Functional Programming

With examples in F#

Pure Functional Programming

- Functional programming involves *evaluating expressions* rather than *executing commands*.
- Computation is largely performed by applying functions to values.
- The value of an expression depends only on the values of its sub-expressions (if any).
  - Evaluation does not produce side effects.
  - The value of an expression cannot change over time.
  - No notion of *state*.
  - Computation may generate new values, but not change existing ones.

Advantages

- Simplicity
  - No explicit manipulation of memory.
  - Values are independent of underlying machine with assignments and storage allocation.
  - Garbage collection.
- Power
  - Recursion
  - Functions as first class values
    - Can be value of expression, passed as argument, placed in data structures. Need not be named.

F#

- May be either interpreted or compiled.
- Interacting with the interpreter
  - Supply an expression to be evaluated
  - Bind a name to a value (could be a function)

```
> 3.14159
  val it : float = 3.14159


> let pi = 3.14159
  val pi : float = 3.14159


> pi
  val it : float = 3.14159
```

Arithmetic Expressions

```
> 5 * 7
val it : int = 35

> 5 * (6 + 4)
val it : int = 50

> 5.0 + 3.2
val it : float = 8.2

> pi * 4.7
val it : float = 14.765473
```

Arithmetic Expressions (2)

```
> 5.0 + 3

  5.0 + 3
  ------^

error: The type 'int' does not
  match the type 'float'


> 5.0 + float 3
  val it : float = 8.0
```

Anonymous Functions

- The fun keyword creates anonymous functions.
  - This can be useful when the function is used immediately or passed as a parameter to another function.

```
(fun param param … -> expression)
```

```
> let double = (fun x -> x + x)
  val double : int -> int

> double 18
  val it : int = 36

> (fun x -> x * 3) 18
  val it : int = 54
```

Functions: Multiple Parameters

- **Consider a function to add two numbers:**

```
> add 3 7
  val it : int = 10
```

- **In F# we would define this as:**

```
> let add = (fun x -> (fun y -> x + y))
  val add : int -> int -> int
```

- **Notice that *add* is really a function that takes one parameter, x, and returns a second function that takes another parameter, y, and adds it to x.**
- **For example, the value of *(add 3)* is a function that takes one parameter and adds it to 3.**

Functions: Multiple Parameters

- If we think of add as a function of two parameters, then (add 3) is a *partial application* of that function.
- Given the previous definition of add, we could define:

```
> let add3 = (add 3)
  val add : int -> int -> int
```

- Now we can use add3:

```
> add3 5
  val it : int = 8
```

- All of this makes sense because functions are true values.

Function Definition Shorthand

```
let name param param … = expression
```

```
> let add x y = x + y
  val add : int -> int -> int
```

```
> let add (x: float) y = x + y
  val add : float -> float -> float
```

```
> add 3.2 1.8
  val it : float = 5.0
```

If-Else Expressions

- Unlike if-else statements in imperative languages, if-else constructs are *expressions*, i.e. they have a value:

  if *test* then
      expression1
  else
      expression2

- If the test is true, the value is the value of expression1, otherwise the value of expression2.

If-Else Example

```
> let x = 5

> let y = 10

> let n =
    if (x > y) then
      x - y
    else
      x + y
  val n : int = 15
```

Naming vs. Assignment

In F# a let expression creates a new variable, it never changes the value of an existing variable. Similar to a declaration (as opposed to assignment) in C:

```
int f(int i)
{
  int x = 0;

  if (i > 10){
    x = 1;
  }
  else {
    x = 2;
  }

  return x;
}
```

```
int f(int i)
{
  int x = 0;

  if (i > 10){
    int x = 1;
  }
  else {
    int x = 2;
  }

  return x;
}
```

Recursion

- Most non-trivial functions are recursive.
  - Why is iteration not very useful in functional programming?
- To create a recursive function in F#, you must use the ***rec*** keyword.

```
> let rec factorial n =
      if n = 0 then 1
      else n * factorial (n-1)

   val factorial : int -> int


> factorial 10
  val it : int = 3628800
```

Tuples

- A tuple is defined as a comma separated collection of values. For example, (10, "hello") is a 2-tuple with the type (int * string).
  - Tuples are useful for creating data structures or defining functions that return more than one value.

```
> let swap (a, b) = (b, a)
  val swap : 'a * 'b -> 'b * 'a
```

  - 'a and 'b denote generic types to be inferred from the function call:

```
> swap (”day", ”night")
  val it : string * string = (”night", ”day")
```

  - Can also be used to bind multiple values in a let:

```
> let x, y = (5, 7)
  val y : int = 7
  val x : int = 5
```

Lists

- A list is a sequence of zero or more values of the same type.
- A list is written by enclosing its elements in [ ] separated by semicolons:

```
> let firstList = [ "a"; "b"; "c" ]
  val firstList : string list = ["a"; "b"; "c"]
```

- F#, like all functional languages, implements its lists as linked lists. Essentially, a list node in F# consists of a value (its **head**) and a **tail**, which is another list.

