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
- lazy evaluation
- list and list comprehension
- recursion
- pattern matching
HWI discussions

-- 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)
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

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

\[ f(x) \]

\[ \begin{array}{l}
\text{quickselect } i \ (p:xs) \\
| \text{lenleft } \geq i \quad = \text{quickselect } i \ \text{left} \\
| (\text{lenleft } + 1) = i = p \\
| \text{otherwise} \quad = \text{quickselect } (i-1-\text{lenleft}) \ \text{right} \\
\end{array} \]

\text{where left } = \text{filter } (< p) \ xs \\
\text{right } = \text{filter } (>= p) \ xs \\
\text{lenleft } = \text{length } left \]
```haskell
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) = 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)
```
```haskell
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)
- Greg will receive extra credit for this elegant solution

```haskell
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
```
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)
HW1 tail-recursion fibonacci

fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

fibaux prev sum 0 = sum
fibaux prev sum n = fibaux sum (prev + sum) (n - 1)

fib n = fibaux 1 0 n
Today

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

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

```haskell
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
```

```haskell
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

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

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

✓ **statically 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)

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

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

```
-- 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)*7)