A Third Look At F#

Additional information

- Notes on tail recursion and continuations
- http://en.wikipedia.org/wiki/Continuation-passing_style

Outline

- Function values and anonymous functions
- Higher-order functions and currying
- Predefined higher-order functions
- Tail recursion
- Continuations

Defining Functions

- **`let`** notation for defining new named functions
- New names for old functions using **`let`** just as for other kinds of values:

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

let addxy = add;;
  val addxy : (int -> int -> int)

addxy 4 3;;
  val it : int = 7

let add4y = add 4;;
  val add4y : (int -> int)

add4y 3;;
  val it : int 7
```

Code as a parameter, evaluate first

```
let printResult x = printfn "%O" x
  val printResult : x:'a -> unit


printResult (4*3);;
 12

printResult (let x = 3 in x);;
 3

printResult (
 let car (L : 'a list) = L.Head
 in
  car [1;2;3]);;
 1
```

*E*valuation normally *eager*

- *Eager* evaluates before call, whether used or not.

```
let f x = printfn "%d" (1 / x);;
```

```
f 0;;
  System.DivideByZeroException: Attempted to
divide by zero.
```

Error because 3/0 always evaluated

lazy evaluation

- *Lazy* evaluates only *when* needed.

```
let f eval x =
 let y = lazy (1 / x)
 if eval then
  printfn "%d" y.Value;;
```

```
f true 0
   System.DivideByZeroException: Attempted to divide by
   zero.
```

```
f false 0
```

No error because 3/0 not evaluated prior to use:

`print(y)`

More later…

Function Values

- Functions in F# *do not have names*
- Function values are bound to variables, as with other kinds of values
- The **`let`** syntax does two separate things:
  - Creates a new function value
  - Binds that function value to a name
  - Syntactic sugar to define functions more easily

Function syntactic sugar

- Named function: value bound to *f*

```
let f (x:int) = x + 2;;
  val f : x:int -> int
f 1;;
  val it : int = 3
```

- Anonymous function: value bound to *f*

```
let f = (fun x -> x + 2);;
  val f : x:int -> int
f 1;;
  val it : int = 3
```

Anonymous Functions

An unbound function value

- Named function: value bound to *f*

```
let f (x:int) = x + 2;;
  val f : x:int -> int
f 1;;
  val it : int = 3
```

- Anonymous function: unbound function value

```
(fun x -> x + 2);;
  val it : x:int -> int = <fun:clo@185>>
(fun x -> x + 2) 1;;
  val it : int = 3
```

Examples - Anonymous Functions

Use – *non-recursive* function used only once.

```
let rec map f L =
 match L with
   | [] -> []
   | h::t -> f h::map f t;;
```

let

```
let inc x = x+1;;
map inc [1;2;3];;
  val it : int list = [2; 3; 4]
```

anonymous

```
map (fun x -> x + 1) [1;2;3];;
  val it : int list = [2; 3; 4]
```

Examples - Anonymous Functions

```
let rec reduce f a L =
 match L with
   | [] -> a
   | h::t -> f h (reduce f a t);;
```

let

```
let add a b = a + b;;

reduce add 0 [1;2;3;4] //  returns 10
```

anonymous

```
reduce (fun a b -> a+b) 0 [1;2;3;4];;
                                  returns 10
reduce (fun a b -> a*b) 1 [1;2;3;4];;
```

Examples - Anonymous Functions

let

```
let intBefore a b = a < b;;

quicksort [1;4;3;2;5] intBefore
  val it : int list = [1;2;3;4;5]
```

anonymous

```
quicksort [1;4;3;2;5]
            (fun a b -> a < b)
  val it : int list = [1;2;3;4;5]

quicksort [1;4;3;2;5]
            (fun a b -> a < b)
  val it : int list = [1;2;3;4;5]
```

Partially Applied Functions

- Functions can return functions as a result:
-

```
let add x y = x + y;;
add 3 2;;
  val it : int = 5

val z = add 3
  val it : (int -> int) = <fun:it@248-8>

z 2
  val it : int = 5
```

- **`add 3`** returns the partially applied function with **`x=3`**
- The function returned can then be called: **`z 2`**

Partially Applied Functions

- *Partially applied* functions fix one or more of multiple arguments

```
let g a b = a + b;;
  val g : a:int -> b:int -> int

let h = g 2;;
  val h : (int -> int)

h 4
  val it : int = 6
```

- *g 2* returns a *partial* function with unbound parameter
  val h = (fun b -> 2 + b)

Exercise 1.6 - Give results

