CITS 3242
Programming Paradigms

Lecture 2:
Introduction to Functional Programming

This lecture introduces the main features of functional programming, contrasts it with imperative programming and shows some simple functional programs in F#. It also gives some background on the particular features of F# and other functional languages, as well as common application areas where they are used.

Reading: Expert F# pages 1-12

What is functional programming?

- At a high level: functional programming focuses on building functions.
- The programmer declares what the program does by defining a function that maps inputs to outputs.
- Complex functions are built by composing simpler functions.

```
let square x = x*x
let sumSqr (x,y) = square x + square y
```

- Generally this means functions in the mathematical sense:
  - In particular, variables are not modified by the code.
  - Instead variables are just names for values.

Imperative programming

- Functional programming can be contrasted with the more traditional *imperative programming*.
  - C, C++, Java, Basic, Fortran, etc are all imperative.
- An imperative programmer declares a set of variables and data structures that represent his/her view of the memory of the computer.
  - Weak abstraction: the ability to uses names instead of addresses.
  - But the programming still largely requires thinking about how memory needs to be modified.
- Procedures/subroutines and objects extend this abstraction, but they still have the same focus on modifying memory via variables.

Imperative programming – Historical Perspective

- Fortran was the first programming language
    - First version in 1954, by John Backus.
- It was designed to make it easy for mathematical *for*mulas to be *tran*slated into programs. E.g. :

  j = i + i ← Looks just like a mathematical formula/equation
- But, we usually want to evaluate a formula for many different input values. How will we write this in a program?
- Fortran allows variables to be assigned many times – the current value of the variable is the one from the most recent assignment.
- Early Fortran also included an "if" and "goto". E.g., to sum the numbers 1,2,...,10

```
      k = 0
      i = 1
(3)   j = i + i        ← Looks just like the mathematical formula
      k = k + j        ← Not a sensible mathematical formula
      i = i + 1        ← Not a sensible mathematical formula
      if i <= 10  goto (3)
```

- [This corresponds to a for loop in C, Java, etc.]

Imperative programming (cont.)

- An imperative program essentially consists of a sequence of assignments to variables.
  - The sequence may contain loops, jumps, branches, etc
  - These are abstracted as *for*-loops, *while*-loops, *case*-statements, etc.
- More precisely, the program describes a set of possible sequences
- Program execution consists of performing these assignments in the sequence described by the program
- The exact sequence actually performed generally depends on the input (and so does the output).

Issues with imperative programming

- Imprecise semantics
  - Different implementations often behave differently
  - Portability problem
- Constrained order of execution
  - side-effects
  - different execution orders give different results
  - difficult to introduce parallelism
- Weak abstraction mechanisms
  - the programmer often has to consider the computer and its store
- Difficulty of storage management
  - The updating of variables and the manual reuse of store leads to errors due to overwritten values, dangling pointers, aliasing, etc.
  - Java, C#, etc. partly avoid this by garbage collection (as in FP).

Functional Programming: Point of Departure

- Functional programming departs from the imperative model by retaining the mathematical form of variables.
- This means that modifying variables is not allowed.
- Instead to evaluate a formula many times, it is placed in a function, and the function is called many times.
- This is exactly what is done in mathematics.
- Instead of loops and gotos, recursive functions are emphasized, as well as applying functions to each element in a list or collection.
- There is also an emphasis on writing simple but general functions.
  E.g., to sum the squares of the numbers from 1 to 10:

```
let k = sum [ for x in 1..10 -> x*x ]
```

Abstraction in Functional Languages

- Functional languages are deliberately less directly based on the way the CPU and memory work.
- The programmer declares a set of data types that define the data on which the program will operate
  - strong abstraction: values and store are distinct
  - *how* values are represented is left up to the implementation
- The program consists of function definitions over values of these data types
- functions may be built from any well-defined operations on the data
- Program execution consists of applying one of the defined functions to some input values

Advantages

- Precise formal semantics
- Flexible order of execution
  - functions operate "in isolation"
  - natural parallel interpretation
  - particularly true for "pure" functional languages
- Rich abstraction mechanisms
  - values are distinct from the store
  - abstract over functions, infinite data structures, etc
- Automatic storage management
  - let the system do the work!