Operations on Lists

```
> firstList
  val it : string list = ["one"; "two"; "three"]

> List.length firstList
  val it : int = 3

> List.head firstList
  val it : string = "one"

> List.tail firstList
  val it : string list = ["two"; "three"]

> firstList @ ["four"; "five"]
  val it : string list = ["one"; "two"; "three";
  "four"; "five"]

> "zero"::firstList
  val it : string list = ["zero"; "one"; "two";
  "three"]
```

List Operations (2)

- **Most of the list operations would be easy to define ourselves.**
- **Example, length function:**

```
> let rec length list =
    if list = [] then 0
    else length (List.tail list) + 1

val length : 'a list -> int when 'a : equality


> length [2; 3; 5; 4; 1]

val it : int = 5
```

Option Types

- An ***option type*** can hold two possible values: *Some(x)* or *None*.
  - Option types are frequently used to represent optional values in calculations, or to indicate whether a particular computation has succeeded or failed.
  - Example: Avoid divide by zero exception

```
> let div x y =
    if y = 0.0 then None
    else Some (x/y)

> div 4.0 3.2
  val it : float option = Some 1.25

> div 4.2 0.0
  val it : float option = None
```

Pattern Matching

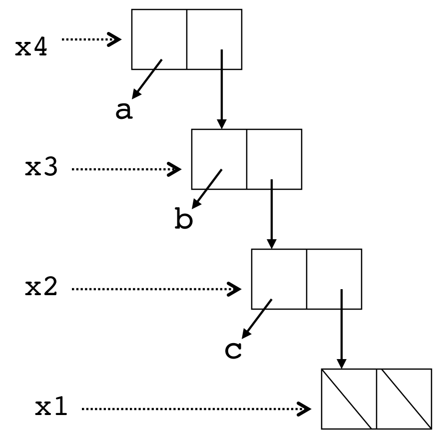
```
> let rec length list =
     match list with
        | [] -> 0
        | _ -> length (List.tail list) + 1


> let div x y =
    match y with
      | 0.0 -> None
      | _ -> Some (x/y)


> let rec last list =
     match list with
        | [] -> None
        | [x] -> Some x
        | _ -> last (List.tail list)
```

Efficiency: Cons vs. Append

```
> let x1 = []
> let x2 = "c"::x1
> let x3 = "b"::x2
> let x4 = "a"::x3
```

```
> x1
```
val x1 : 'a list = [ ]
```
> x2
```
val x2 : string list = ["c"]
```
> x3
```
val x3 : string list = ["b"; "c"]
```
> x4
```
val x4 : string list = ["a"; "b"; "c"]

List Nodes

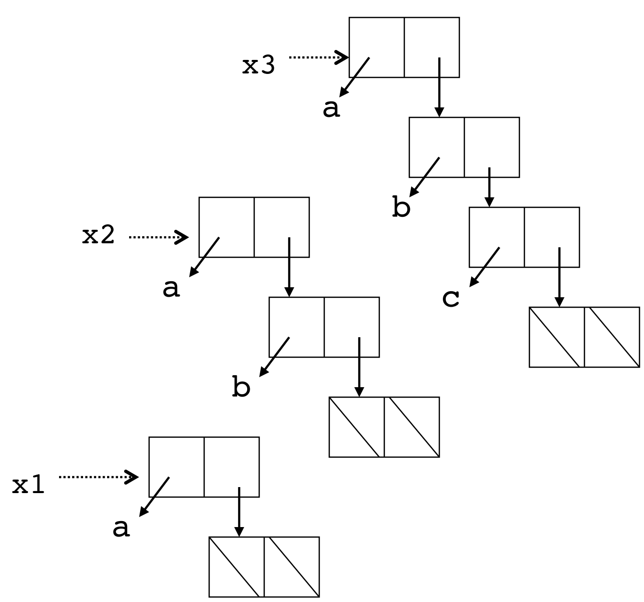


Efficiency: Cons vs. Append (2)

List Nodes

```
> let x1 = ["a"]
> let x2 = x1 @ ["b"]
> let x3 = x2 @ ["c"]
```

```
> x1
```
val x1 : string list = ["a"]
```
> x2
```
val x2 : string list = ["a"; "b"]
```
> x3
```
val x3 : string list = ["a"; "b"; "c"]



