Lecture 2: types

Prof. Liang Huang
huang@qc.cs.cuny.edu
Recap of Lecture 1

- functional programming vs. imperative programming
- basic Haskell syntax
- function definition
- list and list comprehension
- recursion
- pattern matching
-- 1. length
mylength1 [] = 0
mylength1 (x:xs) = 1 + (mylength1 xs)

mylength2 xs = (sum [1 | _ <- xs])   -- note "_"

-- 2. forall
forall p [] = True
forall p (x:xs) = (p x) && (forall p xs)

-- 3. app
app [] a = a
app (x:xs) a = x:(app xs a)

-- bad:
app (x:xs) a = [x] ++ (app xs a)
ghci> interleave [1,2] [3,4]
[[1,2,3,4],[1,3,2,4],[1,3,4,2],[3,1,2,4],[3,1,4,2],[3,4,1,2]]

ghci> interleave [1,2] []
[[1,2]]

ghci> length (interleave "ab" "cdef")
15

interleave [] xs = [xs]
interleave xs [] = [xs]
interleave (x:xs) (y:ys) = [x:s | s <- (interleave xs (y:ys))] ++
[y:s | s <- (interleave (x:xs) ys)]
quickselect i [] = error "Bad" -- raise Exception  
quickselect i (p:xs) = if lenleft >= i then quickselect i left  
else if (lenleft + 1) == i then p  
else quickselect (i-1-lenleft) right  

where left = filter (< p) xs  
right = filter (>= p) xs  
lenleft = length left

using guards notation

quickselect i (p:xs)  
| lenleft >= i       = quickselect i left  
| (lenleft + 1) == i = p  
| otherwise          = quickselect (i-1-lenleft) right  

where left = filter (< p) xs  
right = filter (>= p) xs  
lenleft = length left
The guards notation

\[
f(x) = \begin{cases} 
1 & \text{if } x > 0 \\
0 & \text{otherwise} 
\end{cases}
\]

\[
f(x) = \begin{cases} 
| & x > 0 = 1 \\
| & \text{otherwise} = 0 
\end{cases}
\]

\[
f(x) = \text{if } x > 0 \text{ then } 1 \text{ else } 0
\]

quickselect i (p:xs)

\[
\begin{cases} 
| & \text{lenleft } >= i = \text{ quickselect i left} \\
| & (\text{lenleft } + 1) == i = p \\
| & \text{otherwise } = \text{ quickselect (i-1-lenleft) right} 
\end{cases}
\]

where

\[
\begin{align*}
\text{left} &= \text{ filter (< p) xs} \\
\text{right} &= \text{ filter (>= p) xs} \\
\text{lenleft} &= \text{ length left}
\end{align*}
\]
**HW 1 mergesort**

```
ghci> mergesort [2,4,1,5,3]
[1,2,3,4,5]

ghci> mergesorted [1,2] [3,4,5]
[1,2,3,4,5]
```

```haskell
mergesorted xs [] = xs
mergesorted [] ys = ys
mergesorted (x:xs) (y:ys) = if x <= y then x:(mergesorted xs (y:ys))
                         else y:(mergesorted (x:xs) ys)

mergesort [] = []
mergesort [x] = [x]
mergesort xs = mergesorted (mergesort left) (mergesort right)
    where (left, right) = (splitat xs ((length xs) `div` 2))

splitat xs 0 = ([], xs)
splitat [] i = ([], [])
splitat (x:xs) i = (x:left, right)
    where (left, right) = splitat xs (i-1)
```
HW1 mergesort (guards)

ghci> mergesort [2,4,1,5,3]
[1,2,3,4,5]

ghci> mergesorted [1,2] [3,4,5]
[1,2,3,4,5]

mergesorted xs [] = xs
mergesorted [] ys = ys
mergesorted (x:xs) (y:ys)
    | x<=y      = x:(mergesorted xs (y:ys))
    | otherwise = y:(mergesorted (x:xs) ys)

mergesort [] = []
mergesort [x] = [x]
mergesort xs = mergesorted (mergesort left) (mergesort right)
    where (left, right) = (splitat ((length xs) `div` 2) xs)

splitat 0 xs = ([], xs)
splitat i [] = ([], [])
splitat i (x:xs) = (x:left, right)
    where (left, right) = splitat (i-1) xs
another mergesort (much better)

