Patterns in F#

Functions of a single parameter

Example

```
> let f1 n = n*n;;
val f1 : n:int -> int
> f1 3;;
val it : int = 9
```

- n matches and *binds* to any single int argument.
- `f1 3` binds `n` to 3.

A tuples parameter

Example

```
> let f2 (a, b) = a*b;;
val f2 : (a:int * b:int) -> int
> f2 (3,4);;
val it : int = 12
> f2 4 5;;
error FS0003: This value is not a function and
  cannot be applied
```

- `(a,b)` matches and binds `a` and `b` any 2-tuple int argument.
- `f2 (3, 4)` binds `a` to 3 and `b` to 4.

Pattern-Matching Literal Values

Example

```
> let printRole role =
    match role with
      | "admin" -> printfn "Administrator"
      | "user" -> printfn "User";;
val printRole : role:string -> unit

> printRole "user";;
User
```

- Note indentation of **match** and | lines

_ as a Pattern

Example

```
> let printRole role =
    match role with
      | "admin" -> printfn "Administrator"
      | "user" -> printfn "User"
      | _ -> printfn "Unknown User";;
val printRole : role:string -> unit

> printRole "hacker";;
Unknown User
```

- When no match in a match expression, throws exception.
- The underscore pattern matches any string argument.

Guard statements

Example

```
> let absVal x =
    match x with
      | y when y < 0 -> -y
      | y -> y;;
val printInputType : x: int -> int

> absVal -4;;
val it: int = 4
```

Constants as Patterns

Example

```
> let f x =
    match x with
      | 0 -> "x == 0";;
val f : x:int -> string
> f 0;;
val it : string = "x == 0"
```

- Note the function returns a string result
- The type of `f` is `x:int -> string`

Lists of Patterns as Patterns

Example

```
> let f L =
    match L with
      | [a; _] -> a;;
<warning FS0025>
val f : L:'a list -> 'a
> f [1; 2];;
val it : int = 1
> f [1];;
<MatchFailureException>
> f [1; 2; 3];;
<MatchFailureException>
```

:: of Patterns as a Pattern

Example

```
> let f L =
    match L with
      | h::t -> h;;
<warning FS0025>
val f : L:'a list -> 'a
> f [1;2];;
val it : int = 1
> f [];;
<error FS0030>
```

Tuples as a Pattern

Example

```
> let f T =
    match T with
      | (10, s) -> ("Ten", 10)
      | (a, b) -> (b, a);;
val f : int * string -> string * int
> f (10, "OK");;
val it : string * int = ("Ten", 10)
> f (1, "Hi");;
val it : string * int = ("Hi", 1)
```

F# Patterns So Far

- Single variable matches and binds to anything
- _ matches anything but binds to nothing
- Constant (of an equality type) matches only that constant
- Guards can be added with the `when` keyword
- Tuple of patterns matches any tuple of the right size, whose contents match the sub-patterns
- List of patterns matches any list of the right size, whose contents match the sub-patterns
- Cons (::) of patterns matches any non-empty list whose head and tail match the sub-patterns

Exercises

What are the results?

```
let f n = n*n;;
f 5;;
f (3,5);;
let f a b = a * b;;
f 6 4;;
let f L =
  match L with
    | (a,b,_) -> b;;
f (2, 5, 4);;
f (3, 1);;
let f L =
  match L with
    | x::xs -> xs;;
f [6;4;1];;
```

Exercises

What are the results?

```
let f L =
  match L with
    | x::y::z -> y;;
f [1;2;3];;
f [1];;
let f x y = x @ y;;
f [1;2] [true];;
f [1]@[2] [3];;
```

Patterns Everywhere

Example

```
> let (a,b) = (1,2.3);;
val b : float = 2.3
val a : int = 1

> let a::b = [1;2;3;4;5];;
val b : list = [2;3;4;5]
val a : int = 1
```

- Hallmarks of functional languages:
  - Patterns are not just for match
  - Here we see that you can use them in a **let** assignment
  - More ways to use patterns, later