Disadvantages

- The programmer has less control over exactly what happens with the CPU and memory.
  - More efficient programs are often possible in a lower level language like C (with some work).
- Sometimes a bad order of execution may lead to excessive memory use.
  - *strict* functional languages like F# largely avoid this.
- Sometimes the natural way of expressing an algorithm is via a sequence of assignments.
  - Modern functional languages generally support imperative programming also in some way.
  - F# does this by being impure
  - Pure FLs like Haskell use monads. (We'll see later.)

What is FP used for?

- **Lisp** and **Scheme** are used widely in the field of artificial intelligence, as extension languages (e.g. for AutoCAD) and for general programming
- **ML** (including F#) is used for building robust software, language related tools, formal verification tools, scientific programming, financial programming and general programming
- **Haskell** is used widely for fast prototyping of programs, to build tools for hardware design and verification, to prototype implementations of new languages and for general programming
- **Erlang** is used at Ericsson and related companies for many applications in the field of telecommunications
- Many special purpose languages such as **SQL** (for database queries) and **XSLT** (for XML transformations) are functional languages
- Functional languages have survived the test of time, and continue to spread and influence other languages

F# - Background

- F# is a recent functional language that also supports .NET objects and concurrency.
  - It was originally designed at Microsoft Research in Cambridge.
  - The core language was originally the same as OCaml, the french dialect of ML.
  - Over time many things have been added to F#.
  - F# is in the process of being made ready to release as part of Visual Studio 2010.
- F# has also been significantly influenced by Haskell (including monads) and C#.

F# features (and all dialects of ML)

- It is *strict* or *call-by-value*
  - Arguments to functions are evaluated before starting to evaluate the body of the function.
- It is *Impure*: functions may have effects
  - *Effects* are actions such as modifying a value in memory or printing to the screen.
  - Generally effects are only used where necessary.
- It has a strong static type system.
  - Types are checked (and inferred) at compile time.

F# features (and mostly all dialects of ML)

- It is *polymorphic*.
  - Functions can be applied to values of more than one type.
- It is *higher order*.
  - Functions are first-class citizens.
  - I.e., they can be used just like other data types.
- It has *automatic storage management*
  - all issues of data representation and store re-use are handled by the implementation
- It has algebraic datatypes, exceptions.
- It has workflows/monads (following Haskell) and (.NET) objects (following C#), unlike other ML dialects.

F# examples: "let"

```
// turn on the lightweight syntax and open a namespace
#light
open System

// Use '///' to document a variable for Visual Studio "hovering"

/// A very simple constant integer
let i1 = 1
/// A second very simple constant integer
let i2 = 2

/// Add two integers
let i3 = i1 + i2

/// A function on integers
let f x = 2*x*x - 5*x + 3

/// The result of a simple computation
let result = f (i3 + 4)
```

F# examples: Recursion and Tuples

```
/// Compute the factorial of an integer recursively
let rec factorial =
  function
    | 0 -> 1
    | n -> n * factorial (n-1)

// A simple pair of two integers
let pointA = (32, 42)

// A simple tuple of an integer, a string and a double-precision
   floating point number
let dataB = (1, "fred", 3.1415)

/// A function that swaps the order of two values in a tuple
let swap (a, b) = (b, a)

/// The result of swapping pointA
let pointB = swap pointA
```

F# examples: Lists

```
/// The empty list
let listA = [ ]

/// A list with 3 integers
let listB = [ 1; 2; 3 ]

/// A list with 3 integers, note head::tail constructs a list
let listC = 1 :: [2; 3]

/// Compute the sum of a list of integers using a recursion
let rec SumList xs =
    match xs with
    | []      -> 0
    | y::ys -> y + SumList ys

/// The list of integers between 1 and 10 inclusive
let oneToTen = [1..10]

/// The squares of the first 10 integers
let squaresOfOneToTen = [ for x in 1..10 -> x*x ]
```