```
let f1 x y = if x>y then x else y;;
```

a) **f1 3 8**

b)
```
let z = f1 3
z 8
```

c)
```
let m = f1
m 3 8
```

Functions returning functions

- Functions can return functions as a result:

```
let inc x = x + 1;;
let dubal x = x * 2;;

let pick a =
 match a with
   | 1 -> inc
   | 2 -> dubal;;
  val pick : a:int -> (int -> int)

let p = pick 1;;
```

- The function returned can then be called:

```
p 4;;
val it : int = 5
```

Functions returning Anonymous functions

- Functions can return functions as a result:

```
let pick a =
 match a with
   | 1 -> (fun x -> x + 1)
   | 2 -> (fun x -> x*2);;
    pick: (a: Int)Int => Int
val z = pick 2;;
   z: Int => Int = <function1>
```

- *pick* returns functions having same signature.
- Both functions have one int parameter and return int.
- The function returned can then be called:

```
pick 2 4;;
  val it : int = 8
z 4;;
 val it : int = 8
```

Exercise 1.7 - Give results

1.
```
let f1 x = if x>0 then (fun y -> y-1) else
                       (fun y -> y+1);;
```
   a) **f1 (3)**
   b) **f1 (3) (8)**
   c) **f1**

2.
```
let f2 x = (fun y -> x + y)
```
   a) **f2 (3)**
   b) **f2 (3) (4)**
   c) **f2**

Outline

- More pattern matching
- Function values and anonymous functions
- Higher-order functions and currying
- Predefined higher-order functions

Higher-order Functions

- Every function has an *order*:
  - A function with no functions as a *parameter*, and does not *return* a function value, has *order 1*
  - A function with a function as a parameter or returns a function value has *order n+1*, where *n* is the order of its highest-order parameter or returned value
- The **`quicksort`** we just saw is a second-order function because it has one *function* parameter

```
let eq x y = x=y;;                    order 1
  val eq : x:'a -> y:'a -> bool
let comp f g x = f (g x);;  order 3
  val comp : f:('a -> 'b) -> g:('c -> 'a) ->
  x:'c -> 'b
```

Exercise 2 What is the order and an example.

f: int * double -> bool function of: int & double tuple returns Boolean

```
let f (x:int*double) = fst x > snd x;;
```
order 1

h : y:double -> x:int -> bool function of: double returns function of int returns bool

```
let h (y : double) = (fun (x : int) ->
   y < double x);;
```
order 2

m : g:(int -> 'A) -> x:int -> 'Afunction of: function Int that returns A, & of Int, returns A.

```
let m (g:int -> 'A) (x : int) = g x;;
```
order 2

a) x:int -> int

b) x:(int * int -> int) -> bool

c) L:int list -> int list

d) x:(int -> bool) -> y:(bool -> double) -> string

e) x:('A -> 'B) -> y:('B -> 'C) -> 'D

Exercise 2 Continued

Give the anonymous functions equivalent to:

a) `let dubal x = 2 * x;;`

b) `let mult x y = x * y;;`

Duplicate using anonymous functions.

a) `map dubal [1;2;3;4];;`

b) `reduce mult 1 [1;2;3;4];;`

Currying

- *Currying* transforms function of multiple arguments into chain of functions each taking one argument
-
```
let g a b c = a::b::c::[];;
  val g : a:'a -> b:'a -> c:'a -> 'a list
```

- *g* returns *curried* function, a chain of three one-parameter functions

```
g;;
   val it : ('a -> 'a -> 'a -> 'a list)
g 2;;
   val it : (int -> int -> int list)

g 2 3;;
   val it : (int -> int list)

g 2 3 4;;
   val it : int list = [2; 3; 4]
```

Currying

- *Currying* transforms function of multiple arguments into chain of functions each taking one argument

```
let g a b c = a::b::c::[];;
  val g : a:'a -> b:'a -> c:'a -> 'a list
```

- *g* returns *curried* function, a chain of three one-parameter functions

```
let h = g;;
     val h : ('a -> 'a -> 'a -> 'a list)
h 2;;
     val it : (int -> int -> int list)

h 2 3;;
     val it : (int -> int list)

h 2 3 4;;
     val it : int list = [2; 3; 4]
```

Curried functions

```
let g1 a b c = a::b::[];;

let g2 a b c = a::b::c::[];;

let h1 a = fun b c -> g2 a b c;;

let h2 b = fun a c -> g2 a b c;;

let h3 = g1

let h4 = h3 2

let h5 = h4 3
```

Partially applied/Curried difference

- Partially applied functions fix some of the parameters in *any* position

