FIT2102
Programming Paradigms
Lecture 6

(Haskell . PureScript . λCalc . JavaScript) yourBrain

Faculty of Information Technology

MONASH University

Learning Outcomes

- Apply reduction steps to Lambda Calculus expressions and describe how this leads to a general model for computation
- Describe how Haskell-like languages (and in particular PureScript) improve over JavaScript syntax to better support the functional programming paradigm
- Describe how Haskell uses type inference to perform strong type checking with minimal annotation
- Create small programs in Haskell using:
  - recursion
  - pattern matching
  - guards
  - algebraic data type definitions

Lambda Calculus - Recap

$\mathrm{I} = \lambda x\,.\,x$ lambda calculus expression

```
I = x => x
```

JavaScript

Lambda Calculus - application

$(\lambda x . x)\, y$ lambda calculus expression

```
(x => x)(y)
```

JavaScript

Lambda Calculus

$\mathrm{I} = \lambda\, x\, .\, x$ I-Combinator

$\mathrm{K} = \lambda\, x\, y\, .\, x$ K-Combinator

$$\begin{aligned} \mathrm{K\,I} &= (\lambda\, x\, y\, .\, x)\, (\lambda\, x\, .\, x) \\ &= \lambda\, y\, .\, x\ [x := \lambda\, x\, .\, x] \\ &= \lambda\, y\, .\ (\lambda\, x\, .\, x) \\ &= \lambda\, yx\, .\, x \\ &= \lambda\, xy\, .\, y \end{aligned}$$

⇦ Beta reduction

⇦ Equivalent due to currying

⇦ Alpha equivalence

Lambda's are always curried, i.e.:

$$\lambda\, x\, y\, .\, x = \lambda\, x\, .\, \lambda y\, .\, x$$

Variables are free or bound:

u bound

$$\lambda\, x\, y\, .\, x\, z = \lambda\, a\, b\, .\, a\, c = \lambda\, u\, .\, \lambda v\, .\, u\, w$$

⇧

↖ ↑ ↗

Alpha equivalence

w free

$$\lambda\, x\, .\, M x = M$$

⇧

Eta conversion

Lambda Calculus

Three operations:

- Alpha Equivalence
  - expressions are equivalent if their variables are renamed
- Beta Reduction
  - application of functions involves substituting the argument into the expression
- Eta Conversion
  - wrapping a simple lambda around an expression does not change the expression

Lambda expressions are anonymous *(although we've been making "macros" (e.g. I,K) with =)*

- They can't refer to themselves! *(but there's a trick for recursion: the Y-combinator)*

Lambda Calculus Reduction Recap

```
(λz. z) (λa. a a) (λz. z b)
```

Lambda Calculus Reduction Recap

`( (λz.z) (λa.aa) )    (λz.zb)` ⇦ Function application is left-associative.

Lambda Calculus Reduction Recap

```
(λz. z) (λa. a a) (λz. z b)

 ⇒ (λz.z [z := λa. a  a]) (λz. z b)        ⇦ BETA Reduction

 ⇒ (λa. a a) (λz. z b)

 ⇒ λa [a := λz. z b]. a a                   ⇦ BETA Reduction

 ⇒ (λz. z b) (λz. z b)

 ⇒ (λz [z := λz. z b]. z b)                 ⇦ BETA Reduction

 ⇒ (λz. z b) b

 ⇒ λz [z := b]. z b                         ⇦ BETA Reduction

 ⇒ b b
```

Beta Reduction vs Eta Conversion

```
(λz. z b) x
⇒ λz [z := x]. z b
⇒ x b
```

⇦ BETA Reduction:

```
λz. z b on its own
is irreducible
```

```
(λz. b z) x
⇒ b x
```

⇦ ETA Conversion:

```
λz. b z == b
```

```
z =>
  function (x) {
    return some expression involving x
  }
  (z)
```

} b

Divergence

```
(λx. x x) (λx. x x)

⇒ (λx [x := (λx.x x) ]. x x)
⇒ (λx. x x) (λx. x x)

⇒ (λx [x := (λx.x x) ]. x x)
⇒ (λx. x x) (λx. x x)

⇒ (λx [x := (λx.x x) ]. x x)
⇒ (λx. x x) (λx. x x)

⇒ (λx [x := (λx.x x) ]. x x)
⇒ (λx. x x) (λx. x x)

...
```

Can keep on applying reduction rules forever !

Lecture Activity 1

To be announced...