Factorial with Patterns

Example (Without Patterns)

```
let rec fact n  =
  if n = 0 then 1
  else n * fact (n-1);;
```

Example (With Patterns)

```
let rec fact n =
  match n with
    | 0 -> 1
    | _ -> n * fact (n-1);;
```

Reverse with Patterns

Example (Without Patterns)

```
let rec reverse L =
  if L = [] then []
  else reverse L.Tail@[L.Head];;
```

Example (With Patterns)

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

Sum and Count a List

Example (Sum)

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

Example (Count true's)

```
let rec count L =
  match L with
    | [] -> 0
    | true::t -> 1 + count(t)
    | false::t -> count(t);;
```

Increment list values by 1

Example

```
let rec f L =
  if L = [] then []
  else (L.Head+1) :: f L.Tail;;
```

Example

```
let rec f L =
  match L with
    | [] -> []
    | h::t -> h+1 :: f(t);;
```

Minimum of list

Example

```
let rec smallest (L : int list) : int =
  if L.Tail = [] then L.Head
  else if L.Head < smallest L.Tail then L.Head
    else smallest L.Tail;;
```

Example

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

Guards

Example (Error: **a** appears twice)

```
let f x y =
  match (x,y) with
    | (a,a) -> "=" // pairs of equal elements
    | (a,b) -> "!=";; // pairs of unequal elements
```

Example (`when` keyword is a **guard**)

```
let f x y =
  match (x,y) with
    | (a,b) when (a=b) -> "=" // equal
    | (a,b) -> "!=";; // unequal
```

Exercises

Translate into patterns

```
let f x y =
  if y = 0 then x
  else if x = 0 then y
    else x / y;;

let f (L : int list) =
  if L.IsEmpty then []
  else if L.Tail.IsEmpty then L
    else [L.Tail.Head];;
```

Exercises

Translate into patterns

```
let rec snoc a (L : 'a list) =
  if L.IsEmpty then [a]
  else L.Head::snoc a L.Tail;;

let rec rac (L : 'a list) =
  if L.Tail.IsEmpty then L.Head
  else rac L.Tail;;

let rec rdc (L : 'a list) =
  if L.Tail.IsEmpty then []
  else L.Head::rdc(L.Tail);;
```

Local Vars

Local Variable Definitions

Example

```
let f x =
  let y = 2
  in
  x+y;;
```

- For readability, use multiple lines and indent function expressions like this
- The variable **y** has only the scope of function **f**

Functional Languages

Example

```
> let days2ms days =
    let hours = days * 24.0
    let minutes = hours * 60.0
    let seconds = minutes * 60.0
    in
    seconds * 1000.0;;
val days2ms : days:float -> float
> days2ms 10.0;;
val it : float = 864000000.0
```

- The `in` keyword allows you to break up long expressions and name the pieces
- This can make code more readable

