“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?
<table>
<thead>
<tr>
<th>Goal: Obviously No Deficiencies</th>
<th>Functional Programming?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functional Programming(?)</strong></td>
<td><strong>No Assignment.</strong></td>
</tr>
<tr>
<td></td>
<td><strong>No Mutation.</strong></td>
</tr>
<tr>
<td></td>
<td><strong>No Loops.</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functional Programming?</th>
<th>Functional Programming?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Readable</td>
<td></td>
</tr>
<tr>
<td>Reusable</td>
<td></td>
</tr>
<tr>
<td>Modifiable</td>
<td></td>
</tr>
<tr>
<td>Predictable</td>
<td></td>
</tr>
<tr>
<td>Checkable</td>
<td></td>
</tr>
</tbody>
</table>

- John Carmack
- FPS (Doom, Quake, ...)

“Brutal purity: you have no choice.”
<table>
<thead>
<tr>
<th>So, Who Uses FP?</th>
<th>Functional Programming?</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL Researchers.</td>
<td>Readable</td>
</tr>
<tr>
<td></td>
<td>Reusable</td>
</tr>
<tr>
<td></td>
<td>Modifiable</td>
</tr>
<tr>
<td></td>
<td>Predictable</td>
</tr>
<tr>
<td></td>
<td>Checkable</td>
</tr>
<tr>
<td></td>
<td>Parallelizable</td>
</tr>
</tbody>
</table>

- Google
- MapReduce
- Microsoft
- F#
<table>
<thead>
<tr>
<th>So, Who Uses FP?</th>
<th>So, Who Uses FP?</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="facebook.png" alt="Facebook Logo" /> Facebook</td>
<td><img src="twitter.png" alt="Twitter Logo" /> Twitter</td>
</tr>
<tr>
<td><strong>Erlang</strong></td>
<td><strong>Scala</strong></td>
</tr>
<tr>
<td><img src="wall_street.png" alt="Wall Street Logo" /> Wall Street</td>
<td><strong>CSE 230</strong></td>
</tr>
<tr>
<td>CSE 230: Medium of Instruction</td>
<td>Why Haskell?</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>![Haskell Logo]</td>
<td>Bleeding edge PL.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Why Haskell?</th>
<th>Why Haskell?</th>
</tr>
</thead>
</table>
Why Haskell?

I wanted to learn it.

Alan Perlis
Epigrams In Programming

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

CSE 230: Outline

1. FP & Abstraction
   - Readable
   - Reusable
   - Modifiable
   - Predictable
   - Checkable

2. Types & Analysis

CSE 230: Personnel

Instructor
Ranjit Jhala (jhala@cs)
<table>
<thead>
<tr>
<th>CSE 230: Materials</th>
<th>CSE 230: Grading</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Web</strong></td>
<td>[10%] Class “Clickers”</td>
</tr>
<tr>
<td><a href="http://cseweb.ucsd.edu/classes/wi14/cse230-a/">http://cseweb.ucsd.edu/classes/wi14/cse230-a/</a></td>
<td>[60%] Pair Assignments</td>
</tr>
<tr>
<td><strong>Board</strong></td>
<td>[30%] Take-home Final</td>
</tr>
<tr>
<td><a href="https://piazza.com/class#winter2014/cse230/">https://piazza.com/class#winter2014/cse230/</a></td>
<td></td>
</tr>
<tr>
<td><strong>Book</strong></td>
<td></td>
</tr>
<tr>
<td>Haskell School of Expression (SOE)</td>
<td></td>
</tr>
</tbody>
</table>

What is Haskell?
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 Application: “Fun Call”

Programming in Haskell

“Substitute Equals by Equals”

Really, that’s it!
<table>
<thead>
<tr>
<th>Elements of Haskell</th>
<th>Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expressions, Values, Types</td>
<td></td>
</tr>
<tr>
<td>Values</td>
<td>Types</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>A</th>
<th>→</th>
<th>B</th>
</tr>
</thead>
</table>

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

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

### “Multi-Argument” Function Types

<table>
<thead>
<tr>
<th>A1</th>
<th>→</th>
<th>A2</th>
<th>→</th>
<th>A3</th>
<th>→</th>
<th>B</th>
</tr>
</thead>
</table>

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

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

### Tuples

**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

```
pat :: Int -> Int -> Int -> Bool
pat x y z = x * (y + z)
pat' :: (Int, Int, Int) -> Int
pat' (x, y, z) = x * (y + z)
```
<table>
<thead>
<tr>
<th>Lists</th>
<th>List’s Values Must Have Same Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Lists" /></td>
<td><img src="image" alt="List’s Values Must Have Same Type" /></td>
</tr>
</tbody>
</table>