```
let g1 a b c = a::b::c::[];;
  val g1 : a:'a -> b:'a -> c:'a -> 'a list

let h1 a c = g1 a 2 c;;
  val h1 : a:int -> c:int -> int list
```

- Curried functions fix the parameter of the *first* function

```
let g2 a b c = a::b::c::[];;
  val g2 : a:'a -> b:'a -> c:'a -> 'a list
let h2 = g2 2;;
  val h2 : (int -> int -> int list)
let h3 = g2 2 3;;
  val h3 : (int -> int list)
```

Advantages: Example

- Convert *binary* function to *unary* with fixed first parameter. Useful with *map, reduce,* ...

```
let rec map f L =
 match L with
  | [] -> []
  | h::t -> f h::map f t;;
```

```
map (fun x -> x+1) [1;2;3];; //return [2;3;4]
let sum a b = a+b;;
map (sum 1) [1;2;3];; // return [2;3;4]

map (fun x->[1;x]) [1;2;3]
            returns [[1;1]; [1;2]; [1;3]]
let list a b = [a;b];;
let h = list;;                          // curried
map (h 1) [1;2;3]
            returns [[1;1]; [1;2]; [1;3]]
```

Advantages: Example

- the comparison function is a *first*, curried parameter

```
quicksort (fun a b -> a<b) [1;4;3;2;5];;
  val it : int list = [1;2;3;4;5]

let sortBackward =
      quicksort (fun a b -> a>b);;

sortBackward [1;4;3;2;5];;
  val it : int list = [1;2;3;4;5]
```

Advantages Example

- Compute function at runtime
- Signatures of computed functions must be identical

```
let sortorder order x y =
 match order with
   | "ascending" -> x >= y
   | "descending" -> x <= y;;

sortorder "ascending" 3 4
  val it : bool = false

let orderup : int->int->bool = sortorder "ascending";;
  val orderup : (int -> int -> bool)

orderup (3) (4)
  val it : bool = false
```

Advantages: Example

- Convert *n-ary* function to *lower order* with fixed parameters with *map, reduce, ...*

```
let sortorder order x y =
 match order with
   | "ascending" -> x >= y
   | "descending" -> x <= y;;

let ge:int->int->bool = sortorder "ascending";;

let L = map (ge 3) [1;2;3;4];;
 val it : bool list = [true; true; true; false]
```

Multiple Curried Parameters

```
let f a b c = a+b*c;;
  val f : a:int -> b:int -> c:int -> int

let g a = fun b -> fun c -> a+b*c;;
  val g : a:int -> b:int -> c:int -> int

let h = f;;
  val h : (int -> int -> int -> int)

f 2 3 4;;
  val it : int = 14
h 2 3 4;;
  val it : int = 14
g 2 3 4;;
  val it : int = 14
```

Exercise 3

```
let f a = fun b -> a / b;;
let g a b = a / b;;
let h = g;;
```

What is the result of:

```
let rec map f L =
 match L with
   | [] -> []
   | h::t -> f h::map f t;;
```

1. `f;;`
2. `g;;`
3. `f 13;;`
4. `h 13;;`
5. `f 13 4;;`
6. `map (g 12) [2;3;4];;`
7. `let x = map (g 12);;`
8. `x [2;3;4];;`

Exercise 3 Continued

9. Write a *curried* function of two parameters that subtracts integer parameter 1 from integer parameter 2.
10. Use the previous function and *map* to subtract integer 10 from each integer of [1;2;3;4;5].
11. Write a *curried* function of two parameters that returns true when parameter 2 is greater than parameter 1.
12. Use the previous function and *filter* to return a list of integers greater than 3, similar to the following:
    let p x = x > 3;
    filter p [1;2;3;4;5] returns [4;5]

```
let f x y = x + y;;
let f1 = f 1
```

```
let rec filter f L =
  match L with
    | [] -> []
    | h::t when f h -> h::filter f t
    | h::t -> filter f t
```

Computing Anonymous Functions

- *map* has two parameters, a unary function *f* and list
- Returns partial function where *map* function has parameter *f* bound to *inc* and one unbound list parameter.
- *mapINC* is computed to be an unary function that increments every element of an integer list.