Debugging with let...in

Example

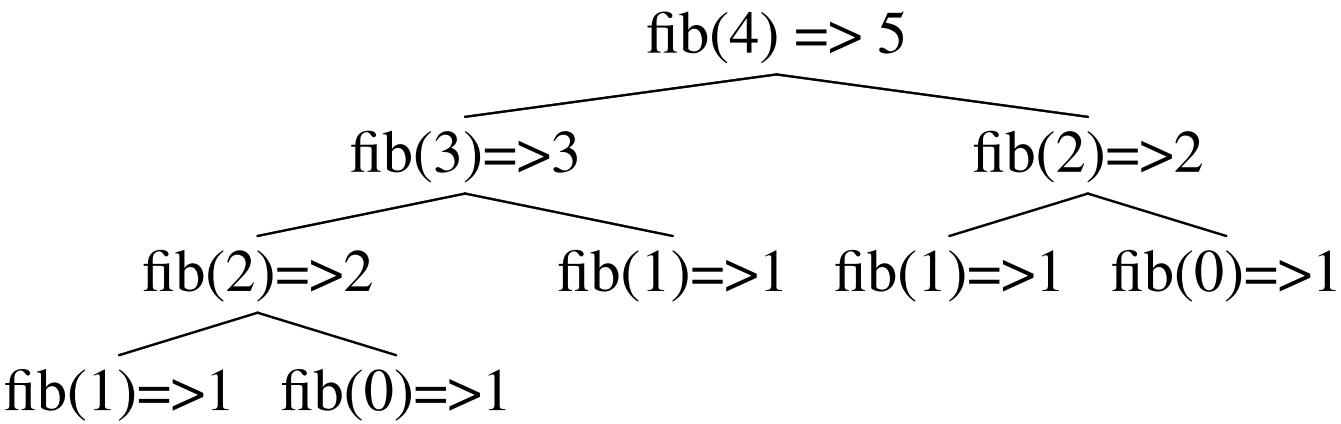
```
let rec fac n =
  match n with
    | 0 | 1 -> 1
    | n ->
        let _ = printfn "%d" n
        in
        n * fac(n-1);;
```

- Notice the newline and indention
- `let _ =` ... binds the unit result of **printfn** to nothing
- Returns the value `n * fac(n-1)`

Exponential time trace

Example

```
let rec fib n =
  match n with
    | 0 | 1 -> 1
    | n -> fib (n-1) + fib (n-2);;
```



Patterns with linear time

Example

```
let rec fib n =
  match n with
    | 0 -> (1, 0)
    | 1 -> (1, 1)
    | 2 -> (2, 1)
    | n ->
      let (a, b) = fib (n-1)
      in
      (a+b, a);;
```

What is the trace for `fib 4`?

Patterns with in

- Cases more clearly stated with patterns than conditionals.
- Patterns with **let...in** can improve program readability.
- The **halve** function divides a list into two lists as a tuple.
- **halve [1;2;3;4;5]** returns **([1;3;5],[2;4])**
- **fst (halve cs)** is first element of tuple **halve(cs)**
- Recall that **fst ([1;3;5],[2;4])** is **[1;3;5]**.

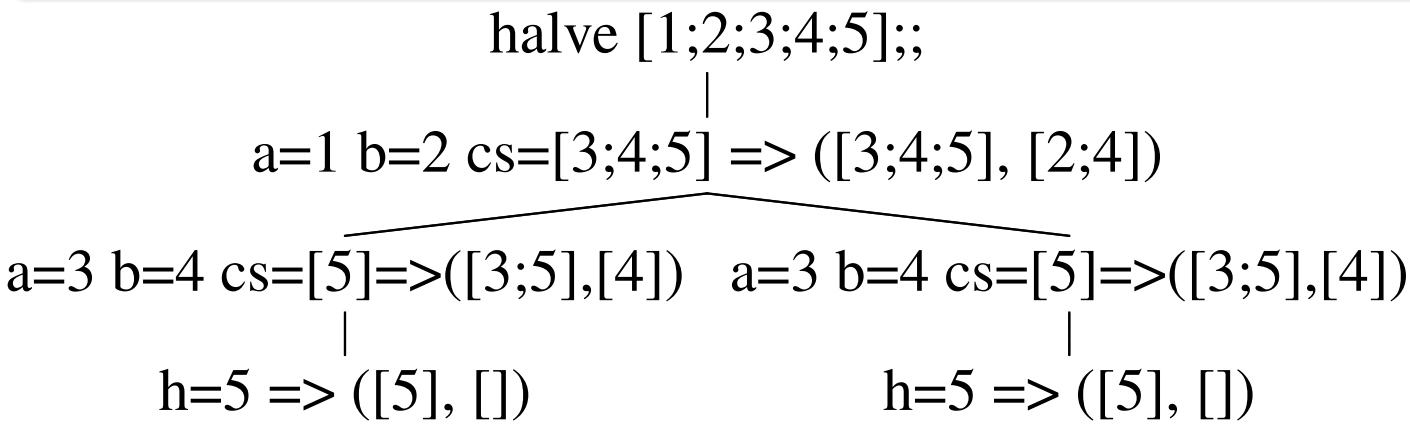
Example

```
let rec halve L =
  match L with
    | [] -> ([], [])
    | h::[] -> ([h], [])
    | a::b::cs ->
      (a::fst (halve cs), b::snd (halve cs));;
```

