

# CMPS 112: Spring 2019

## Comparative Programming Languages

### *Intro to Haskell*

Owen Arden  
UC Santa Cruz

Based on course materials developed by Nadia Polikarpova

## What is Haskell?

- A **typed**, **lazy**, **purely functional** programming language
  - Haskell =  $\lambda$ -calculus +
    - Better syntax
    - Types
    - Built-in features
      - Booleans, numbers, characters
      - Records (tuples)
      - Lists
      - Recursion
      - ...

2

## Why Haskell?

- Haskell programs tend to be *simple* and *correct*
- **Quicksort in Haskell**

```
sort []      = []
sort (x:xs) = sort ls ++ [x] ++ sort rs
  where
    ls      = [ l | l <- xs, l <= x ]
    rs      = [ r | r <- xs, x <  r ]
```

- **Goals for this week**
  - Understand the above code
  - Understand what **typed**, **lazy**, and **purely functional** means (and why you care)

3

## Haskell vs $\lambda$ -calculus: Programs

- A program is an expression (not a sequence of statements)
- It evaluates to a value (it does not perform actions)
  - $\lambda$ :  
`(\x -> x) apple`  $-- \rightsquigarrow$  *apple*
  - Haskell:  
`(\x -> x) "apple"`  $-- \rightsquigarrow$  *"apple"*

4

## Haskell vs $\lambda$ -calculus: Functions

- Functions are first-class values:
    - can be *passed as arguments* to other functions
    - can be *returned as results* from other functions
    - can be *partially applied* (arguments passed *one at a time*)
- `(\x -> (\y -> x (x y))) (\z -> z + 1) 0`  $-- \rightsquigarrow$  *2*
- **BUT:** unlike  $\lambda$ -calculus, not everything is a function!

5

## Haskell vs $\lambda$ -calculus: top-level bindings

- Like in Elsa, we can name terms to use them later
- Elsa:

```
let T    = \x y -> x
let F    = \x y -> y

let PAIR = \x y -> \b -> ITE b x y
let FST  = \p -> p T
let SND  = \p -> p F

eval fst:
  FST (PAIR apple orange)
  ==> apple
```

6

## Haskell vs $\lambda$ -calculus: top-level bindings

- Like in Elsa, we can name terms to use them later
- **Haskell:**

```
haskellIsAwesome = True
pair = \x y -> \b -> if b then x else y
fst = \p -> p haskellIsAwesome
snd = \p -> p False

-- In GHCi:
> fst (pair "apple" "orange") -- "apple"
```

- The names are called **top-level variables**
- Their definitions are called **top-level bindings**

7

## Syntax: Equations and Patterns

- You can define function bindings using **equations**:

```
pair x y b = if b then x else y -- pair = \x y b -> ...
fst p      = p True             -- fst = \p -> ...
snd p      = p False            -- snd = \p -> ...
```

8

## Syntax: Equations and Patterns

- A single function binding can have *multiple* equations with different **patterns** of parameters:

```
pair x y True  = x -- If 3rd arg matches True,
                  -- use this equation;
pair x y False = y -- Otherwise, if 3rd arg matches
                  -- False, use this equation.
```

- The first equation whose pattern matches the actual arguments is chosen
- For now, a pattern is:
  - a variable (matches any value)
  - or a value (matches only that value)

9

## Syntax: Equations and Patterns

- A single function binding can have *multiple* equations with different **patterns** of parameters:

```
pair x y True  = x  -- If 3rd arg matches True,  
                  -- use this equation;  
pair x y False = y  -- Otherwise, if 3rd arg matches  
                  -- False, use this equation.
```

- Same as:

```
pair x y True  = x  -- If 3rd arg matches True,  
                  -- use this equation;  
pair x y b     = y  -- Otherwise use this equation.
```