```
let rec map f L =
 match L with
   | [] -> []
   | h::t -> f h::map f t;;
```

```
let inc x = x+1;;

let mapINC = map inc;;          f bound to inc
     val mapINC : (int list -> int list)

mapINC [1;2;3];; // returns [2; 3; 4]
```

Computing Functions

- *bu* has two parameters, a binary function *f* and *f*'s first parameter
- Returns unary function where *f* and *first* parameter are bound and second parameter is unbound.
- *bu* stands for *binary* to *unary*, since bu takes a binary function and returns a unary function.

```
let sub x y = x - y;;

let bu f a b = f a b;;


bu sub 1 3;;
  val it : int = -2
let SUB1 = bu sub 1;;
  val SUB1 : (int -> int)
map SUB1 [1;2;3];;     // return [0;-1;-2]

map (bu sub 1) [1;2;3];;  // return [0;-1;—2]
```

Automating Computing Functions

- *rev* has one parameter, a binary function *f*
- Returns binary function where *f* is bound and the first and second parameters are reversed.
- *rev* stands for *reverse* the binary parameters.

```
let bu f a b = f a b;;

let rev f a b = f b a;;

sub 1 3;;                          Int = -2

rev sub 1 3;;                      Int = 2

bu (rev sub) 1 3;;                 Int = 2

map (bu (rev sub) 1) [0;1;2];;  [0;1;2]
```

Exercise 3 Continued

```
let bu f a b = f a b;;
let rev f a b = f b a;;
let cons a b = a::b;;
let rec map f L =
 match L with
   | [] -> []
   | h::t -> f h::map f t;;
let sub x y = x - y;;
```

```
15. bu cons 3;;
16. bu cons 3;;
17. map (bu cons 3) [[4;5];[6]];;
18. let a1 = bu sub 1;;
19. let m = map a1;;
20. m [1;2;3];;
21. map m [[1;2];[4;5];[6]];;
22. map (map (bu sub 1)) [[1;2];[4;5];[6]];;
23. map (map (bu rev sub 1)) [[1;2];[3;4]]
```

Computing Anonymous Functions

- Old *reduce* has parameters *f* and *a* fixed but repeatedly passed
- *Staged reduce* returns anonymous reduce function of one parameter
- Returns *red* function of one unbound parameter.

```
let rec reduce f a L =
  match L with
    | [] -> a
    | h::t -> f h (reduce f a t);;
```

```
let reduce f a L =
 let rec red L =
  match L with
    | [] -> a
    | h::t -> f h (red t)
 in
 red L;;

let r = reduce sub 0;;

r [1;2;3];;  //          returns 2
```

23. Give the results. Exercise 3

```
let rec reduce f a L =
  match L with
    | [] -> a
    | h::t -> f h (reduce f a t);;
```

```
let x = reduce sub 1;;
x [100; 25; 5];;
```

```
let reduce f a L =
 let rec red L =
  match L with
    | [] -> a
    | h::t -> f h (red t)
 in
 red L;;
```

```
let x = reduce sub 1;;
x [100; 25; 5];;
```

Exercise 3 Continued

24. Give a staged version for *filter*.

```
let reduce f a L =
 let rec red L =
  match L with
    | [] -> a
    | h::t -> f h (red t)
 in
 red L;;
```

```
let rec filter f L =
  match L with
    | [] -> []
    | h::t when f h -> h::filter f t
    | h::t -> filter f t
let lt3 = x < 3;;

filter lt3 [1;2;3;4];;  // returns [1;2]
```

Outline

- More pattern matching
- Function values and anonymous functions
- Higher-order functions and currying
- Predefined higher-order functions
- Tail recursion and Continuations

Predefined Higher-Order Functions

- Three important predefined higher-order functions:
  - **`List.map`**
  - **`List.fold`**
  - **`List.foldBack`**
- **`List.fold`** and **`List.foldBack`** are similar

- Defined for List package
- https://msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/collections.list-module-%5bfsharp%5d
- https://fsharpforfunandprofit.com/posts/recursive-types-and-folds-2b/

The **`map`** Function

Apply *unary* function to every element of a *List*, and collect a list of results

List.map{ unary function }

```
List.map (fun i -> -i) [1;2;3];;
     val it : int list = [-1; -2; -3]

List.map (fun x -> x+1) [1;2;3];;
     val it : int list = [2; 3; 4]