No let...in

Example

```
let rec halve L =
  match L with
    | [] -> ([], [])
    | h::[] -> ([h], [])
    | a::b::cs ->
      (a::fst (halve cs), b::snd (halve cs));;
```



With let...in

Example

```
let rec halve L =
  match L with
    | [] -> ([], [])
    | a::[] -> ([a], [])
    | a::b::cs ->
      let (x, y) = halve(cs)
      in
      (a::x, b::y);;
```

- Runs in linear time instead of exponential time
- Pattern matching usually gives a better solution

Sorting Example

Merge Sort

- The halve function divides a list into two nearly-equal parts
- This is the first step in a merge sort
- For practice, we will look at the rest

The Merge

Example

```
let rec merge (L:int list * int list) =
  match L with
    | ([], ys) -> ys
    | (xs, []) -> xs
    | (x::xs, y::ys) when x < y ->
      x::merge (xs, y::ys)
    | (x::xs, y::ys) -> y::merge (x::xs, ys);;
```

- Merges two *sorted* lists

Merge At Work

Example

```
> merge ([2], [1;3]);;
val it : int list = [1; 2; 3]

> merge ([1;3;4;7;8], [2;3;5;6]);;
val it : int list = [1; 2; 3; 3; 4; 5; 6; 7; 8]
```

Merge Trace

Example

```
let rec merge (L:int list * int list) =
  let _ = printfn "L=%A" L
  in
  match L with
    | ([], ys) -> ys
    | (xs, []) -> xs
    | (x::xs, y::ys) when x < y ->
      x::merge (xs, y::ys)
    | (x::xs, y::ys) -> y::merge (x::xs, ys);;
```

```
> merge ([1;3;4], [2;3;5]);;
L=([1; 3; 4], [2; 3; 5])
L=([3; 4], [2; 3; 5])
L=([3; 4], [3; 5])
L=([3; 4], [5])
L=([4], [5])
L=([], [5])
val it : int list = [1; 2; 3; 3; 4; 5]
```

mergeSort

Example

```
let rec mergeSort L =
  match L with
    | [] -> []
    | a::[] -> [a]
    | theList ->
      let (x, y) = halve theList
      in
      merge (mergeSort x, mergeSort y);;
```

- Merge sort of a list
- Type is `mergeSort :L:int list -> int list`

mergeSort Example

Example

```
> mergeSort [6;3;4;1;7];;
L=([6], [7])
L=([], [7])
L=([6; 7], [4])
L=([6; 7], [])
L=([3], [1])
L=([3], [])
L=([4; 6; 7], [1; 3])
L=([4; 6; 7], [3])
L=([4; 6; 7], [])
val it : int list = [1; 3; 4; 6; 7]
```

- Runtime is $O(n \lg n)$
- n operations at each list size
- List is cut in half $\lg n$ times before reaching a base case

Nested mergeSort

Example

```
let rec mergeSort L =
  match L with
    | [] -> []
    | _::[] -> L
    | theList ->
      let rec halve L =
        match L with
          | [] -> ([], [])
          | a::[] -> ([a], [])
          | a::b::cs ->
            let (x, y) = halve cs
            (a::x, b::y)
      let rec merge (L1, L2) =
        match (L1, L2) with
          | ([], ys) -> ys
          | (xs, []) -> xs
          | (x::xs, y::ys) when x<y -> x::merge(xs,y::ys)
          | (x::xs, y::ys) -> y::merge(x::xs,ys)
      let (x, y) = halve theList
      in
      merge (mergeSort x, mergeSort y);;
```