CIS 1904: Haskell

Polymorphism & Recursion Patterns

Logistics

- HW 1 grades will be posted tomorrow
  - **Please read comments!** Some things we did not take points off and only left a comment for this week will get points off in future weeks.
- HW 3 will be available this evening
- Reminder: ChatGPT and Copilot are **<u>not</u>** permitted on homework or in-class assignments

Polymorphism

Polymorphism

A symbol is *polymorphic* if it can have more than one type

- Parametric polymorphism
- Ad-hoc polymorphism
- Others not as directly supported in Haskell
    - (Arbitrary) subtyping

Parametric Polymorphism

```
data IntList
    = Nil
    | Cons Int IntList


data CharList
    = Nil
    | Cons Char CharList
```

Parametric Polymorphism

```haskell
data IntList
    = Nil
    | Cons Int IntList
```

```haskell
data CharList
    = Nil
    | Cons Char CharList
```

```haskell
data List a
    = Nil
    | Cons a (List a)
```

Parametric Polymorphism

```
data List a
    = Nil
    | Cons a (List a)
```

Parametric Polymorphism

```
data List a
    = Nil
    | Cons a (List a)
```

`List` is a type *constructor*: it takes an argument and returns a type.

Parametric Polymorphism

```
data List a
    = Nil
    | Cons a (List a)
```

a is a type *variable*: we can substitute any type in for it

Parametric Polymorphism

```
data List a
    = Nil
    | Cons a (List a)
```

Note: this is our custom list definition; the usual Haskell one is `[] a`, or `[a]`

Parametric Polymorphism

- Takes a type as a *parameter*
- Must be able to work with any type*

*we will see a caveat when we get to typeclasses

Parametric Polymorphism

- Takes a type as a *parameter*
- Must be able to work with any type*

How many functions are there with type `a -> a`?

Parametric Polymorphism

- Takes a type as a *parameter*
- Must be able to work with any type*

How many functions are there with type `a`?

Parametric Polymorphism

```
data WeatherForecast
    = Forecast Weather
    | Unknown


data FailableDouble
    = Failure
    | OK Double
```

Parametric Polymorphism

```haskell
data WeatherForecast
    = Forecast Weather
    | Unknown


data FailableDouble
    = Failure
    | OK Double
```

```haskell
data Maybe a
    = Nothing
    | Just a
```

Parametric Polymorphism

```
safeDiv :: Double -> Double -> Maybe Double
safeDiv _ 0 = Nothing
safeDiv m n = Just (m / n)
```

Parametric Polymorphism

```haskell
head :: [a] -> a
head [] = errorEmptyList "head"
head (x : _) = x
```

```haskell
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x : _) = Just x
```

Aside

Side note: do not use partial functions in this class!

- It's error prone
- Type does not make it obvious it may crash

Instead, you should:

- pattern match to avoid having to call functions like `head`
- use `Maybe`

Ad Hoc Polymorphism

- Idea: Using the same interface (in Haskell, the same set of function names) for a related concept across multiple types
- Example: `+`
    - `1 + 0`   `+ :: Int -> Int -> Int`
    - `2.5 + 3.8`   `+ :: Double -> Double -> Double`
- In Haskell, this is done via *typeclasses*

Recursion Patterns

Recursion

```
squares :: [Int] -> [Int]
squares [] = []
squares (x : xs) = x * x : squares xs
```

```
lengths :: [[a]] -> [Int]
lengths [] = []
lengths (x : xs) = length x : lengths xs
```

Recursion

```
squares :: [Int] -> [Int]
squares [] = []
squares (x : xs) = x * x : squares xs
```

→ `map`

```
lengths :: [[a]] -> [Int]
lengths [] = []
lengths (x : xs) = length x : lengths xs
```

Recursion

```
squares :: [Int] -> [Int]
squares [] = []
squares (x : xs) = x * x : squares xs
```

→ `map`

What is the type of `map`?

```
lengths :: [[a]] -> [Int]
lengths [] = []
lengths (x : xs) = length x : lengths xs
```

Recursion

```
squares :: [Int] -> [Int]
squares [] = []
squares (x : xs) = x * x : squares xs
```

→ `map :: (a -> b) -> [a] -> [b]`

```
lengths :: [[a]] -> [Int]
lengths [] = []
lengths (x : xs) = length x : lengths xs
```

```
map :: (a -> b) -> [a] -> [b]
```

- Applies a function to each element of a list
  - Remember, functions are *first class*
  - Functions can be anonymous or named
    - e.g., `add1` vs `(\x -> 1 + x)`
    - For historical reasons, we read `\` as “lambda” here
      - Look up the lambda calculus if you’re curious
    - `\x ->` is like `fun x =>` in OCaml
  - Functions can be operator sections `(+3)`
    - `map (+3) [0,1,2] == [0+3,1+3,2+3] == [3,4,5]`

```
map :: (a -> b) -> [a] -> [b]
```

- Pros
  - Much more concise
  - Avoids duplicated work
    - avoids potential mistakes
    - allows reuse of optimizations
  - Self-documenting

Filter

```
upperOnly :: [Char] -> [Char]
upperOnly [] = []
upperOnly (x : xs)
  | isUpper x = x : upperOnly xs
  | otherwise = upperOnly xs
```

Filter

```
upperOnly :: [Char] -> [Char]
upperOnly [] = []
upperOnly (x : xs)
  | isUpper x = x : upperOnly xs
  | otherwise = upperOnly xs
```

```
upperOnly = filter isUpper
```

Filter

```
filter :: (a -> Bool) -> [a] -> [a]
```

Useful for filtering elements of a list.