10

## Syntax: Equations and Patterns

- A single function binding can have *multiple* equations with different **patterns** of parameters:

```
pair x y True  = x  -- If 3rd arg matches True,  
                  -- use this equation;  
pair x y False = y  -- Otherwise, if 3rd arg matches  
                  -- False, use this equation.
```

- Same as:

```
pair x y True  = x  -- If 3rd arg matches True,  
                  -- use this equation;  
pair x y _     = y  -- Otherwise use this equation.
```

11

## Equations with guards

- An equation can have multiple guards (Boolean expressions):

```
cmpSquare x y | x > y*y = "bigger :)"  
              | x == y*y = "same :|"  
              | x < y*y = "smaller :("
```

- Same as:

```
cmpSquare x y | x > y*y = "bigger :)"  
              | x == y*y = "same :|"  
              | otherwise = "smaller :("
```

12

## Recursion

- Recursion is built-in, so you can write:

```
sum n = if n == 0
      then 0
      else n + sum (n - 1)
```

- Or you can write:

```
sum 0 = 0
sum n = n + sum (n - 1)
```

13

## Scope of variables

- Top-level variables have global scope

```
message = if haskellIsAwesome -- this var defined below
          then "I love CSE 130"
          else "I'm dropping CSE 130"

haskellIsAwesome = True
```

- Or you can write:

```
-- What does f compute?
f 0 = True
f n = g (n - 1) -- mutual recursion!
g 0 = False
g n = f (n - 1) -- mutual recursion!
```

- Answer: f is isEven, g is isOdd

14

## Scope of variables

- Is this allowed?

```
haskellIsAwesome = True
```

```
haskellIsAwesome = False -- changed my mind
```

- Answer: no, a variable can be defined once per scope; no mutation!

15

## Local variables

- You can introduce a *new* (local) scope using a **let**-expression

```
sum 0 = 0
sum n = let n' = n - 1
        in n + sum n'  -- the scope of n'
                       -- is the term after in
```

- Syntactic sugar for nested **let**-expressions:

```
sum 0 = 0
sum n = let
    n'   = n - 1
    sum' = sum n'
in n + sum'
```

16

## Local variables

- If you need a variable whose scope is an equation, use the **where** clause instead:

```
cmpSquare x y | x > z = "bigger :)"
               | x == z = "same :|"
               | x < z = "smaller :("
where z = y*y
```

17

## Types

- What would *Elsa* say?

```
let FNORD = ONE ZERO
```

- Answer:** Nothing. When evaluated, it will crunch to *something*, but it will be nonsensical.
  - $\lambda$ -calculus is **untyped**.

18

## Types

- What would *Python* say?

```
def fnord():  
    return 0(1)
```

- **Answer:** Nothing. When evaluated will cause a run-time error.
  - Python is **dynamically typed**

19

## Types

- What would *Java* say?

```
void fnord() {  
    int zero;  
    zero(1);  
}
```

- **Answer:** Java compiler will reject this.
  - Java is **statically typed**.

20

## Types

- In *Haskell* every expression either **has a type** or is **ill-typed** and rejected statically (at compile-time, before execution starts)
  - like in Java
  - unlike  $\lambda$ -calculus or Python

```
fnord = 1 0    -- rejected by GHC
```

21

## Type Annotations

- You can annotate your bindings with their types using `::`, like so:

```
-- | This is a Boolean:
haskellIsAwesome :: Bool
haskellIsAwesome = True

-- | This is a string
message :: String
message = if haskellIsAwesome
          then "I love CMPS 112"
          else "I'm dropping CMPS 112"
```

22

## Type Annotations

```
-- | This is a word-size integer
rating :: Int
rating = if haskellIsAwesome then 10 else 0

-- | This is an arbitrary precision integer
bigNumber :: Integer
bigNumber = factorial 100
```

- If you omit annotations, GHC will infer them for you
  - Inspect types in GHCi using `:t`
  - You should annotate all top-level bindings anyway! (Why?)

