CSE 230

Programming Languages

Goal: Obviously No Deficiencies

Tony Hoare

Turing Award Lecture 1980

"There are two ways of constructing software.
One way is to make it so simple,
that there are obviously no deficiencies,
The other way is to make it so complicated
that there are no obvious deficiencies. "

Goal: Obviously No Deficiencies

Readable

Reusable

Goal: Obviously No Deficiencies

Modifiable

Goal: Obviously No Deficiencies

Predictable

Goal: Obviously No Deficiencies

*Check*able

Goal: Obviously No Deficiencies

Yes, but how?

Goal: Obviously No Deficiencies

Functional Programming(?)

Functional Programming ?

No Assignment.

No Mutation.

No Loops.

Functional Programming ?

John Carmack

FPS (Doom, Quake,…)

" Brutal purity: you have no choice."

Functional Programming ?

Readable

Reusable

Modifiable

Predictable

Checkable

So, Who Uses FP ?

PL Researchers.

Functional Programming ?

Readable

Reusable

Modifiable

Predictable

Checkable

Parallelizable

So, Who Uses FP ?

Google

MapReduce

So, Who Uses FP ?

Microsoft®

F#

So, Who Uses FP ?

Erlang

So, Who Uses FP ?

Scala

So, Who Uses FP ?

Wall Street

...

So, Who Uses FP ?

CSE 230

CSE 230: Medium of Instruction

Haskell

Why Haskell ?

Bleeding edge PL.

Why Haskell ?

Beautiful.

Why Haskell ?

Blows Your Mind.

Why Haskell ?

Alan Perlis

Epigrams In Programming

"A language that doesn't affect how you think about programming, isn't worth knowing"

Why Haskell ?

I wanted to learn it.

CSE 230 : Outline

Readable
Reusable
1. FP & Abstraction

Modifiable
Predictable
Checkable
2. Types & Analysis

CSE 230 : Personnel

Instructor

Ranjit Jhala (jhala@cs)

CSE 230 : Materials

Web

http://cseweb.ucsd.edu/classes/wi14/cse230-a/

Board

https://piazza.com/class#winter2014/cse230/

Book

Haskell School of Expression (SOE)

CSE 230 : Grading

[10%] Class "Clickers"

[60%] Pair Assignments

[30%] Take-home Final

?

What is Haskell?

What is Haskell?

Programming in Haskell

"Computation by Calculation"

Programming in Haskell

"Substitute Equals by Equals"

Substituting Equals

```
3 * (4 + 5)
```

$\Downarrow$

```
3 * 9
```

$\Downarrow$

```
27
```

That's it!

What is Abstraction?

Pattern Recognition

Pattern Recognition

```
pat x y z  =  x * (y  + z )
```

```
pat 31 42 56 =  31 * (42 + 56)
pat 70 12 95 =  70 * (12 + 95)
pat 90 68 12 =  90 * (68 + 12)
```

Pattern Application: “Fun Call”

```
pat x y z  =  x * (y  + z )
```

```
pat 31 42 56
     ⇩
31 * (42 + 56)
     ⇩
  31 * 98
     ⇩
   3038
```

Programming in Haskell

“Substitute Equals by Equals”

Really, that’s it!

Elements of Haskell

Expressions, Values, Types

Expressions

Values

Types

expression :: **Type**

$\Downarrow$

value :: **Type**

The GHC System

Batch Compiler "ghc"

Compile & Run Large Programs

Interactive Shell "ghci"

Tinker with Small Programs

Interactive Shell: ghci

:load *foo.hs*

:type `expression`

:info `variable`

Basic Types

```
31 * (42 + 56) :: Integer
3 * (4.2 + 5.6) :: Double
            'a' :: Char
           True :: Bool
```

Note: `+` and `*` overloaded …

Function Types

A -> B

Function taking input of **A**, yielding output of **B**

```
pos :: Integer -> Bool
pos x = (x > 0)
```

