CIS 1904: Haskell

Higher-Order Patterns

Logistics

- HW 4 will be released tonight
- Autograder should show total autograded score now

Higher-order functions

Recall: functions in Haskell are *first-class*

- They can be passed around as inputs to other functions
- They can be returned as outputs of other functions

Higher-order functions

Recall: functions in Haskell are *first-class*

- They can be passed around as inputs to other functions
- They can be returned as outputs of other functions

A function that takes in OR returns another function is called *higher-order*.

Higher-order functions

Examples:

- `map :: (a -> b) -> [a] -> [b]`
- `filter :: (a -> Bool) -> [a] -> [a]`
- `fold :: (a -> b -> b) -> b -> [a] -> b`

These all take in functions and, if partially applied, return functions.

Higher-order functions

Example:

```
compose :: (b -> c) -> (a -> b) -> a -> c
compose f g = \x -> f (g x)
```

Recall: `\x ->` syntax is used for anonymous functions, like `fun x =>` in OCaml

Higher-order functions

Example:

```
compose :: (b -> c) -> (a -> b) -> a -> c
compose f g = f . g
```

The standard library provides function composition, written as `.` .

Higher-order functions

Composition helps write code that is:

- Concise
- Understandable
- In keeping with the “wholemeal” programming style

```
foo :: String -> String
foo s = length (filter (== ‘C’) (toUpper s))
```

Higher-order functions

Composition helps write code that is:

- Concise
- Understandable
- In keeping with the “wholemeal” programming style

```
foo :: String -> String
foo s = length (filter (== ‘C’) (toUpper s))

foo = length . filter (== ‘C') . toUpper
```

Higher-order functions

Composition helps write code that is:

- Concise
- Understandable
- In keeping with the "wholemeal" programming style

```
foo :: String -> String
foo s = length (filter (== ‘C’) (toUpper s))

foo = length . filter (== ‘C') . toUpper
```

Partial application

```haskell
add :: Int -> Int -> Int
add x y = x + y
```

All functions in Haskell take one argument*!

Partial application

```haskell
add :: Int -> Int -> Int
add x y = x + y
```

All functions in Haskell take one argument*!

```haskell
add :: Int -> (Int -> Int)
```

```haskell
add 0 :: Int -> Int
```

Partial application

```haskell
add :: Int -> Int -> Int
add x y = x + y
```

All functions in Haskell take one argument*!

```haskell
add :: Int -> (Int -> Int)
```

```haskell
add 0 :: Int -> Int
```

Note: `(Int -> Int) -> Int` is **not** the same as `Int -> Int -> Int`.
`->` is *right-associative*.

Partial application

`->` is right-associative.

`Int -> Int -> Int` is the same as `Int -> (Int -> Int)`

Function *application* is left-associative.

`add 0 1` is the same as `(add 0) 1`

Partial application

Note: Haskell lets us write anonymous functions like `\x y z -> …`

This is just syntactic sugar for `\x -> (\y -> (\z -> …))`.

Similarly, `add x y =` is just syntactic sugar for `add = \x -> \y -> .`

Currying

```
add :: (Int, Int) -> Int
add (x,y) = x + y

add :: Int -> Int -> Int
add x y = x + y
```

These are equivalent! We call going from the first to the second *currying*, after Haskell Curry, and the reverse *uncurrying*.

Currying makes partial application easier.

Partial application

When writing functions, consider:

Which argument are you most likely to want to partially apply with?

`filter f xs` – we often want to filter multiple lists using the same criterion

`filter xs f` – we rarely want to filter the same original list using multiple criteria

Eta reduction

Eta reduction: removing unnecessary function abstractions.

`\x -> f x` is equivalent to `f`

Eta reduction

Eta reduction: removing unnecessary function abstractions.

`\x -> f x` is equivalent to `f`

```
foo :: String -> String
foo xs = map toUpper xs
```

Eta reduction

Eta reduction: removing unnecessary function abstractions.

`\x -> f x` is equivalent to `f`

```
foo :: String -> String
foo xs = map toUpper xs

foo = \xs -> map toUpper xs
```

Eta reduction

Eta reduction: removing unnecessary function abstractions.

`\x -> f x` is equivalent to `f`

```
foo :: String -> String
foo xs = map toUpper xs

foo = \xs -> map toUpper xs

foo = map toUpper
```

Prefix Operators

```
add :: Int -> Int -> Int
```

- `add 0 1`
- ``0 `add` 1``

Infix Operators

```
(+) :: Int -> Int -> Int
```

- `(+) 0 1`
- `0 + 1`

Haskell lets us use *operator sections*, i.e., partially apply these on either side:

- `map (+ 1) xs`
- `filter (0 <) xs`