List.map (fun (a,b) -> a+b) [(1,2); (3,4)];;
     val it : int list = [3; 7]

List.map (fun i -> i*2) [1..5];;
     val it : int list = [2; 4; 6; 8; 10]
```

Exercise 4.0

Evaluate:

1. `List.map (fun x -> x*x) [1;2;3];;`

2. `List.map (fun x -> if x < 0 then -x else x) [-1;0;1];;`

3. `List.map (fun x -> ("A", x)) [1;2;3];;`

4. `let f a L = List.map (fun x -> (a, x)) L;;`

   `f "A" [1;2;3];;`

The **`foldBack`** Function

- Similar to **`reduce`**.
- Used to combine all the elements of a list
- Binary function *f*,
  a starting value *c*,
  List $x = (x_1, \ldots, x_n)$ and computes:

$$f\left(x_1, f\left(x_2, \cdots f\left(x_{n-1}, f\left(x_n, c\right)\right)\cdots\right)\right)$$

```
List.foldBack (fun a b -> a+b) [1;2;3] 0;;
```

evaluates as 1+(2+(3+0)) = 6

`foldBack` Examples

$$f\left(x_1, f\left(x_2, \cdots f\left(x_{n-1}, f\left(x_n, c\right)\right)\cdots\right)\right)$$

```
List.foldBack (fun x c -> x-c) [1;2;3;4] 0;;
  Int = -2      (1-(2-(3-(4-0))))

List.foldBack (fun x c -> if x<c then x else c)
     [4;2;8;5] 9999;;
  Int = 2

List.foldBack (fun x c -> x::c) [1;2;3] [10];;
     1::(2::(3::[10]))
     int list = [1; 2; 3; 10]
```

`fold` Function

- Used to combine all the elements of a list
- Same results as **`foldBack`** in some cases
- Note base case *c* and List elements *x* are switched.

```
List.fold (fun c x -> x+c) 0 [1;2;3];;
   Int = 6

List.fold (fun c x -> x*c) 1 [1;2;3;4];;
   Int = 24
```

The **`fold`** Function

- Takes a function *f*, a starting value *c*, list $x = (x_1, \ldots, x_n)$ and computes:

$$f\left(x_n, f\left(x_{n-1}, \cdots f\left(x_2, f\left(x_1, c\right)\right)\cdots\right)\right)$$

-

```
List.fold (fun c x -> x+c) 0 [1;2;3];;
```

  evaluates as 3+(2+(1+0)) = 6

- Remember, **`foldBack`** did 1+(2+(3+0))=6

The **`fold`** Function

- **`fold`** starts at the left, **`foldBack`** starts at the right
- Difference does not matter when the function is *associative* and *commutative*, like **+** and *****
- For other operations, such as – or /, it does matter
- Note base case *c* and List elements *x* are switched.

$$f\left(x_n, f\left(x_{n-1}, \cdots f\left(x_2, f\left(x_1, c\right)\right)\cdots\right)\right)$$

```
List.fold (fun c x -> x-c) 0 [1;2;3];;
     Int = 2                    1-(2-(3-0))
```

Compare **fold foldBack**

- **fold** $f(x_n, f(x_{n-1}, \cdots f(x_2, f(x_1, c))\cdots))$
- **foldBack** $f(x_1, f(x_2, \cdots f(x_{n-1}, f(x_n, c))\cdots))$

```
let cons a L = a::L;;
let L = [];;

List.foldBack (fun x c -> cons x c) [1;2;3] L;;
     val it : int list = [1; 2; 3]
     cons (1, cons(2, cons(3, Nil)))

List.fold (fun c x -> cons x c) L [1;2;3];;
     val it : int list = [3; 2; 1]
     cons (3, cons(2, cons(1, Nil)))
```

Exercise 4.1

1. Show evaluation:

- **fold** $f(x_n, f(x_{n-1}, \cdots f(x_2, f(x_1, c))\cdots))$
- **foldBack** $f(x_1, f(x_2, \cdots f(x_{n-1}, f(x_n, c))\cdots))$

```
List.foldBack (fun x c -> x/c) [128;16;4] 1;;

List.fold (fun c x -> x/c) 1 [128;16;4];;
```

2. Evaluate:

```
List.fold (fun c x -> x::c) [] [1;2;3;4];;

