What is Haskell?
Programming in Haskell

“Computation by Calculation”
Programming in Haskell

“Substitute Equals by Equals”
Substituting Equals

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

\[ \downarrow \]

\[ 3 \times 9 \]

\[ \downarrow \]

\[ 27 \]

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”

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

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

\[ A \rightarrow B \]

*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

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

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

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

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

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

\[
\begin{align*}
('a', 5) &:: (\text{Char}, \text{Int}) \\
('a', 5.2, 7) &:: (\text{Char}, \text{Double}, \text{Int}) \\
((7, 5.2), \text{True}) &:: \\
\end{align*}
\]
Extracting Values From Tuples

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

Pattern Matching extracts values from tuple

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

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

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

\[
pat' (x, y, z) = x * (y + z)
\]
Lists

Unbounded Sequence of values of types A

[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

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

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

\[
\text{cons2 \ ‘a’ \ ‘b’ \ [‘c’]} \Rightarrow [‘a’, ‘b’, ‘c’]
\]
\[
\text{cons2 1 2 [3,4,5,6]} \Rightarrow [1,2,3,4,5,6]
\]
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’, ‘a’]
close 1.1 3  ⇒  [1.1, 1.1, 1.1, 1.1]
Function Practice : List Generation

clonex :: a -> Int -> [a]
clonex x 0 = []
clonex x n = x:(clonex 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))))
⇒ [áà:(‘a’;(áà]::([])))}
Function Practice : List Generation

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

Ugly, Complex Expression
Definition:

```haskell
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

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

\[
\text{range} :: \text{Int} \to \text{Int} \to [\text{Int}]
\]

\[
\text{range} \; i \; j = \begin{cases} 
\text{[]} & \text{if } j < i \\
\text{i:} (\text{range} \; (i+1) \; j) & \text{else}
\end{cases}
\]

\[
\text{range} \; 2 \; 8 \Leftrightarrow [2,3,4,5,6,7,8]
\]
Function Practice : List Generation

\[
\text{range :: Int} \to \text{Int} \to \text{[Int]} \\
\text{range } i \ j \ | \ j < i \ = \ [\] \\
| \ True = i:(\text{range} \ (i+1) \ j)
\]

Define with multiple guards
More Readable
Function Practice: List Access

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

Access elements by Pattern Matching

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

Execution = Substitute Equals

Expressions, Values, Types

Base Vals, Tuples, Lists, Functions