301AA - Advanced Programming

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

Course pages:
http://pages.di.unipi.it/corradini/Didattica/AP-18/

AP-2018-19: Algebraic Datatypes and Higher Order Functions
Datatype Declarations

• Examples

  ```haskell
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

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

  – Type name and constructors must be Capitalized.
Datatypes and Pattern Matching

- Recursively defined data structure
  ```haskell
data Tree = Leaf Int | Node (Int, Tree, Tree)
```

- Constructors can be used in Pattern Matching

- Recursive function
  ```haskell
sum (Leaf n) = n
sum (Node(n,t1,t2)) = n + sum(t1) + sum(t2)
```
Case Expression

• Datatype

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

• Case expression

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

– Indentation matters in case statements in Haskell.
Function Types in Haskell

In Haskell, \( f :: A \rightarrow B \) means for every \( x \in A \),

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

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

In ML, functions with type \( A \rightarrow B \) 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, `\( \bar{f} \)` 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)

```haskell
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' _ (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

```haskell
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

```haskell
(+) :: 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.

```haskell
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
```
Examples

\[ \text{sum'} :: (\text{Num } a) \Rightarrow [a] \rightarrow a \]
\[ \text{sum'} \ x s = \text{foldl} \ (\\text{\textbackslash acc x} \rightarrow \text{acc} + x) \ 0 \ x s \]

\[ \text{maximum'} :: (\text{Ord } a) \Rightarrow [a] \rightarrow a \]
\[ \text{maximum'} = \text{foldr1} \ (\\text{\textbackslash x acc} \rightarrow \text{if } x > \text{acc} \text{ then } x \text{ else acc}) \]

\[ \text{reverse'} :: [a] \rightarrow [a] \]
\[ \text{reverse'} = \text{foldl} \ (\\text{\textbackslash acc x} \rightarrow x : \text{acc}) \ [\ ] \]

\[ \text{product'} :: (\text{Num } a) \Rightarrow [a] \rightarrow a \]
\[ \text{product'} = \text{foldr1} \ (*) \]

\[ \text{filter'} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \]
\[ \text{filter'} p = \text{foldr} \ (\\text{\textbackslash x acc} > \text{if } p \ x \text{ then } x : \text{acc} \text{ else acc}) \ [\ ] \]

\[ \text{head'} :: [a] \rightarrow a \]
\[ \text{head'} = \text{foldr1} \ (\\text{\textbackslash x _} \rightarrow x) \]

\[ \text{last'} :: [a] \rightarrow a \]
\[ \text{last'} = \text{foldl1} \ (\_ x \rightarrow x) \]
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:

  ```c
  tail-recursive
  int trfun()
  {
    ...
    return trfun();
  }

  not tail-recursive
  int rfun()
  {
    ...
    return 1+rfun();
  }
  ```

- 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

```c
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);
}
```

```c
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; }
}
```

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

- **Tail-call**: a function returns calling another function, not necessarily itself
- Optimization still possible, reusing the stack frame
- Note: Number/size of parameters can differ
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

\[
\text{reverse } [] = [] \quad \text{-- quadratic}
\]
\[
\text{reverse } (x:xs) = (\text{reverse } xs) \oplus [x]
\]

can be rewritten into a tail-recursive function:

\[
\text{reverse } xs = \quad \text{-- linear, tail recursive}
\]
\[
\text{let } \text{rev} ( [], \text{accum} ) = \text{accum}
\]
\[
\quad \text{rev} ( y:ys, \text{accum} ) = \text{rev} ( ys, y:\text{accum} )
\]
\[
in \text{rev} ( xs, [] )
\]

Equivalently, using the \textit{where} syntax:

\[
\text{reverse } xs = \quad \text{-- linear, tail recursive}
\]
\[
\text{rev} ( xs, [] )
\]
\[
\text{where } \text{rev} ( [], \text{accum} ) = \text{accum}
\]
\[
\quad \text{rev} ( y:ys, \text{accum} ) = \text{rev} ( ys, y:\text{accum} )
\]
Converting Recursive Functions to Tail-Recursive Functions

- Another example: the non-tail-recursive function computing \( \sum_{n=\text{low}}^{\text{high}} f(n) \)

\[
\text{summation} = \ (f, \ \text{low}, \ \text{high}) \rightarrow \\
\quad \text{if (low == high) then (f low) } \\
\quad \text{else (f low) + summation (f, low + 1, high)}
\]

This can be rewritten into a tail-recursive function:

\[
\text{summationTR} = \ (f, \ \text{low}, \ \text{high}, \ \text{subtotal}) \rightarrow \\
\quad \text{if (low == high) } \\
\quad \quad \text{then subtotal + (f low) } \\
\quad \quad \text{else summationTR (f, low + 1, high, subtotal + (f low))}
\]
Converting recursion into tail recursion: Fibonacci

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

\[
fib = \begin{cases} 
1 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
 fib(n - 1) + fib(n - 2) & \text{else}
\end{cases}
\]

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

\[
fibTR = \begin{cases} 
let fibhelper(f1, f2, i) = 
  \begin{cases} 
    f2 & \text{if } n = i \\
    fibhelper(f2, f1 + f2, i + 1) & \text{else}
  \end{cases}
\end{cases}
\]

\[
in fibhelper(0,1,0)
\]
Comparing \texttt{foldl} and \texttt{foldr}

\begin{Verbatim}
-- 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)
\end{Verbatim}

\begin{Verbatim}
-- 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
\end{Verbatim}
static int indexOf(char[] source, int sourceOffset, int sourceCount, 
    char[] target, int targetOffset, int targetCount, 
    int fromIndex) {

    char first  = target[targetOffset];
    int max = sourceOffset + (sourceCount - targetCount);

    for (int i = sourceOffset + fromIndex; i <= max; i++) {
        /* Look for first character. */
        if (source[i] != first) {
            while (++i <= max && source[i] != first);
        }

        /* Found first character, now look at the rest of v2 */
        if (i <= max) {
            int j = i + 1;
            int end = j + targetCount - 1;
            for (int k = targetOffset + 1; j < end && source[j] == 
            target[k]; j++, k++);

            if (j == end) {
                /* Found whole string. */
                return i - sourceOffset;
            }
        }
    }
    return -1;
}
Searching a Substring: Exploiting Laziness

isPrefixOf :: Eq a => [a] -> [a] -> Bool
-- returns True if first list is prefix of the second
isPrefixOf [] x = True
isPrefixOf (y:ys) [] = False
isPrefixOf (y:ys)(x:xs) =  
  if (x == y) then isPrefixOf ys xs else False

suffixes:: [a]-> [[a]]
-- All suffixes of s
suffixes[]     = [[]]
suffixes(x:xs) = (x:xs) : suffixes xs

or :: [Bool] -> Bool
-- (or bs) returns True if any of the bs is True
or []       = False
or (b:bs)   = b || or bs

isSubString :: [a] -> [a] -> Bool
x `isSubString` s = or [ x `isPrefixOf` t  
                        | t <- suffixes s ]