

Haskell 101

nicuveo@

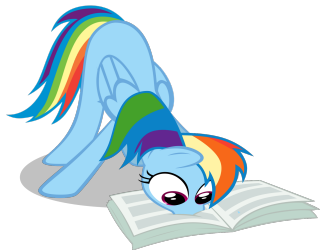
April 9, 2019

- ▶ 101
 - ▶ Concepts and generalities
 - ▶ Syntax overview
 - ▶ Data structures
 - ▶ Declaring functions



- ▶ **Project environment**
 - ▶ Cabal? Cabal hell?
 - ▶ Stackage? Stack?
 - ▶ Haskell at Google?
- ▶ **Advanced stuff**
 - ▶ Functors? Monads?
 - ▶ Monad Transformers?





- ▶ Programming knowledge
- ▶ FP knowledge is a plus
- ▶ `apt-get install haskell-platform`

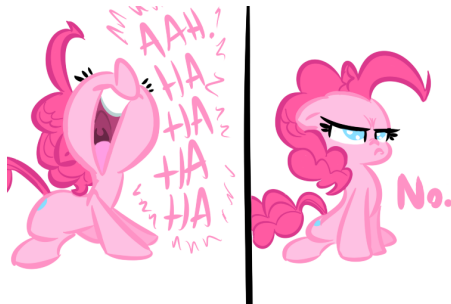


- ▶ Type expressions, get result.
- ▶ Test and debug your code.
- ▶ `:t`

- ▶ Strongly statically typed
- ▶ Purely functional
- ▶ Lazily evaluated
- ▶ General purpose



- ▶ A silver bullet
- ▶ For category theorists

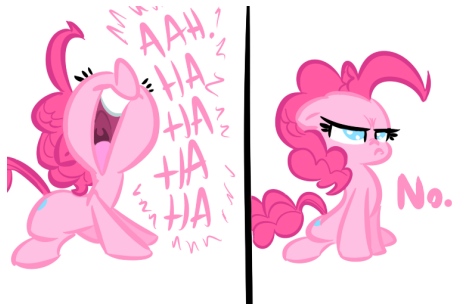


- ▶ A silver bullet
- ▶ For category theorists
- ▶ Hard!



Haskell is NOT...

- ▶ A silver bullet
- ▶ For category theorists
- ▶ Hard! Just different...



► Everything is a function

$$f :: \text{Int} \rightarrow \text{Int}$$
$$f\ x = x + 1$$
$$f : \mathbb{Z} \rightarrow \mathbb{Z}$$
$$f(x) = x + 1$$

- ▶ Everything is a function
- ▶ Everything is immutable

```
let a = 3 in  
  a := a + 1 -- compile error
```

- ▶ Everything is a function
- ▶ Everything is immutable
- ▶ Everything is an expression

```
let a = if someBool then 1 else 0 in  
      a + 1
```

- ▶ Everything is a function
- ▶ Everything is immutable
- ▶ Everything is an expression

```
let a = if someBool then 1 else 0 in  
      a + (let b = 2 in b)
```

- ▶ Everything is a function
- ▶ Everything is immutable
- ▶ Everything is an expression

```
let offset = case colour of
    Red    → 0
    Green  → 8
    Blue   → 16
in baseValue + offset
```

- ▶ Everything is a function
- ▶ Everything is immutable
- ▶ Everything is an expression
- ▶ No side effects!

```
foo :: Int → String
```

- ▶ Everything is a function
- ▶ Everything is immutable
- ▶ Everything is an expression
- ▶ No side effects unless explicitly stated

```
readFile :: String → IO String
```


- ▶ All side effects are in IO

- ▶ All side effects are in IO
- ▶ Functions \notin IO are deemed pure



- ▶ All side effects are in IO
- ▶ Functions \notin IO are deemed pure
- ▶ Functions \in IO are deemed impure



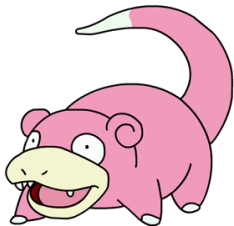
▶ $\exists f :: a \rightarrow \text{IO } a$ (from pure to impure)

▶ $\nexists f :: \text{IO } a \rightarrow a$ (from impure to pure)

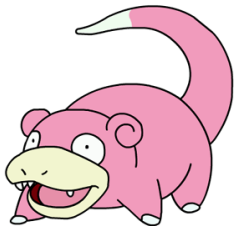
▶ $\exists f :: a \rightarrow \text{IO } a$ (from pure to impure)

▶ $\nexists f :: \text{IO } a \rightarrow a$ (from impure to pure)

IO corrupts.