"Multi-Argument" Function Types

A1 -> A2 -> A3 -> B

Function taking args of **A1**, **A2**, **A3**, giving out **B**

```
pat :: Int -> Int -> Int -> Bool
pat x y z = x * (y + z)
```

Tuples

(A1,…,An)

Bounded Sequence of values of type **A1**,…,**An**

```
      ('a', 5) :: (Char, Int)
 ('a', 5.2, 7) :: (Char, Double, Int)
((7, 5.2), True) ::
```

Extracting Values From Tuples

(A1,A2,…,An)

Pattern Matching extracts values from tuple

```
pat :: Int -> Int -> Int -> Bool
pat x y z = x * (y + z)

pat' :: (Int, Int, Int) -> Int
pat' (x, y, z) = x * (y + z)
```

Lists

[A]

Unbounded Sequence of values of types **A**

```
['a','b','c'] :: |
[1,3,5,7] :: |
[(1,True),(2,False)] :: |
[[1],[2,3],[4,5,6]] ::
```

List's Values Must Have Same Type

[A]

Unbounded Sequence of values of types **A**

```
[1, 2, 'c']
```

What is A ?

List's Values Must Have Same Type

[A]

Unbounded Sequence of values of types **A**

```
[1, 2, 'c']
```

(Mysterious) Type Error!

"Cons"tructing Lists

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

Input: element ("head") and list ("tail")

Output: new list with head followed by tail

```
'a':['b','c'] ⇨ ['a','b','c']
1:[] ⇨ [1]
[]:[] ⇨
```

"Cons"tructing Lists

```
cons2 ::
cons2 x y zs = x:y:zs
```

```
cons2 'a' 'b' ['c'] ⇨['a','b','c']
cons2 1 2 [3,4,5,6] ⇨[1,2,3,4,5,6]
```

Syntactic Sugar

```
[x1,x2,…,xn]
```

Is actually a pretty way of writing

```
x1:x2:…:xn:[]
```

Function Practice : List Generation

```
clone :: a -> Int -> [a]
clone x n = if n==0
            then []
            else x:(clone x (n-1))
```

```
clone 'a' 4 ⇨ ['a','a','a','a']
clone 1.1 3 ⇨ [1.1, 1.1,1.1]
```

Function Practice : List Generation

```
clone :: a -> Int -> [a]
clone x 0 = []
clone x n = x:(clone x (n-1))
```

Define with multiple equations

More Readable

Function Practice : List Generation

```
clone :: a -> Int -> [a]
clone x 0 = []
clone x n = x:(clone x (n-1))
```

```
clone 'a' 3
⇨'a':(clone 'a' 2)
⇨'a':('a':(clone 'a' 1))
⇨'a':('a':('a':(clone 'a' 0)))
⇨'a':('a':('a':([])))
['a','a','a']
```

Function Practice : List Generation

```
clone :: a -> Int -> [a]
clone x 0 = []
clone x n = x:(clone x (n-1))
```

Ugly, Complex Expression

Function Practice : List Generation

```
clone :: a -> Int -> [a]
clone x 0 = []
clone x n = let tl = clone x (n-1)
            in x:tl
```

Define with local variables

More Readable

Function Practice : List Generation

```
clone :: a -> Int -> [a]
clone x 0 = []
clone x n = x:tl
            where tl = clone x (n-1)
```

Define with local variables

More Readable

Function Practice : List Generation

```
range :: Int -> Int -> [Int]

range i j = if i<=j
            then []
            else i:(range (i+1) j)
```

```
range 2 8 ⇨ [2,3,4,5,6,7,8]
```

Function Practice : List Generation

```
range :: Int -> Int -> [Int]

range i j | i<=j  = []
          | True  = i:(range (i+1) j)
```

Define with multiple guards

More Readable

Function Practice : List Access

```
listAdd :: [Integer] -> Integer

listAdd [2,3,4,5,6]  ⇨  20
```

Access elements By Pattern Matching

```
listAdd []     = 0
listAdd (x:xs) = x + listAdd xs
```

Recap

Execution = Substitute Equals

Expressions, Values, Types

Base Vals, Tuples, Lists, Functions

Next: Creating Types

Type Synonyms

Names for Compound Types

```
type XY = (Double, Double)
```

Not a new type, just shorthand

Type Synonyms

Write types to represent:

Circle : x-coord, y-coord, radius

```
type Circle = (Double, Double, Double)
```

Square: x-coord, y-coord, side

```
type Square = (Double, Double, Double)
```

Type Synonyms

Bug Alarm!

