What is Haskell?
Programming in Haskell

“Computation by Calculation”
Programming in Haskell

“Substitute Equals by Equals”
Substituting Equals

\[ 3 \times (4 + 5) \]

\[ 3 \times 9 \]

\[ 27 \]

That’s it!
What is Abstraction?

Pattern Recognition
Pattern Recognition

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

\[ \text{pat } 31 \ 42 \ 56 \ = \ 31 \ * \ (42 \ + \ 56) \]

\[ \text{pat } 70 \ 12 \ 95 \ = \ 70 \ * \ (12 \ + \ 95) \]

\[ \text{pat } 90 \ 68 \ 12 \ = \ 90 \ * \ (68 \ + \ 12) \]
Pattern Application: “Fun Call”

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

\[ \text{pat } 31 \ 42 \ 56 \]
\[-\rightarrow\]
\[31 \ast (42 \ + \ 56)\]
\[-\rightarrow\]
\[31 \ast 98\]
\[-\rightarrow\]
\[3038\]
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

\[ A \rightarrow B \]

Function taking input of \( A \), yielding output of \( B \)

\[
\text{pos} :: \text{Integer} \rightarrow \text{Bool} \\
\text{pos} \ x = (x > 0)
\]
“Multi-Argument” Function Types

\( A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow B \)

Function taking args of \( A_1, A_2, A_3 \), giving out \( B \)

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

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

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

Bounded Sequence of values of type \(A_1, \ldots, A_n\)

\[
(\text{‘a’, 5}) :: (\text{Char, Int})
\]

\[
(\text{‘a’, 5.2, 7}) :: (\text{Char, Double, Int})
\]

\[
((7, 5.2), \text{True}) ::
\]
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

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

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

**Input:** element (“head”) and list (“tail”)

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

\[
\begin{align*}
\text{\textquoteleft a\textquoteright : [\textquoteleft b\textquoteright, \textquoteleft c\textquoteright] } & \Leftrightarrow [\text{\textquoteleft a\textquoteright, \textquoteleft b\textquoteright, \textquoteleft c\textquoteright]} \\
1 : [] & \Leftrightarrow [1] \\
[] : [] & \Rightarrow
\end{align*}
\]
“Cons”tructing Lists

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

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

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

close 'a' 4 ⇒ ['a','a','a','a']
close 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

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

\[
\text{clone} \; \text{‘a’} \; 3
\]
\[\rightarrow \text{‘a’} : (\text{clone} \; \text{‘a’} \; 2)\]
\[\rightarrow \text{‘a’} : (\text{‘a’} : (\text{clone} \; \text{‘a’} \; 1))\]
\[\rightarrow \text{‘a’} : (\text{‘a’} : (\text{‘a’} : (\text{clone} \; \text{‘a’} \; 0)))\]
\[\rightarrow [\text{‘a’} : (\text{‘a’} : (\text{‘a’} : (\text{[]})))]
\]
Function Practice: List Generation

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

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

\[
\text{listAdd} :: [\text{Integer}] \rightarrow \text{Integer}
\]

\[
\text{listAdd} \ [2,3,4,5,6] \Rightarrow 20
\]

Access elements By Pattern Matching

\[
\text{listAdd} \ [] = 0
\]

\[
\text{listAdd} \ (x:xs) = x + \text{listAdd} \ xs
\]
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

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

**Square**: x-coord, y-coord, side

```haskell
type Square = (Double, Double, Double)
```
Type Synonyms

Bug Alarm!
Call areaSquare on circle, get back junk

def type Circle = (Double, Double, Double)
    areaCircle (__, __, r) = pi * r * r

def type Square = (Double, Double, Double)
    areaSquare (__, __, d) = d * d
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

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

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

\[ \text{area (Polygon (v1:v2:v3:vs))} = \text{triArea v1 v2 v3} + \text{area (Polygon (v1:v3:vs))} \]

\[ \text{area (Polygon _)} = 0 \]
“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 = 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 = (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
Just “do” it

\texttt{do\ act1}
\texttt{act2}
\texttt{...}
\texttt{actn}

Block Begin/End via Indentation

“Offside Rule” (Ch3. RWH)
Example: Input Action

Action that returns a value

getLine :: IO String

Read and Return Line from StdIn
Example: Input Action

Name result via “assignment”

\[ x <- \text{act} \]

\(x\) refers to result in later code
Name result via “assignment”

```haskell
main :: IO ()
main = do
    putStrLn "What is your name?"
    n <- getLine
    putStrLnL ("Happy New Year " ++ n)
```