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

Modifiable

Predictable

Checkable

Yes, but how?
Goal: Obviously No Deficiencies

Functional Programming(?)

No Assignment.

No Mutation.

No Loops.

John McEnroe
Wimbledon, 1980

“You’ve got to be kidding me!”

Readable

Reusable

Modifiable

Predictable

Checkable
So, Who Uses FP?

PL Researchers.

Functional Programming?

Readable
Reusuable
Modifiable
Predictable
Checkable
Parallelizable

Google
MapReduce

Microsoft
F#
So, Who Uses FP?

- Facebook
- Twitter
- Erlang
- Scala
- Wall Street
- CSE 230
Haskell

Why Haskell?

Bleeding edge PL.

Beautiful.

Blows Your Mind.
"A language that doesn't affect how you think about programming, isn’t worth knowing"
CSE 230 : Personnel

Instructor
Ranjit Jhala (jhala@cs)

CSE 230 : Materials

Web
http://cseweb.ucsd.edu/classes/wi13/cse230-a

Board
https://piazza.com/class#winter2013/cse230/

Book
Haskell School of Expression (SOE)

CSE 230 : Grading

[10%] Class Participation

[60%] Pair Assignments

[30%] Take-home Final
What is Haskell?

Programming in Haskell

“Computation by Calculation”

Programming in Haskell

“Substitute Equals by Equals”
Substituting Equals

3 * (4 + 5)

↓

3 * 9

↓

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

↓

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

Pattern Matching extracts values from tuple

\[(A_1, A_2, \ldots, A_n)\]

\[\text{pat} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}\]

\[\text{pat } x \ y \ z = x \ast (y + z)\]

\[\text{pat'} :: (\text{Int}, \text{Int}, \text{Int}) \rightarrow \text{Int}\]

\[\text{pat'} (x, y, z) = x \ast (y + z)\]

Lists

Unbounded Sequence of values of types \(A\)

\[
[‘a’, ‘b’, ‘c’] :: |

\[1,3,5,7] :: |

\[(1,\text{True}),(2,\text{False})\] :: |

\[[1],[2,3],[4,5,6]] :: |

List’s Values Must Have Same Type

\[\text{[A]}\]

\[\text{Unbounded Sequence of values of types } A\]

\[[1, \ 2, \ ‘c’]\]

What is \(A\)?

(Mysterious) Type Error!
### “Cons”tructing Lists

\((:) :: a \rightarrow [a] \rightarrow [a]\)

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

**Output:** new list with head followed by tail

\[ 'a' : [ 'b', 'c' ] \Rightarrow [ 'a', 'b', 'c' ] \]
\[ 1 : [] \Rightarrow [1] \]
\[ [] : [] \Rightarrow \]

\[ \text{cons2} :: \]
\[ \text{cons2} \ x \ y \ zs = x : y : zs \]

\[ \text{cons2} \ 'a' \ 'b' \ ['c'] \Rightarrow [ 'a', 'b', 'c' ] \]
\[ \text{cons2} \ 1 \ 2 \ [3,4,5,6] \Rightarrow [1,2,3,4,5,6] \]

### Syntactic Sugar

\([x_1, x_2, \ldots, x_n]\)

Is actually a pretty way of writing

\[ x_1 : x_2 : \ldots : x_n : [] \]

### Function Practice: List Generation

\[ \text{clone} :: a \rightarrow \text{Int} \rightarrow [a] \]
\[ \text{clone} \ x \ n = \text{if} \ n == 0 \]
\[ \text{then} \ [] \]
\[ \text{else} \ x : (\text{clone} \ x \ (n-1)) \]

\[ \text{clone} \ 'a' \ 4 \Rightarrow [ 'a', 'a', 'a', 'a' ] \]
\[ \text{clone} \ 1.1 \ 3 \Rightarrow [1.1, 1.1, 1.1] \]
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

Clone ‘a’ 3
→ ‘a’:(clone ‘a’ 2)
→ ‘a’:('a':(clone ‘a’ 1))
→ ‘a’:('a':('a':(clone ‘a’ 0)))
→ ['a','a','a']

Clone ‘a’ 0
→ ‘a’:('a':('a':(clone ‘a’ 0)))
→ ['a','a','a']

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']

Clone ‘a’ 0
→ ‘a’:('a':('a':(clone ‘a’ 0)))
→ ['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

Define with local variables

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

Define with local variables

More Readable

range :: Int -> Int -> [Int]
range i j = if i<=j
  then []
  else i:(range (i+1) j)

range 2 8 \rightarrow [2,3,4,5,6,7,8]

Define with multiple guards

More Readable

range :: Int -> Int -> [Int]
range i j | i<=j  = []
  | True   = i:(range (i+1) j)

listAdd :: [Integer] -> Integer
listAdd [2,3,4,5,6] \rightarrow 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

Write types to represent:

type Circle = (Double, Double, Double)

Circle: x-coord, y-coord, radius

Square: x-coord, y-coord, side

type Square = (Double, Double, Double)
**Type Synonyms**

**Bug Alarm!**

Call areaSquare on circle, get back junk

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

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

**Solution: New Data Type**

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

**Creates New Types**

CircleT
SquareT

**Solution: New Data Type**

```haskell
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**

```haskell
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

How to access/deconstruct values?
Pattern Match...!

Call areaSquare on CircleT?
Different Types: GHC catches bug!

How to build a list with squares & circles?
Restriction: List elements have same type!

Solution: Create a type to represent both!

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

areaSquare :: SquareT -> Double
areaSquare (Square(_,_,d)) = d * d
Variant (aka Union) Types

Create a type to represent both!

```haskell
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]
```

Access/Deconstruct by Pattern Match

```haskell
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

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

Lets drop the parens...
```

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

Lets drop the parens...
```
**A Richer Shape**

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

Why can’t we drop last case’s parens?

```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
```

**Calculating Area of Polygon**

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

GHC warns about missing case!
“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 = putStrLn "Hello World! \n"
```

Example: Output Action

Compile and Run
```
ghc -o hello helloworld.hs
```

```
main :: IO ()
main = putStrLn "Hello World! \n"
```
Example: Output Action

“Execute” in ghci

:load helloworld.hs

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

Actions Just Describe Effects

Writing does not trigger Execution

act2 :: (IO (), IO ())
act2 = (putStrLn "Hello", putStrLn "World")

Just creates a pair of actions...

main :: IO ()

How to do many actions?

By composing small actions
**Example: Input Action**

```
do act1
  act2
  ...
  actn
```

*Action that returns a value*

```
getLine :: IO String
```

*Read and Return Line from StdIn*

```
Block Begin/End via Indentation
  "Offside Rule" (Ch3. RWH)
```
Name result via “assignment”

\[ x \leftarrow \text{act} \]

\( x \) refers to result in later code

Example: Haskell “Script”

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

main = do putStrLn "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