# 301AA - Advanced Programming

Lecturer: **Andrea Corradini**
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

***AP-15**: Laziness, Algebraic Datatypes and Higher Order Functions*

# Laziness

- Haskell is a **lazy** language
- Functions and data constructors (also user-defined ones) don't evaluate their arguments until they need them

```
cond True  t e = t
cond False t e = e
cond :: Bool -> a -> a -> a

cond True [] [1..]   => []
```

- Programmers can write control-flow operators that have to be built-in in eager languages

Short-circuiting "or"

```
(||) :: Bool -> Bool -> Bool
True  || x = True
False || x = x
```

# List Comprehensions

- Notation for constructing new lists from old ones:

```
myData = [1,2,3,4,5,6,7]

twiceData = [2 * x | x <- myData]
-- [2,4,6,8,10,12,14]

twiceEvenData = [2 * x| x <- myData, x `mod` 2 == 0]
-- [4,8,12]
```

- Similar to "set comprehension"

$$\{ x \mid x \in A \;\wedge\; x > 6 \}$$

# More on List Comprehensions

```
ghci> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20] -- more predicates

ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]    -- more lists

length xs = sum [1 | _ <- xs] -- anonymous (don’t care) var

-- strings are lists…
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

# Datatype Declarations

- Examples

```
data Color = Red | Yellow | Blue
```

elements are Red, Yellow, Blue

```
data Atom = Atom String | Number Int
```

elements are Atom “A”, Atom “B”, …, Number 0,  ...

```
data List     = Nil   |    Cons  (Atom, List)
```

elements are Nil, Cons(Atom “A”, Nil), …
Cons(Number 2, Cons(Atom(“Bill”), Nil)), ...

- General form

```
data <name> = <clause> | … | <clause>
<clause> ::= <constructor> | <contructor> <type>
```

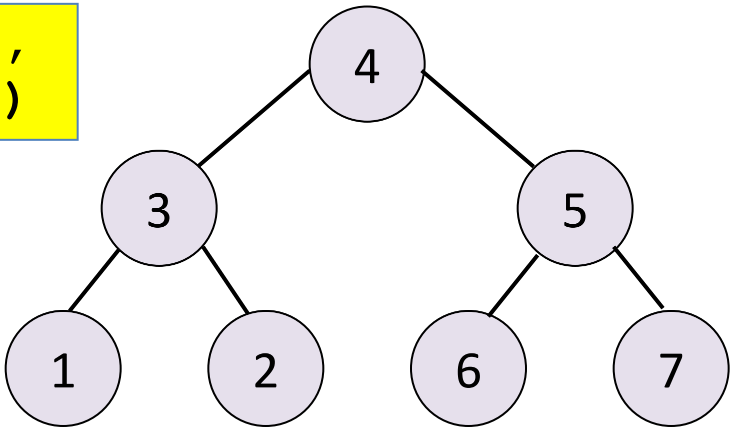
– Type name and constructors must be Capitalized.

# Datatypes and Pattern Matching

- Recursively defined data structure

```
data Tree = Leaf Int | Node (Int, Tree, Tree)
```

```
Node(4, Node(3, Leaf 1, Leaf 2),
        Node(5, Leaf 6, Leaf 7))
```



- Constructors can be used in Pattern Matching
- Recursive function

```
sum (Leaf n) = n
sum (Node(n,t1,t2)) = n + sum(t1) + sum(t2)
```

# Case Expression

- Datatype

```
data Exp = Var Int | Const Int | Plus (Exp, Exp)
```

- Case expression

```
case e of
     Var n ->  …
     Const n -> …
     Plus(e1,e2) -> …
```

  - Indentation matters in case statements in Haskell.

# Function Types in Haskell

In Haskell, **`f :: A -> B`** means for every x ∈ A,

$$f(x) = \begin{cases} \text{some element } y = f(x) \in B \\ \text{run forever} \end{cases}$$

In words, “if f(x) terminates, then f(x) ∈ B.”

In ML, functions with type A → B can throw an exception or have other effects, but not in Haskell