Efficiency: Tail Recursion

- When a recursive function returns the result of its recursive call, there is no need to maintain a stack of activation records.

```
> let rec last list =
     match list with
        | [] -> None
        | [x] -> Some x
        | _ -> last (List.tail list)
```

Efficiency: Tail Recursion (2)

```
> let rec length list =
     match list with
        | [] -> 0
        | _ -> length (List.tail list) + 1
```

```
> let rec length_help list acc =
     match list with
        | [] -> acc
        | _ -> length-help (List.tail list) (acc + 1)
```

```
> let length list = length_help list 0
```

Nested Functions

- F# allows programmers to nest functions inside other functions. This prevents the top level name space from becoming cluttered with helper functions:

```
> let length list =

      let rec length_help list acc =
         match list with
            | [] -> acc
            | _ -> length_help (List.tail list) (acc + 1)

      length_help list 0
```

Applications: Insertion Sort

```
let rec insertion_sort list =

  let rec insert n list =
    if List.length list = 0 then
      [n]
    else if n < (List.head list) then
      n::list
    else
      (List.head list)::(insert n (List.tail list))


  if List.length list = 0 then
    list
  else
    insert (List.head list) (insertion_sort (List.tail list))
```

Applications: Merge Sort

```
let rec merge_sort list =

  // merge two sorted lists (helper function)
  let rec merge x y =
    if x = [] then y
    else if y = [] then x
    else if (List.head x) < (List.head y) then
      (List.head x)::(merge (List.tail x) y)
    else (List.head y)::(merge x (List.tail y))
```

Applications: Merge Sort (2)

```
// split a list in two (helper function)
let rec split list (a, b) =
   if list = [] then (a, b)
   else split (List.tail list) (b, (List.head list)::a)

// sort
if (List.length list) < 2 then list
else
   let fstHalf, sndHalf = split list ([], [])
   merge (merge_sort fstHalf) (merge_sort sndHalf)
```

Higher Order Functions: Map

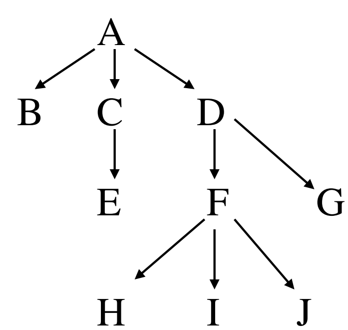
- Higher order functions are functions that take functions are arguments or return functions.
- The List.map function takes a function and a list as arguments, and returns a list of values obtained by applying the function to each element of the list:

```
> let double x = x + x

> List.map double [ 2; 4; 6]
  val it : int list = [ 4; 8; 12 ]
```

Example: Searching

- A directed acyclic graph in F#

```
let g = [
   ("a", ["b"; "c"; "d"]);
   ("b", []);
   ("c", ["e"]);
   ("d", ["f"; "g"]);
   ("e", []);
   ("f", ["h"; "i"; "j"]);
   ("g", []);
   ("h", [])
   ("i", []);
   ("j", []) ]
```



Searching (2)

```
let rec successors node graph =
    if graph = [ ] then [ ]
    else
      let head = List.head graph
      if fst head = node then snd head
      else successors node (List.tail graph)


let path_extensions path graph =
    List.map (fun node -> node::path) (successors (List.head path) graph)
```

Searching (3)

```
let rec expand graph goal paths =
    if paths = [ ] then [ ]
    else
      let first_path = List.head paths
      let remaining_paths = List.tail paths
      if List.head first_path = goal then
         first_path
      else
         expand graph goal ((path_extensions first_path graph) @ remaining_paths)


let search graph start goal =
    List.rev (expand graph goal [ [ start ] ])
```

```
search g "a" "i"

expand g "i" [["a"]]
```

	paths	first-path	remaining-paths	successors	path-extensions
1	((a))	(a)	( )	(b c d)	((b a) (c a) (d a))
2	((b a) (c a) (d a))	(b a)	((c a) (d a))	( )	( )
3	((c a) (d a))	(c a)	((d a))	(e)	((e c a))
4	((e c a) (d a))	(e c a)	((d a))	( )	( )
5	((d a))	(d a)	( )	(f g)	((f d a) (g d a))
6	((f d a) (g d a))	(f d a)	((g d a))	(h i j)	((h f d a) (i f d a) (j f d a))
7	((h f d a) (i f d a) (j f d a) (g d a))	((h f d a))	((i f d a) (j f d a) (g d a))	( )	( )
8	((i f d a) (j f d a) (g d a))	((i f d a))			