List.fold (fun c x -> List.map (fun a -> a*a) (x::c))
        [] [1;2;3;4];;

let f L = List.fold (fun c x -> x::c) [] L;;
    f [(1, "A"); (2, "B"); (3, "C"); (4, "D")];;
```

3. Evaluate. Exercise 4.1

```
let reverse (s:string) = new string(Array.rev
     (s.ToCharArray()));;
let toUpper (s: string) = s.ToUpper();;
let appendBar (s: string) = s + "bar";;

appendBar (toUpper (reverse "Foo"));;    //OOFbar
reverse (toUpper (appendBar "Foo"));;    //RABOOF

List.fold (fun c xf -> (xf c)) "foo"
     [reverse; toUpper; appendBar];;

List.foldBack (fun xf c -> (xf c))
     [reverse; toUpper; appendBar] "foo";;
```

Outline

- More pattern matching
- Function values and anonymous functions
- Higher-order functions and currying
- Predefined higher-order functions
- Tail recursion and Continuations

Tail recursion

A function call is tail recursive if there is nothing to do after the function returns except return its value.

Tail recursive functions translated by compiler to iteration.

Not tail recursive because multiply by *n* before return.

```
let rec factorial n =
  match n with
     | 1 -> 1
     | n -> factorial (n-1) * n;;
```

Tail recursive.

```
let rec tailFact a n =
  match n with
     | 1 -> a
     | n -> tailFact (a*n) (n-1);;
let factorial n = tailFact 1 n;;
```

Tail recursion

Non-tail recursion accumulates returned results after recursion.

```
let rec factorial n =
  match n with
    | 1 -> 1
    | n -> factorial (n-1) * n;;
```

Tail recursion accumulates results before recursion

```
let rec tailFact a n =
  match n with
    | 1 -> a
    | n -> tailFact (a*n) (n-1);;
```

`factorial 3`

n	return
3	6
2	2
1	1

`tailFact 1 3`

a	n	return
1	3	6
3	2	6
6	1	6

Equivalent tail recursion

```
let rec tailFact a n =
  match n with
    | 1 -> a
    | n -> tailFact (a*n) (n-1);;
let factorial n = tailFact 1 n;;
```

```
let factorial (n:int):int =
    let rec tailFact a n =
    match n with
      | 1 -> a
      | n -> tailFact (a*n) (n-1)
    in
      tailFact(1,n);;
```

Exercise 5 - Tail recursion

```
let rec tailFact a n =
  match n with
    | 1 -> a
    | n -> tailFact (a*n) (n-1);;
let factorial n = tailFact 1 n;;
```

Give tail recursive versions.

```
let rec sum L =
  match L with
    | [] -> 0
    | h::t -> h+(sum t);;
```

```
let rec rdc L =
  match L with
    | h::[] -> []
    | h::t -> h::rdc t;;
```

Exercise 5 Solutions - Tail recursion

```
let rec sum L =
  match L with
    | [] -> 0
    | h::t -> h+(sum t);;
```

```
let rec sumTail a l =
 match L with
   | [] -> a
   | h::t -> sumTail (a+h) t;;
```

```
let rec rdc L =
  match L with
    | h::[] -> []
    | h::t -> h::rdc t;;
```

```
let rec rdcTail a L =
  match L with
   | h::[] -> a
   | h::t -> rdcTail (h::a) t;;
```

```
rdc [1;2;3];;          returns [1;2]
rdcTail [] [1;2;3];;   returns [2;1]!!
```

Tail recursion problem

Not tail recursive but preserves order of :: application.

```
let rec id L =
  match L with
     | [] -> []
     | h::t -> h::id t;;
```

Tail recursive but :: not associative results in reverse order.

```
let rec idTail a L =
  match L with
     | [] -> a
     | h::t -> idTail (h::a) t;;
```

```
idTail [] [1;2;3;4];; = 4::3::2::1::Nil
```

Convert end-of-function recursion to iteration

```
int factorial(int n)
  if n = 1
     then factorial ← 1
     else factorial ← factorial(n -1) * n
```

```
int factorial(int n)
  int factorial ← 1;
  while  n ≠ 1
       factorial ← factorial * n
       n ← n -1
```

Convert Function End Recursive operation to Iterative Operation

1. change `if` to `while`
2. change `while` test to the `not` of `if` test (De Morgan's Laws)
   `if n = 1` to `while n ≠ 1`
3. change recursive call to assignment statement that modifies value of variable controlling `while` loop
   `factorial(n - 1)` to `n ← n-1`