Call areaSquare on circle, get back junk

```
type Circle = (Double, Double, Double)
    areaCircle (_,_,r) = pi * r * r

type Square = (Double, Double, Double)
    areaSquare (_,_,d) = d * d
```

Solution: New Data Type

```
data CircleT = Circle (Double,Double,Double)
data SquareT = Square (Double,Double,Double)
```

Creates New Types

`CircleT`

`SquareT`

Solution: New Data Type

```
data CircleT = Circle (Double,Double,Double)
data SquareT = Square (Double,Double,Double)
```

Creates New Constructors

```
Circle :: (Double,Double,Double) -> CircleT
Square :: (Double,Double,Double) -> SquareT
```

Only way to create values of new type

Solution: New Data Type

```
data CircleT = Circle (Double,Double,Double)
data SquareT = Square (Double,Double,Double)
```

Creates New Constructors

```
Circle :: (Double,Double,Double) -> CircleT
Square :: (Double,Double,Double) -> SquareT
```

How to access/deconstruct values?

Deconstructing Data

```
areaSquare :: CircleT -> Double
areaCircle (Circle(_,_,r)) = pi * r * r

areaSquare :: SquareT -> Double
areaSquare (Square(_,_,d)) = d * d
```

How to access/deconstruct values?

Pattern Match…!

Deconstructing Data

```
areaSquare :: CircleT -> Double
areaCircle (Circle(_,_,r)) = pi * r * r

areaSquare :: SquareT -> Double
areaSquare (Square(_,_,d)) = d * d
```

Call `areaSquare` on `CircleT` ?

Different Types: GHC catches bug!

How to build a list with squares & circles?

Restriction: List elements have same type!

How to build a list with squares & circles?

Solution: Create a type to represent both!

Variant (aka Union) Types

Create a type to represent both!

```
data CorS =
    | Circle (Double,Double,Double)
    | Square (Double,Double,Double)

               Circle(1,1,1) :: CorS
               Square(2,3,4) :: CorS
[Circle(1,1,1),Square(2,3,4)] :: [CorS]
```

Variant (aka Union) Types

Access/Deconstruct by Pattern Match

```
data CorS =
    | Circle (Double,Double,Double)
    | Square (Double,Double,Double)

area :: CorS -> Double
area (Circle(_,_,r)) = pi*r*r
area (Square(_,_,d)) = d*d
```

A Richer Shape

```
data Shape =
    | Rectangle (Double, Double)
    | Ellipse   (Double, Double)
    | RtTriangle(Double, Double)
    | Polygon  [(Double, Double)]
```

Lets drop the parens...

A Richer Shape

```
data Shape =
    | Rectangle  Double Double
    | Ellipse    Double Double
    | RtTriangle Double Double
    | Polygon  [(Double, Double)]
```

Lets drop the parens...

A Richer Shape

```
data Shape =
    | Rectangle  Double Double
    | Ellipse    Double Double
    | RtTriangle Double Double
    | Polygon  [(Double, Double)]
```

Why can't we drop last case's parens?