Lambda Calculus and Computation

```
TRUE  = λxy. x
FALSE = λxy. y

IF = λbtf. b t f

AND = λxy. IF x  y FALSE
OR  = λxy. IF x TRUE y
NOT = λx . IF x FALSE TRUE
```

PureScript

If you want to try out the examples on the next couple of slides you need to install PureScript:

```
$ npm install -g purescript pulp bower
```

Download purescriptstartercode.zip and unzip

```
$ cd purescriptstartercode
```

Your code goes under the import statements in src/Main.purs

```
$ pulp run
```

Some PureScript code:

```
fibs 0 = 1                                -- two base cases,
fibs 1 = 1                                -- resolved by pattern matching
fibs n = fibs (n-1) + fibs (n-2)-- recursive definition

fibsArray = map fibs (0..9)

main = log ( show fibsArray )
```

> [1,1,2,3,5,8,13,21,34,55]

Some PureScript code:

```
fibs 0 = 1
fibs 1 = 1
fibs n = fibs (n-1) + fibs (n-2)



main = log ( map fibs (0..9) )
```

> [1,1,2,3,5,8,13,21,34,55]

Some PureScript code:

```
fibs 0 = 1
fibs 1 = 1
fibs n = fibs (n-1) + fibs (n-2)


main = log $ map fibs $ 0..9
```

> [1,1,2,3,5,8,13,21,34,55]

$ is an "infix" function with low precedence:

f $ x = f x

Allows us to eliminate some ( )

Think of expressions like these as "pipelines of functions", chained from right-to-left

PureScript source

```
fibs 0 = 1
fibs 1 = 1
fibs n = fibs (n-1) + fibs (n-2)
```

Generated JavaScript

```
var fibs = function (v) {
    if (v === 0) {
        return 1;
    };
    if (v === 1) {
        return 1;
    };
    return
        fibs(v - 1 | 0)
        + fibs(v - 2 | 0) | 0;
};
```

Tail Recursive Form

```
fibs :: Int -> Int
fibs n = fibsTC n 0 1
 where
 fibsTC 0 _ b = b
 fibsTC i a b = fibsTC (i-1) b (a+b)
```

local scope

```
var fibs = function (n) {
    var fibsTC = function ($copy_v) {
        return function ($copy_v1) {
            return function ($copy_b) {
                var $tco_var_v = $copy_v;
                var $tco_var_v1 = $copy_v1;
                var $tco_done = false;
                var $tco_result;
                function $tco_loop(v, v1, b) {
                    if (v === 0) {
                        $tco_done = true;
                        return b;
                    };
                    $tco_var_v = v - 1 | 0;
                    $tco_var_v1 = b;
                    $copy_b = v1 + b | 0;
                    return;
                };
                while (!$tco_done) {
                    $tco_result = $tco_loop($tco_var_v,
                                            $tco_var_v1, $copy_b);
                };
                return $tco_result;
            };
        };
    };
    return fibsTC(n)(0)(1);
};
```

Introduction to Haskell

syntax is very similar to Purescript

- Only variance from the previous example was the list range operator - PureScript: `1..10` vs Haskell: `[1..10]`

repl available with ghci (if you installed stack, run with `> stack ghci`)

can compile to native code (GHC) or JavaScript (GHCJ)

- But with a run-time system

uses lazy evaluation by default

is strongly typed with a powerful type inference system

Haskell 101

Make a file: fibs.hs

```
fibs 0 = 1                       -- two base cases,
fibs 1 = 1                       -- resolved by pattern matching
fibs n = fibs (n-1) + fibs (n-2) -- recursive definition
```

```
$ stack ghci fibs.hs
> fibs 6
13

> fibs 6 == 13
True

> if fibs 6 == 13 then "yes" else "no"
"yes"

> if fibs 6 == 13 && fibs 7 == 12 then "yes" else "no"
"no"
```

To reload your .hs file into ghci after an edit:
> :r

If-then-else expressions return a result (like javascript ternary ? :)

Basic logic operators same as C/Java/etc:
==, &&, ||

Haskell 101

`where` lets you place local definitions after expression body:

```haskell
fibonacci n = fibs n 1 1
  where
    fibs 0 a b = a
    fibs n a b = fibs (n-1) b (a+b)
```

`let ... in` allows you to place definitions before expression body:

```haskell
fibonacci :: Int -> Int
fibonacci n =
  let
    fibs 0 a b = a
    fibs n a b = fibs (n-1) b (a+b)
  in
    fibs n 1 1
```

Whitespace rule: Expressions can continue across a line break but must be indented.
Definitions in the same scope must be left-aligned.