• splitting the list in an alternating fashion: 1-3-5-7; 2-4-6-8

• much better for linked-lists (as in all functional languages)

• Marios will receive extra credit for this elegant solution

• but this solution is no longer “stable sort”

```plaintext
mergesort [] = []
mergesort [x] = [x]
mergesort xs = mergesorted (mergesort left) (mergesort right)
    where (left, right) = newsplit xs

newsplit [] = ([], [])
newsplit [x] = ([x], [])
newsplit (x:y:xs) = (x:left, y:right)
    where (left, right) = newsplit xs
```
ghci> perm [1,2,3]
[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

perm [] = [[]]
perm xs = [ x:ys | x <- xs, ys <- perm (delete x xs) ]
delete x [] = []
delete x (y:ys)
    | x == y     = ys
    | otherwise  = y:(delete x ys)
two different appends/reverses

\[
\text{app } a \; [ ] = a \\
\text{app } a \; (x:xs) = \text{app } (a++[x]) \; xs
\]

\[
\text{app'} \; [ ] \; b = b \\
\text{app'} \; (x:xs) \; b = x : (\text{app'} \; xs \; b)
\]

\[
\text{rev } [ ] = [] \\
\text{rev } (x:xs) = \text{rev } xs ++ [x]
\]

\[
\text{revapp } [ ] \; b = b \\
\text{revapp } (x:xs) \; b = \text{revapp } xs \; (x:b)
\]

\[
\text{rev'} \; a = \text{revapp } a \; [ ]
\]
two reverses and tail recursion

\[
\text{rev} \; [\;] = \; [
\text{rev} \; (x:\text{xs}) = \; \text{rev} \; \text{xs} \; ++ \; [x]
\]

\[
\text{revapp} \; [\;] \; b = \; b
\text{revapp} \; (x:\text{xs}) \; b = \; \text{revapp} \; \text{xs} \; (x:b)
\text{rev'} \; a = \; \text{revapp} \; a \; [\;]
\]

tail recursion! -- the recursive call appears at and only at the last step in a function call

optimized by compiler to be a loop
Today

- polymorphism and different approaches to typing
- types, type variables, and type classes
- currying
- defining new types
- recursive data types
**Types and Type Variables**

- :t command; the `a` in `[a]` is a type variable

```
Prelude> :t 'a'
'a' :: Char
Prelude> :t True
True :: Bool
Prelude> :t "HELLO!"
"HELLO!" :: [Char]
Prelude> :t (True, 'a')
(True, 'a') :: (Bool, Char)
Prelude> :t 4 == 5
4 == 5 :: Bool
```

```
Prelude> :t head
head :: [a] -> a
Prelude> :t last
last :: [a] -> a
Prelude> :t fst
fst :: (a, b) -> a
Prelude> :t snd
snd :: (a, b) -> b
```
Side Note: Int vs. Integer

- Int is [system] 32 or 64 bits; Integer is arbitrary precision

```haskell
Prelude> (12345678901234567890 :: Integer, 12345678901234567890 :: Int)
(12345678901234567890,-350287150)
```
Type Classes

- Num is the numeric typeclass (interface)
  - members of Num class function as numbers
- Eq is the class that supports equality test
- Ord is the class that has an ordering
  - Ord assumes Eq, but Num does not assume Ord or Eq

Prelude> let f x = x + 1
Prelude> :t f
f :: Num a => a -> a

Prelude> :t (<)
(<) :: Ord a => a -> a -> Bool

Prelude> :t (==)
(==) :: Eq a => a -> a -> Bool

Prelude> let f x = x < 5
Prelude> :t f
f :: (Num a, Ord a) => a -> Bool

Prelude> let f x = x == 5
Prelude> :t f
f :: (Num a, Eq a) => a -> Bool

Prelude> let f x = x < 5 && x == 3
Prelude> :t f
f :: (Num a, Ord a) => a -> Bool
Multi-parameter Functions