Making Shape Readable

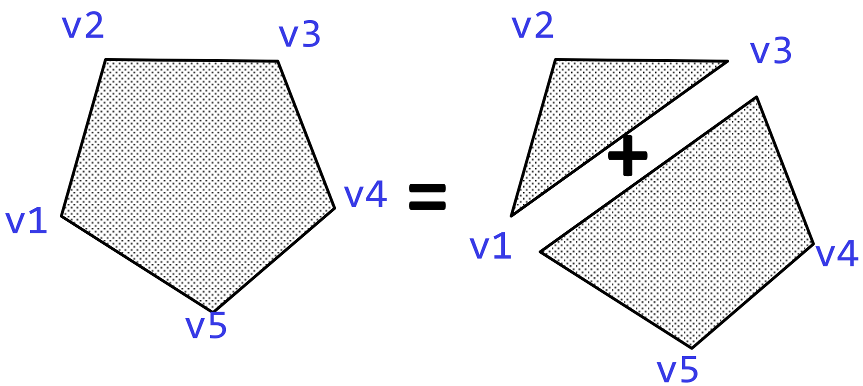
```haskell
data Shape =
    | Rectangle  Side Side
    | Ellipse    Radius Radius
    | RtTriangle Side Side
    | Polygon    [Vertex]

type Side   = Double
type Radius = Double
type Vertex = (Double, Double)
```

Calculating The Area

```haskell
area :: Shape -> Double
area (Rectangle l b)  = l*b
area (RtTriangle b h) = b*h/2
area (Ellipse r1 r2)  = pi*r1*r2
```

GHC warns about missing case!

Calculating Area of Polygon




```haskell
area (Polygon (v1:v2:v3:vs))
  = triArea v1 v2 v3 + area (Polygon (v1:v3:vs))
area (Polygon _)
  = 0
```

“Hello World”

Input/Output in Haskell

Programs Interact With The World

(Don't just compute values!)

Programs Interact With The World

Read files,

Display graphics,

Broadcast packets, …

Programs Interact With The World

How to fit w/ values & calculation ?

I/O via an "Action" Value

Action

Value describing an effect on world

`IO a`

Type of an action that returns an `a`

Example: Output Action

Just do something, return nothing

```
putStr :: String -> IO ()
```

takes input string, returns action

that writes string to stdout

Example: Output Action

Only one way to "execute" action

make it the value of name `main`

```
main :: IO ()
main = putStr "Hello World! \n"
```

Example: Output Action

Compile and Run

```
ghc -o hello helloworld.hs
```

```
main :: IO ()
main = putStr "Hello World! \n"
```

Example: Output Action

"Execute" in ghci

```
:load helloworld.hs
```

```
main :: IO ()
main = putStr "Hello World! \n"
```

Actions Just Describe Effects

Writing does not trigger Execution

```
act2 :: (IO (), IO ())
act2 = (putStr "Hello", putStr "World")
```

Just creates a pair of actions...

```
main :: IO ()
```

How to do many actions?

```
main :: IO ()
```

By composing small actions

Just "do" it

```
do putStr "Hello"
   putStr "World"
   putStr "\n"
```

Single Action

"Sequence" of sub-actions

Just “do” it

```
do act1
   act2
   …
   actn
```

Single Action

“Sequence” of sub-actions

Example: Input Action

Action that returns a value

```
getLine :: IO String
```

Read and Return Line from StdIn

Example: Input Action

Name result via “assignment”

```
x <- act
```

`x` **refers to result in later code**

Example: Input Action

Name result via “assignment”

```
main :: IO ()
main = do putStr “What is your name?”
          n <- getLine
          putStrLn (“Happy New Year ” ++ n)
```

Example: Haskell “Script”

```
#!/usr/bin/env runhaskell

main = do putStr “What is your name?”
          n <- getLine
          putStrLn (“Happy New Year ” ++ n)
```

```
tom-kha:~ jhala$ ./hello.hs
What is your name ?
Ranjit
Happy New Year Ranjit
```

A few syntax “gotchas”

1. Whitespace Sensitive Blocks

```
do act1
   act2
   …
   actn
```

A few syntax “gotchas”

2. Literate Haskell

```
> do act1
>    act2
>    …
>    actn
```

lec-intro.hs **vs** lec-intro.lhs