23

## Function Types

- Functions have **arrow types**
  - `\x -> e` has type `A -> B`
  - If `e` has type `B`, assuming `x` has type `A`

- For example:

```
> :t (\x -> if x then 'a' else 'b')
(\x -> if x then 'a' else 'b') :: Bool -> Char
```

24



## Function Types

- You should annotate your function bindings:

```
sum :: Int -> Int
sum 0 = 0
sum n = n + sum (n - 1)
```

- With multiple arguments:

```
pair :: String -> (String -> (Bool -> String))
pair x y b = if b then x else y
```

- Same as:

```
pair :: String -> String -> Bool -> String
pair x y b = if b then x else y
```

25

## Lists

- A list is

- either an *empty list*

```
[] -- pronounced "nil"
```

- or a *head element* attached to a *tail list*

```
x:xs -- pronounced "x cons xs"
```

26

## Terminology: constructors and values

```
[] -- A list with zero elements
```

```
1:[] -- A list with one element: 1
```

```
(:) 1 [] -- Same thing: for any infix op,  
-- (op) is a regular function!
```

```
1:(2:(3:(4:[]))) -- A list with four elements: 1, 2, 3, 4
```

```
1:2:3:4:[] -- Same thing (: is right associative)
```

```
[1,2,3,4] -- Same thing (syntactic sugar)
```

27

## Lists

- `[]` and `(:)` are called the list **constructors**
- We've seen constructors before:
  - `True` and `False` are `Bool` constructors
  - `0`, `1`, `2` are... well, it's complicated, but you can think of them as `Int` constructors
  - these constructions didn't take any parameters, so we just called them *values*
- In general, a **value** is a constructor applied to *other values* (e.g., *list values* on previous slide)

28

## Type of a list

- A list has type `[A]` if each one of its elements has type `A`

- Examples:

```
myList :: [Int]
myList = [1,2,3,4]

myList' :: [Char]           -- or :: String
myList' = ['h', 'e', 'l', 'l', 'o'] -- or = "hello"

myList'' = [1, 'h'] -- Type error: elements have
                    -- different types!

myList''' :: [t] -- Generic: works for any type t!
myList''' = []
```

29

## Functions on lists: range

```
-- | List of integers from n upto m
upto :: Int -> Int -> [Int]
upto n m
  | n > m    = []
  | otherwise = n : (upto (n + 1) m)
```

- There is also syntactic sugar for this!

```
[1..7]    -- [1,2,3,4,5,6,7]
[1,3..7]  -- [1,3,5,7]
```

30

## Functions on lists: length

```
-- | Length of the List
length :: ???
length xs = ???
```

31

## Pattern matching on lists

```
-- | Length of the List
length :: [Int] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

- A pattern is either a *variable* (incl. `_`) or a *value*
- A pattern is
  - either a *variable* (incl. `_`)
  - or a *constructor* applied to other *patterns*
- **Pattern matching** attempts to match *values* against *patterns* and, if desired, *bind* variables to successful matches.

32

## Some useful library functions

```
-- | Is the List empty?
null :: [t] -> Bool

-- | Head of the List
head :: [t] -> t -- careful: partial function!

-- | Tail of the List
tail :: [t] -> [t] -- careful: partial function!

-- | Length of the List
length :: [t] -> Int

-- | Append two Lists
(++) :: [t] -> [t] -> [t]

-- | Are two Lists equal?
(==) :: [t] -> [t] -> Bool
```

You can search for library  
functions (by type!) at  
[hoogle.haskell.org](http://hoogle.haskell.org)

33

## Pairs

```
myPair :: (String, Int) -- pair of String and Int
myPair = ("apple", 3)
```

- `(,)` is the pair constructor

```
-- Field access using library functions:
whichFruit = fst myPair -- "apple"
howMany    = snd myPair -- 3
```

```
-- Field access using pattern matching:
isEmpty (x, y) = y == 0
```

```
-- same as:
isEmpty      = \ (x, y) -> y == 0
```

```
-- same as:
isEmpty p    = let (x, y) = p in y == 0
```

You can use pattern matching not only in equations, but also in  $\lambda$ -bindings and `let`-bindings!

34

## Pattern matching with pairs

- Is this pattern matching correct? What does this function do?