**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**

**“Cons”tructing Lists**

- ![Lists](image)

**Unbounded Sequence** of values of types $A$

- [1, 2, ‘c’]

**What is $A$?**

**(Mysterious) Type Error!**

<table>
<thead>
<tr>
<th>Input: element (“head”) and list (“tail”)</th>
<th>Output: new list with head followed by tail</th>
</tr>
</thead>
<tbody>
<tr>
<td>$‘a’$:[‘b’, ‘c’] $\Rightarrow$[‘a’, ‘b’, ‘c’]</td>
<td>$1$:[[]] $\Rightarrow$[1]</td>
</tr>
<tr>
<td>$[]:[[]]$ $\Rightarrow$</td>
<td></td>
</tr>
</tbody>
</table>
“Cons”tructing Lists

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

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

Syntactic Sugar

```haskell
[x1, x2, ..., xn]

Is actually a pretty way of writing

```haskell
x1 : x2 : ... : xn : []
```

Function Practice: List Generation

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

clone 'a' 4  \Rightarrow ['a', 'a', 'a', 'a']
clone 1.1 3  \Rightarrow [1.1, 1.1, 1.1]
```

Define with multiple equations

More Readable
<table>
<thead>
<tr>
<th>Function Practice : List Generation</th>
<th>Function Practice : List Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>clone : a -&gt; Int -&gt; [a]</td>
<td>clone : a -&gt; Int -&gt; [a]</td>
</tr>
<tr>
<td>clone x 0 = []</td>
<td>clone x 0 = []</td>
</tr>
<tr>
<td>clone x n = x:(clone x (n-1))</td>
<td>clone x n = x:(clone x (n-1))</td>
</tr>
<tr>
<td>clone ‘a’ 3</td>
<td>clone ‘a’ 3</td>
</tr>
<tr>
<td>‘a’:(clone ‘a’ 2)</td>
<td>‘a’:(clone ‘a’ 2)</td>
</tr>
<tr>
<td>‘a’:(‘a’:(clone ‘a’ 1))</td>
<td>‘a’:(‘a’:(clone ‘a’ 1))</td>
</tr>
<tr>
<td>‘a’:(‘a’:(‘a’:(clone ‘a’ 0)))</td>
<td>‘a’:(‘a’:(‘a’:(clone ‘a’ 0)))</td>
</tr>
<tr>
<td>[‘a’;‘a’;(‘a’;([],[]))]</td>
<td>[‘a’;‘a’;(‘a’;([],[]))]</td>
</tr>
</tbody>
</table>

Ugly, Complex Expression

<table>
<thead>
<tr>
<th>Function Practice : List Generation</th>
<th>Function Practice : List Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>clone : a -&gt; Int -&gt; [a]</td>
<td>clone : a -&gt; Int -&gt; [a]</td>
</tr>
<tr>
<td>clone x 0 = []</td>
<td>clone x 0 = []</td>
</tr>
<tr>
<td>clone x n = let tl = clone x (n-1)</td>
<td></td>
</tr>
<tr>
<td>in x:tl</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Define with local variables</td>
<td></td>
</tr>
<tr>
<td>More Readable</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function Practice : List Generation</th>
<th>Function Practice : List Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>clone : a -&gt; Int -&gt; [a]</td>
<td>clone : a -&gt; Int -&gt; [a]</td>
</tr>
<tr>
<td>clone x 0 = []</td>
<td>clone x 0 = []</td>
</tr>
<tr>
<td>clone x n = x:tl</td>
<td></td>
</tr>
<tr>
<td>where tl = clone x (n-1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Define with local variables</td>
<td></td>
</tr>
<tr>
<td>More Readable</td>
<td></td>
</tr>
</tbody>
</table>
**Function Practice : List Generation**

```
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

**Recap**

```
Execution = Substitute Equals
Expressions, Values, Types
Base Vals, Tuples, Lists, Functions
```

**Function Practice : List Access**