Lecture Activity 2

To be announced...

Specifying types of top-level functions is a good idea

Instead of letting them be inferred automatically, we can specify the type explicitly:

```
factorial :: Int -> Int
factorial 1 = 1
factorial n = n * factorial (n-1)
```

Guards are a more flexible alternative to pattern matching

```
factorial :: Int -> Int
factorial n
      | n <= 1 = 1
      | otherwise = n * factorial (n-1)
```

You can put a full expression here

Specifying types of top-level functions is a good idea

What is the type of this function?

```
factorial 1 = 1
factorial n = n * factorial (n-1)
```

We can ask GHCI:

```
$stack ghci test.hs
*Main> :t factorial
factorial :: (Num t, Eq t) => t -> t
```

compiler tries to infer the most generic type possible

We can ask ghci about type classes

```
> :i Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
        -- Defined in `GHC.Num'
instance Num Word -- Defined in `GHC.Num'
instance Num Integer -- Defined in `GHC.Num'
instance Num Int -- Defined in `GHC.Num'
instance Num Float -- Defined in `GHC.Float'
instance Num Double -- Defined in `GHC.Float'
```

⇦ Type classes are like a TypeScript interface, a promise that certain functions are available for types that are 'instances' of the type class.

⇦ These are all of the instances of the Num typeclass

What operations are missing here compared to what you can do with a JavaScript number?

We can ask ghci about type classes

```
> :i Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (/=) #-}
        -- Defined in `GHC.Classes'
instance (Eq a, Eq b) => Eq (Either a b)
  -- Defined in `Data.Either'
instance Eq a => Eq [a] -- Defined in `GHC.Classes'
instance Eq Word -- Defined in `GHC.Classes'
instance Eq Ordering -- Defined in `GHC.Classes'
instance Eq Int -- Defined in `GHC.Classes'
instance Eq Float -- Defined in `GHC.Classes'
instance Eq Double -- Defined in `GHC.Classes'
instance Eq Char -- Defined in `GHC.Classes'
instance Eq Bool -- Defined in `GHC.Classes'

etc...
```

What is still missing here compared to what you can do with a JavaScript number?

We can ask ghci about type classes

```
> :i Int
data Int = GHC.Types.I# GHC.Prim.Int#   -- Defined in `GHC.Types'
instance Eq Int -- Defined in `GHC.Classes'
instance Ord Int -- Defined in `GHC.Classes'
instance Show Int -- Defined in `GHC.Show'
instance Read Int -- Defined in `GHC.Read'
instance Enum Int -- Defined in `GHC.Enum'
instance Num Int -- Defined in `GHC.Num'
instance Real Int -- Defined in `GHC.Real'
instance Integral Int -- Defined in `GHC.Real'
instance Bounded Int -- Defined in `GHC.Enum'
```

More numeric types:

Integer - arbitrarily big ints

Double - 64 bit floats (on x86)

Rational - Integer / Integer

Data

```
data Student = Student Int String Int
```

```
> t = Student 123 "Tim" 45
```

⇦ Student is now a constructor function

```
name (Student _ n _) = n
> name t
"Tim"
```

⇦ Use pattern matching to bind variables inside the data structure

```
best :: [Student] -> Student -> Student
best [] b = b
best (a@(Student _ _ am):rest) b@(Student _ _ bm) =
 if am > bm
 then best rest a
 else best rest b
```

⇧
"as" pattern
Binds b to the whole data structure
then pattern matches whatever's inside the brackets

Record Syntax

```
data Student = Student { id::Integer, name::String, mark::Int }
```

Creates named getter functions for each field

```
> t = Student 123 "Tim" 95
> mark t
95
> name t
"Tim"
> id t
123
```

Algebraic Data Types

```
data ConsList = Null | Cons Int ConsList
l = Cons 1 $ Cons 2 $ Cons 3 Null

consLength :: ConsList -> Int
consLength Null = 0
consLength (Cons _ rest) = 1 + consLength rest
```

⇦ Defined through algebraic operations

A | B and A B

⇧ Or

⇧ And

Haskell Lists - cons lists

Lists use syntax that looks like JavaScript arrays (but they are linked-lists):

```
> [1,2,3]
[1,2,3]
```

Cons operator:

```
> 1:[]
[1]

> 1:2:3:[4,5,6]
[1,2,3,4,5,6]
```

Concat operator:

```
> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
```

Basic list functions

```
> length [1,2,3]
3

> minimum [1,2,3]   -- assuming the type of things in the list is orderable
1

> maximum [1,2,3]   -- ditto
3
```