```
Prelude> :t not     -- type of some predefined functions
not :: Bool -> Bool
Prelude> :t (+)
(+) :: Num a => a -> a -> a
Prelude> :t (:)
(:) :: a -> [a] -> [a]
Prelude> :t elem
elem :: Eq a => a -> [a] -> Bool
```

**Note: if *f* is a standard binary function, `` `f` `` is its infix version**

**If *x* is an infix (binary) operator, *(x)* is its prefix version.**

# From loops to recursion

- In functional programming, **for** and **while** loops are replaced by using recursion
- **Recursion**: subroutines call themselves directly or indirectly (mutual recursion)

```
length' [] = 0
length' (x:s) = 1 + length'(s)

// definition using guards and pattern matching
take' :: (Num i, Ord i) => i -> [a] -> [a]
take' n _
    | n <= 0   = []
take' _ []     = []
take' n (x:xs) = x : take' (n-1) xs
```

# Higher-Order Functions

- Functions that take other functions as arguments or return a function as a result are **higher-order functions**.
- Pervasive in functional programming

```
applyTo5 :: Num t1 => (t1 -> t2) -> t2 -- function as arg
applyTo5 f = f 5
> applyTo5 succ     =>   6
➢ applyTo5 (7 +)    =>  12

applyTwice :: (a -> a) -> a -> a  -- function as arg and res
applyTwice f x = f (f x)
> applyTwice (+3) 10   => 16
> applyTwice (++ " HAHA") "HEY" => "HEY HAHA HAHA"
> applyTwice (3:) [1]           =>  [3,3,1]
```

# Higher-Order Functions

- Can be used to support alternative syntax
- Example: From functional to stream-like

```
(|>) :: t1 -> (t1 -> t2) -> t2
(|>) a f = f a

> length ( tail ( reverse [1,2,3]))  =>  2

> [1,2,3] |> reverse |> tail |> length   =>  2
```

# Higher-Order Functions… everywhere

- Any curried function with more than one argument is higher-order: applied to one argument it returns a function

```
(+)  :: Num a => a -> a -> a
> let f = (+) 5          // partial application
>:t f      ==>   f :: Num a => a -> a
> f 4      ==>   9

elem :: (Eq a, Foldable t) => a -> t a -> Bool
> let isUpper = (`elem` ['A'..'Z'])
>:t isUpper  ==> isUpper :: Char -> Bool
> isUpper 'A' ==> True
> isUpper '0' ==> False
```

# Higher-Order Functions: the map combinator

**map**: applies argument function to each element in a collection.

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

```
> map (+3) [1,5,3,1,6]
[4,8,6,4,9]
> map (++ "!") ["BIFF", "BANG", "POW"]
["BIFF!","BANG!","POW!"]
> map (replicate 3) [3..6]
[[3,3,3],[4,4,4],[5,5,5],[6,6,6]]
> map (map (^2)) [[1,2],[3,4,5,6],[7,8]]
[[1,4],[9,16,25,36],[49,64]]
> map fst [(1,2),(3,5),(6,3),(2,6),(2,5)]
[1,3,6,2,2]
```

# Higher-Order Functions: the filter combinator

**filter**: takes a collection and a boolean predicate, and returns the collection of the elements satisfying the predicate

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

```
> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
> filter (==3) [1,2,3,4,5]
[3]
> filter even [1..10]
[2,4,6,8,10]
> let notNull x = not (null x)
  in filter notNull [[1,2,3],[],[3,4,5],[2,2],[],[],[]]
[[1,2,3],[3,4,5],[2,2]]
```

# Higher-Order Functions: the reduce combinator

**reduce** (**foldl, foldr**): takes a collection, an initial value, and a function, and combines the elements in the collection according to the function.

