after the call returns
  - Allows compiler optimisation replacing subroutine branch by a jump with no stack use
  - The tail recursion turns into a loop
- Tail Recursive call is always outermost: except for conditional constructs that get executed first
- Tail recursion is more efficient - uses loops instead of function calls

non-tail recursive call

tail recursive call

Recursive function

```
let rec tTail x y z =
    if z = 1
    then tTail x y (z-1)
    else if z = 2
         then if y = 0
              then tTail x (tTail x x y) (z+2)
              else 5
         else if z = 3
              then tTail x (y-1) z + tTail x (y-1) z
              else tTail (x-1) y z + 5
```

loop - reuse locals -no stack needed!

When is a loop not just a loop....

When its a tail recursive function call!