Quick Sort

A simple version of the quick sort algorithm:

QuickSort list:
  Take head of list as a *pivot*
  Take tail of list as *rest*
  return

cons ⇩ filter ⇩

QuickSort(elements of rest < pivot) ++ (pivot : QuickSort(elements of rest >= pivot))

⇧ filter ⇧ concat

```
forEach(console.log)(sort(fromArray(marks)))
```

```
0
6.73
7.15
7.23
8.16
9.5
10.91
11.56
11.88
14.68
...
```

We could do functional programming in JavaScript

… but it wasn't ideal!

```
const
   sort = order=>
       list=> !list ? null :
           (pivot=>rest=>
               (lesser=>greater=>
                       concat(sort(order)(lesser))(cons(pivot)(sort(order)(greater)))
               )(filter(a=> order(a)(pivot))(rest))(filter(a=> !order(a)(pivot))(rest))
           )(head(list))(tail(list))
```

We could do functional programming in JavaScript

… but it wasn't ideal!

```
const
   sort = order=>
      list=> !list ? null :
         (pivot=>rest=>









         )(head(list))(tail(list))
```

We could do functional programming in JavaScript

… but it wasn't ideal!

```
const
   sort = order=>
      list=> !list ? null :
         (pivot=>rest=>
            (lesser=>greater=>




            )(filter(a=> order(a)(pivot))(rest))
             (filter(a=> !order(a)(pivot))(rest))
         )(head(list))(tail(list))
```

We could do functional programming in JavaScript

… but it wasn't ideal!

```
const
   sort = order=>
       list=> !list ? null :
           (pivot=>rest=>
               (lesser=>greater=>
                       concat (sort(order)(lesser))
                              (cons (pivot)
                                    (sort(order)(greater)))
               )(filter(a=> order(a)(pivot))(rest))
                (filter(a=> !order(a)(pivot))(rest))
           )(head(list))(tail(list))
```

Compare:

JavaScript

```
const
  sort = order=>
    list=> !list ? null :
      (pivot=>rest=>
        (lesser=>greater=>
          concat(sort(order)(lesser))(cons(pivot)(sort(order)(greater)))
        )(filter(a=> order(a)(pivot))(rest))(filter(a=> !order(a)(pivot))(rest))
      )(head(list))(tail(list))
```

Haskell

```
sort [] = []
sort (pivot:rest) = lesser ++ [pivot] ++ greater
 where
   lesser = sort (filter (<pivot) rest)
   greater = sort (filter (>=pivot) rest)
```

Expressive, declarative code

Pattern matching: like destructuring of parameters in TypeScript, but better:

```
sort [] = []
```
← the pattern will be matched to args to determine which version of the function to run

↙️

```
sort (pivot:rest) = lesser ++ [pivot] ++ greater
 where
   lesser = sort $ filter (<pivot) rest
   greater = sort $ filter (>=pivot) rest
```

← Think of code like this as a “pipeline” or “chain” of function application, working right-to-left

↑

$ is an “infix” function with low precedence:

f $ x = f x

Allows us to eliminate some ( )

Type definitions

Type-class
constraint on t

⇩

```
sort :: Ord t => [t] -> [t]
sort [] = []
sort (pivot:rest) = let
   below = partition (<pivot)
   above = partition (>=pivot)
 in
   below rest ++ [pivot] ++ above rest
 where
   partition comparison = sort . filter comparison
```

⇦ Type definition, list with elements of type t

↑

. is compose

Compare definitions of . and $:

```
infixr 9 .
(.) :: (b->c) -> (a->b) -> (a->c)
(f . g) x = f (g x)

infixr 0 $
($) :: (a->b) -> a -> b
f $ x = f x
```

Expressive, declarative code - *preview*

```haskell
qsort :: Ord t => [t] -> [t]
qsort [] = []
qsort (pivot:rest) = rest `below` pivot ++ pivot : rest `above` pivot
where
  below = flip (part .  (>))
  above = flip (part .  (<=))
  part = (qsort .) . filter
```

Point-free gung-fu
- we’ll dig into how we do this in week 8

Conclusions

- There's nothing to be scared of in Haskell
- It's just a more elegant way to express the functional programming concepts we have already covered in JavaScript and TypeScript
- We'll be seeing a lot more of Haskell in coming weeks, the aim is to make you proficient enough to do something interesting in Assignment 2
- In particular, we'll be looking at some interesting type classes that provide powerful abstractions of common types of computation, and elegant ways to combine pure and effectful code (e.g. with IO)