```
-- folds values from end to beginning of list
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)

-- folds values from beginning to end of list
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs

-- variants for non-empty lists
foldr1 :: Foldable t => (a -> a -> a) -> t a -> a
foldl1 :: Foldable t => (a -> a -> a) -> t a -> a
```

Binary function

Initial value

List/collection

# Examples

```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
foldr1 :: Foldable t => (a -> a -> a) -> t a -> a
```

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs

maximum' :: (Ord a) => [a] -> a
maximum' = foldr1 (\x acc -> if x > acc then x else acc)

reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []

product' :: (Num a) => [a] -> a
product' = foldr1 (*)

filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x acc > if p x then x : acc else acc) []

head' :: [a] -> a
head' = foldr1 (\x _ -> x)

last' :: [a] -> a
last' = foldl1 (\_ x -> x)
```

*The remaining slides of this presentation were not presented during the lesson. They are left here for the interested reader.*

# On efficiency

- Iteration and recursion are equally powerful in theoretical sense: Iteration can be expressed by recursion and vice versa
- Recursion is the natural solution when the solution of a problem is defined in terms of simpler versions of the same problem, as for **tree traversal**
- In general a procedure call is *much more expensive* than a conditional branch
- Thus recursion is in general less efficient, but good compilers for functional languages can perform good code optimization
- Use of **combinators**, like *map*, *reduce* (*foldl, foldr*), *filter*, *foreach*,… strongly encouraged, because they are highly optimized by the compiler.

# Tail-Recursive Functions

- ***Tail-recursive functions*** are functions in which no operations follow the recursive call(s) in the function, thus the function returns immediately after the recursive call:

| *tail-recursive* | *not tail-recursive* |
|---|---|
| `int trfun()`<br>`{ …`<br>`  return trfun();`<br>`}` | `int rfun()`<br>`{ …`<br>`  return 1+rfun();`<br>`}` |

- A tail-recursive call could *reuse* the subroutine's frame on the run-time stack, since the current subroutine state is no longer needed
  - Simply eliminating the push (and pop) of the next frame will do
- In addition, we can do more for ***tail-recursion optimization***: the compiler replaces tail-recursive calls by jumps to the beginning of the function

# Tail-Recursion Optimization: Example

```
int gcd(int a, int b) // tail recursive
   { if (a==b) return a;
     else if (a>b) return gcd(a-b, b);
     else return gcd(a, b-a);
   }
```

```
int gcd(int a, int b) // possible optimization
   { start:
       if (a==b) return a;
       else if (a>b) { a = a-b; goto start; }
       else { b = b-a; goto start; }

   }
```

```
int gcd(int a, int b) // comparable efficiency
   { while (a!=b)
       if (a>b) a = a-b;
       else b = b-a;
     return a;
   }
```

# Converting Recursive Functions to Tail-Recursive Functions

- Remove the work after the recursive call and include it in some other form as a computation that is passed to the recursive call
- For example

```
reverse [] = [] -- quadratic
reverse (x:xs) = (reverse xs) ++ [x]
```

can be rewritten into a tail-recursive function:

```
reverse xs =  -- linear, tail recursive
    let rev ( [], accum ) = accum
        rev ( y:ys, accum ) = rev ( ys, y:accum )
    in rev ( xs, [] )
```

Equivalently, using the **where** syntax:

```
reverse xs =  -- linear, tail recursive
    rev ( xs, [] )
    where rev ( [], accum ) = accum
          rev ( y:ys, accum ) = rev ( ys, y:accum )
```

# Converting recursion into tail recursion: Fibonacci

- The Fibonacci function implemented as a recursive function is very inefficient as it takes exponential time to compute:

```
fib = \n -> if  n == 0 then  1
     else if n == 1 then 1
       else fib (n - 1) + fib (n - 2)
```

with a tail-recursive helper function, we can run it in O(n) time:

```
fibTR = \n ->  let fibhelper (f1, f2, i) =
                     if (n == i) then f2
                       else fibhelper (f2, f1 + f2, i + 1)
          in fibhelper(0,1,0)
```

# Comparing foldl and foldr

```
-- folds values from end to beginning of list
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)


-- folds values from beginning to end of list
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

- `foldl` is tail-recursive, `foldr` is not. But because of laziness Haskell has no tail-recursion optimization.
- `foldl'` is a variant of `foldl` where `f` is evaluated strictly. It is more efficient.

See **`https://wiki.haskell.org/Foldr_Foldl_Foldl'`**