Fold

Useful for combining all the elements of a list in some way.

Fold

```haskell
sum :: [Int] -> Int
sum [] = 0
sum (x : xs) = x + sum xs
```

```haskell
foldr :: (a -> b) -> b -> [a] -> b
```

Fold

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x : xs) = f x (foldr f z xs)
```

z is the “default” value

f tells us how to combine the elements of the list and the default value

Fold

```
sum :: [Int] -> Int
sum [] = 0
sum (x : xs) = x + sum xs
```

```
sum = foldr (+) 0
```

Fold

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z [] = z
foldr f z (x : xs) = f x (foldr f z xs)
```

```
foldr add 0 [1,2,3]
add 1 (foldr add 0 [2,3])
add 1 (add 2 (foldr add 0 [3]))
add 1 (add 2 (add 3 (foldr 0 [])))
add 1 (add 2 (add 3 0))
add 1 (add 2 3)
add 1 5
6
```

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))
```

```
foldl f z [a,b,c] == f (f (f z a) b) c
```

Fold

```
foldl f z [a,b,c] == f (f (f z a) b) c

foldl add 0 [1,2,3]
let x1 = 0+1 in (foldl add x1 [2,3])
let x1 = 0+1 in (let x2 = x1+2 in (foldl add x2 [3]))
let x1 = 0+1 in (let x2 = x1+2 in (let x3 = x2+3 in (foldl add x3 [])))
let x1 = 0+1 in (let x2 = x1+2 in (let x3 = x2+3 in x3))
let x1 = 0+1 in (let x2 = x1+2 in (x2+3))
let x1 = 0+1 in ((x1+2)+3)
((0+1)+2)+3
(1+2)+3
3+3
6
```

Fold

Haskell provides a third option: `foldl’`

- Designed to be space efficient but semantically the same as `foldl`
- There are very few reasons to use `foldl`
    - `foldl’` is recommended by Haskell

Which is preferred between `foldl’` and `foldr`?

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))
```

```
foldl f z [a,b,c] == f (f (f z a) b) c
```

These are only equivalent if f is commutative and associative!

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))

foldl f z [a,b,c] == f (f (f z a) b) c



foldr (+) 0 [4,5,6]


foldl (+) 0 [4,5,6]
```

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))

foldl f z [a,b,c] == f (f (f z a) b) c


foldr (+) 0 [4,5,6] = 15

foldl (+) 0 [4,5,6] = 15
```

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))

foldl f z [a,b,c] == f (f (f z a) b) c


foldr (-) 0 [1,1,1] ==

foldl (-) 0 [1,1,1] ==
```

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))

foldl f z [a,b,c] == f (f (f z a) b) c


foldr (-) 0 [1,1,1] == 1

foldl (-) 0 [1,1,1] == -3
```

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))

foldl f z [a,b,c] == f (f (f z a) b) c


foldr (\x y -> (x + y) / 2) 0 [8,4,16]

foldl (\x y -> (x + y) / 2) 0 [8,4,16]
```

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))

foldl f z [a,b,c] == f (f (f z a) b) c


foldr (\x y -> (x + y) / 2) 0 [8,4,16] == 7

foldl (\x y -> (x + y) / 2) 0 [8,4,16] == 10
```

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))

foldl f z [a,b,c] == f (f (f z a) b) c


foldr (++) "z" ["a","b","c"] ==


foldl (++) "z" ["a","b","c"] ==
```

Fold

```
foldr f z [a,b,c] == f a (f b (f c z))

foldl f z [a,b,c] == f (f (f z a) b) c


foldr (++) "z" ["a","b","c"] == "abcz"


foldl (++) "z" ["a","b","c"] == "zabc"
```

Fold

Which is preferred between `foldl’` and `foldr`?

When they are equivalent, typically `foldr`.

Foldr

- follows natural structure of the list
    - `f a (f b (f c z))`
- Allows for short-circuiting
    - `False && (b && (c && z))`
- We can sometimes use it with infinite lists!
    - We will discuss more in the laziness unit

Examples

```
reverse :: [Int] -> [Int]
reverse xs =
```

Examples

```
reverse :: [Int] -> [Int]
reverse xs = foldr comb [] xs
    where
        comb x res = res ++ [x]
```

Examples

```
reverse :: [Int] -> [Int]
reverse = foldr comb []
    where
        comb x res = res ++ [x]
```

Examples

```
reverse :: [Int] -> [Int]
reverse = foldr comb []
    where
        comb x res = res ++ [x]
```

What would we need to change to implement this with `foldl`?

```
foldl f z [a,b,c] == f (f (f z a) b) c
```

Examples

```
reverse :: [Int] -> [Int]
reverse = foldl comb []
    where
        comb res x = x : res
```

```
foldl f z [a,b,c] == f (f (f z a) b) c
```