Prelude> let f (x,y) = x+y
Prelude> :t f
f :: Num a => (a, a) -> a

Prelude> let g x y = x+y
Prelude> :t g
g :: Num a => a -> a -> a

Prelude> :t (g 5)
(g 5) :: Num a => a -> a -> a
### Approaches to Typing

- **Strongly typed**: types are strictly enforced. No implicit type conversion. Preventing invalid memory access.
- **Weakly typed**: not strictly enforced.
- **Safely typed**: type-checking done at compile-time.
- **Dynamically typed**: types are checked at runtime.

<table>
<thead>
<tr>
<th></th>
<th>Weak</th>
<th>Strong</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Static</strong></td>
<td>C/C++</td>
<td>Java, ML, Haskell</td>
</tr>
<tr>
<td><strong>Dynamic</strong></td>
<td>Perl, VB</td>
<td>Python, Lisp/Scheme</td>
</tr>
</tbody>
</table>
Polymorphism

- parametric polymorphism: Haskell, ML
  - functions such as “last” work *uniformly* over different arguments
- ad hoc polymorphism: overloading (C++, Java)
  - operations behave *differently* when applied to different types
- subtype polymorphism: Java
Defining New Types

- how to simulate OOP to ensure the type is correct?
- using keyword “data” and constructors (like in C++/Java)

```haskell
areaOfCircle (_ _ d) = d ^ 2      -- bad
surface (Circle (_ _ d)) = d ^ 2  -- good

data Bool = False | True
data Shape = Circle Float Float Float
            | Square Float Float Float
            | Rectangle Float Float Float Float

constructors are analogous to functions
(very similar to Java)

Prelude> :t Circle
Circle :: Float -> Float -> Float -> Shape
Prelude> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape

surface :: Shape -> Float
surface (Circle _ _ r) = pi * r ^ 2
surface (Square _ _ r) = r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs (x2 - x1)) * (abs (y2 - y1))
```
A “heterogenous” list

```haskell
-- shape.hs
data Shape = Circle Float Float Float
  | Square Float Float Float
  | Rectangle Float Float Float Float

surface :: Shape -> Float
surface (Circle _ _ r) = pi * r ^ 2
surface (Square _ _ r) = r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs (x2 - x1)) * (abs (y2 - y1))
```

Prelude> :l "shape.hs"
[1 of 1] Compiling Main             ( shape.hs, interpreted )
Ok, modules loaded: Main.
*Main> let l = [ Circle 5 5 5, Square 5 6 6 ]
*Main> map surface l
[78.53982,36.0]
Recursive Data Types (trees)

```haskell
data Ast = ANum Integer
          | APlus Ast Ast
          | ATimes Ast Ast

eval (ANum x) = x
eval (ATimes x y) = (eval x) * (eval y)
eval (APlus x y) = (eval x) + (eval y)
```

Prelude> eval (ATimes (APlus (ANum 5) (ANum 6)) (ANum 7))
77

-- this tree corresponds to the expression \(((5+6)\times 7)\)
Recursive Data Types (trees)

```haskell
data Ast = ANum Integer
         | APlus Ast Ast
         | ATimes Ast Ast
       deriving (Show, Eq)

eval (ANum x) = x
eval (ATimes x y) = (eval x) * (eval y)
eval (APlus x y) = (eval x) + (eval y)
```

Prelude> let t = ATimes (APlus (ANum 5) (ANum 6)) (ANum 7)
Prelude> eval t
77
Prelude> t
ATimes (APlus (ANum 5) (ANum 6)) (ANum 7)
Prelude> t == t
True
Prelude> t == APlus (ANum 5) (ANum 7)
False

__str__ and __eq__ automatically implemented for recursive data types!