```
listAdd :: [Integer] -> Integer
listAdd [2,3,4,5,6] \rightarrow 20
```

**Access elements By Pattern Matching**

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

Type Synonyms

Names for Compound Types

*type* $XY = (\text{Double, Double})$

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)

Bug Alarm!

Call areaSquare on circle, get back junk

*type* Circle = (Double, Double, Double, Double)

areaCircle (_,_,r) = pi * r * r

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

areaSquare (_,_,d) = d * d
<table>
<thead>
<tr>
<th>Solution: New Data Type</th>
<th>Solution: New Data Type</th>
</tr>
</thead>
</table>
| **data** CircleT = Circle (Double,Double,Double)  
**data** SquareT = Square (Double,Double,Double) | **data** CircleT = Circle (Double,Double,Double)  
**data** SquareT = Square (Double,Double,Double) |
| Creates New Types  
CircleT  
SquareT | Creates New Constructors  
Circle :: (Double,Double,Double) -> CircleT  
Square :: (Double,Double,Double) -> SquareT |
| Only way to create values of new type |

<table>
<thead>
<tr>
<th>Solution: New Data Type</th>
<th>Deconstructing Data</th>
</tr>
</thead>
</table>
| **data** CircleT = Circle (Double,Double,Double)  
**data** SquareT = Square (Double,Double,Double) | **areaSquare** :: CircleT -> Double  
**areaCircle** (Circle(_,_,r)) = pi * r * r  
**areaSquare** :: SquareT -> Double  
**areaSquare** (Square(_,_,d)) = d * d |
| Only way to create values of new type |
| How to access/deconstruct values? | 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`
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!

Solution: 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

```hs
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**

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

Lets drop the parens...
```

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

```
"Hello World"
```
<table>
<thead>
<tr>
<th>Programs Interact With The World</th>
<th>Programs Interact With The World</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Don’t just compute values!)</td>
<td>Read files,</td>
</tr>
<tr>
<td></td>
<td>Display graphics,</td>
</tr>
<tr>
<td></td>
<td>Broadcast packets, ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Programs Interact With The World</th>
<th>I/O via an “Action” Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>How to fit w/ values &amp; calculation?</td>
<td>Action Value describing an effect on world</td>
</tr>
<tr>
<td></td>
<td>IO a</td>
</tr>
<tr>
<td></td>
<td>Type of an action that returns an a</td>
</tr>
</tbody>
</table>
Example: Output Action

**Just do something, return nothing**

```haskell
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`

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

Example: Output Action

**Compile and Run**

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

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

Example: Output Action

**“Execute” in ghci**

```bash
:load helloworld.hs
```

```haskell
main :: IO ()
main = putStr "Hello World! \n"
```
<table>
<thead>
<tr>
<th>Actions</th>
<th>Just Describe Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Writing does not trigger Execution</strong></td>
<td></td>
</tr>
<tr>
<td>act2 :: (IO (), IO ())</td>
<td></td>
</tr>
<tr>
<td>act2 = (putStr “Hello”, putStr “World”)</td>
<td></td>
</tr>
<tr>
<td>Just creates a pair of actions...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>main :: IO ()</strong></th>
<th>Just “do” it</th>
</tr>
</thead>
<tbody>
<tr>
<td>By composing small actions</td>
<td>do putStr “Hello”</td>
</tr>
<tr>
<td></td>
<td>putStr “World”</td>
</tr>
<tr>
<td></td>
<td>putStr “\n”</td>
</tr>
<tr>
<td></td>
<td><strong>Single Action</strong></td>
</tr>
<tr>
<td></td>
<td>“Sequence” of sub-actions</td>
</tr>
<tr>
<td>Just “do” it</td>
<td>Example: Input Action</td>
</tr>
<tr>
<td>--------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>\texttt{do act1}</td>
<td>Action that returns a value</td>
</tr>
<tr>
<td>\texttt{act2}</td>
<td>\texttt{getLine :: IO String}</td>
</tr>
<tr>
<td>…</td>
<td>Read and Return Line from StdIn</td>
</tr>
<tr>
<td>\texttt{actn}</td>
<td></td>
</tr>
<tr>
<td>Single Action</td>
<td></td>
</tr>
<tr>
<td>“Sequence” of sub-actions</td>
<td></td>
</tr>
<tr>
<td>Example: Input Action</td>
<td>Example: Input Action</td>
</tr>
<tr>
<td>Name result via “assignment”</td>
<td>Name result via “assignment”</td>
</tr>
<tr>
<td>\texttt{x &lt;- act}</td>
<td>\texttt{main :: IO ()}</td>
</tr>
<tr>
<td>\texttt{x} refers to result in later code</td>
<td>\texttt{main = do putStrLn “What is your name?”}</td>
</tr>
<tr>
<td></td>
<td>\texttt{n &lt;- getLine}</td>
</tr>
<tr>
<td></td>
<td>\texttt{putStrLn (“Happy New Year ” ++ n)}</td>
</tr>
</tbody>
</table>
Example: Haskell “Script”

```haskell
#!/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

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

2. Literate Haskell

```hs
> do act1
>    act2
>    ...
>    actn
```

lec-intro.hs vs lec-intro.lhs