```
f :: String -> [(String, Int)] -> Int
f _ [] = 0
f x ((k,v) : ps)
  | x == k    = v
  | otherwise = f x ps
```

35

## Tuples

- Can we implement triples like in  $\lambda$ -calculus?
- Sure! But Haskell has native support for  $n$ -tuples:

```
myPair :: (String, Int)
myPair = ("apple", 3)
```

```
myTriple :: (Bool, Int, [Int])
myTriple = (True, 1, [1,2,3])
```

```
my4tuple :: (Float, Float, Float, Float)
my4tuple = (pi, sin pi, cos pi, sqrt 2)
```

```
...
-- And also:
myUnit :: ()
myUnit = ()
```

36

## List comprehensions

- A convenient way to construct lists from other lists:

```
[toUpper c | c <- s] -- Convert string s to upper case
```

```
[(i,j) | i <- [1..3],  
         j <- [1..i] ] -- Multiple generators
```

```
[(i,j) | i <- [0..5],  
         j <- [0..5],  
         i + j == 5] -- Guards
```

37

## Quicksort in Haskell

```
sort []      = []  
sort (x:xs) = sort ls ++ [x] ++ sort rs  
  where  
    ls      = [ l | l <- xs, l <= x ]  
    rs      = [ r | r <- xs, x < r ]
```

38

## What is Haskell?

- A typed, lazy, purely functional programming language

39

## Haskell is statically typed

- Every expression either has a type, or is *ill-typed* and rejected at compile time
- **Why is this good?**
  - catches errors early
  - types are contracts (you don't have to handle ill-typed inputs!)
  - enables compiler optimizations

40

## Haskell is purely functional

- **Functional** = functions are *first-class values*
- **Pure** = a program is an expression that evaluates to a value
  - No side effects! unlike in Python, Java, etc:

```
public int f(int x) {  
    calls++;                // side effect!  
    System.out.println("calling f"); // side effect!  
    launchMissile();        // side effect!  
    return x * 2;  
}
```
  - in Haskell, a function of type `Int -> Int` computes a *single integer output* from a *single integer input* and does **nothing else**

41

## Haskell is purely functional

- **Referential transparency:** The same expression always evaluates to the same value
  - More precisely: In a scope where  $x_1, \dots, x_n$  are defined, all occurrences of  $e$  with  $FV(e) = \{x_1, \dots, x_n\}$  have the same value
- **Why is this good?**
  - easier to reason about (remember `x++` vs `++x` in C?)
  - enables compiler optimizations
  - especially great for parallelization ( $e_1 + e_2$ : we can always compute  $e_1$  and  $e_2$  in parallel!)

42

## Haskell is lazy

- An expression is evaluated only when its result is needed
- **Example:** `take 2 [1 .. (factorial 100)]`

```
      take 2 (  upto 1 (factorial 100))
=>      take 2 (  upto 1 933262154439...)
=>      take 2 (1:(upto 2 933262154439...)) -- def upto
=> 1: (take 1 (  upto 2 933262154439...)) -- def take 3
=> 1: (take 1 (2:(upto 3 933262154439...)) -- def upto
=> 1:2:(take 0 (  upto 3 933262154439...)) -- def take 3
=> 1:2:[]                                -- def take 1

-
```

43

## Haskell is lazy

- **Why is this good?**
  - Can implement cool stuff like infinite lists: `[1..]`  
*-- first n pairs of co-primes:*

```
take n [(i,j) | i <- [1..],
                j <- [1..i],
                gcd i j == 1]
```
  - encourages simple, general solutions
  - but has its problems too :(

44

## That's all folks!

45