- ▶ Deferred expression evaluation
- ▶ Not used \Rightarrow not computed



- ▶ Deferred expression evaluation
- ▶ Not used \Rightarrow not computed

```
if (obj != NULL && obj→value > 0)
```

► Strict evaluation: inner to outer

```
add( 12 + 8 , 20 + 2 )  
add(  20   ,  22   )  
    20   +   22  
      42
```


- ▶ Strict evaluation: inner to outer
- ▶ Lazy evaluation: outer to inner

$$\begin{array}{c} \text{add}(\quad 12 + 8 \quad , \quad 20 + 2 \quad) \\ \quad 12 + 8 \quad + \quad 20 + 2 \\ \quad \quad \quad 42 \end{array}$$

- Memory pitfalls

Delayed computations (but escape hatches)

- Memory pitfalls
- IO and parallelism pitfalls

Delayed computations (but escape hatches)

- Memory pitfalls
- IO and parallelism pitfalls
- + Huge optimizations

Equation reduction and short-circuiting

- Memory pitfalls
- IO and parallelism pitfalls
- + Huge optimizations
- + Greater expressivity (e.g. infinite structures)

```
> let naturalNumbers = [0,1..]  
> let squaredNumbers = map (^2) naturalNumbers  
> take 5 squaredNumbers  
[0,1,4,9,16]
```

`f :: Int → Int → [Int]`

`f` $::$ `Int` \rightarrow (`Int` \rightarrow [`Int`])

`f` $::$ `Int` \rightarrow `Int` \rightarrow [`Int`]

`f` `::` `Int` `→` `(Int` `→` `[Int]` `)`

`f` `::` `Int` `→` `Int` `→` `[Int]`

`f 1` `::` `Int` `→` `[Int]`

Curried functions, partial application

`f` `::` `Int` `→` `(Int` `→` `[Int]` `)`

`f` `::` `Int` `→` `Int` `→` `[Int]`

`f 1` `::` `Int` `→` `[Int]`

`(f 1) 2` `::` `[Int]`

`f` $::$ `Int` \rightarrow (`Int` \rightarrow [`Int`])

`f` $::$ `Int` \rightarrow `Int` \rightarrow [`Int`]

`f 1` $::$ `Int` \rightarrow [`Int`]

`f 1 2` $::$ [`Int`]

`(f 1) 2` $::$ [`Int`]

??? :: (a → b) → [a] → [b]

??? :: (a → b) → [a] → [b]

Lowercase letter: type parameter

??? :: (a → b) → [a] → [b]

(a → b) function from type A to type B

[a] list of values of type A

[b] list of values of type B

`map` :: $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$

$(a \rightarrow b)$ function from type A to type B

$[a]$ list of values of type A

$[b]$ list of values of type B

`map` :: (a → b) → [a] → [b]

?????? :: (a → Bool) → [a] → [a]

`map` :: (a → b) → [a] → [b]

`filter` :: (a → Bool) → [a] → [a]

`map` :: $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$

`filter` :: $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$

`($)` :: $(a \rightarrow b) \rightarrow a \rightarrow b$

```
map    :: (a → b) → [a] → [b]
filter :: (a → Bool) → [a] → [a]
($)    :: (a → b) → a → b
```

```
let a = fun (x + y)
```

```
map    :: (a → b) → [a] → [b]
filter :: (a → Bool) → [a] → [a]
($)    :: (a → b) → a → b
```

```
let a = fun $ x + y
```

```
map    :: (a → b) → [a] → [b]
filter :: (a → Bool) → [a] → [a]
($)    :: (a → b) → a → b
(.)     :: (b → c) → (a → b) → (a → c)
```

```
map    :: (a → b) → [a] → [b]
filter :: (a → Bool) → [a] → [a]
($)    :: (a → b) → a → b
(.)    :: (b → c) → (a → b) → (a → c)
```

$$(f \circ g)(x) = f(g(x))$$

`map` :: $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$

`filter` :: $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$

$(\$)$:: $(a \rightarrow b) \rightarrow a \rightarrow b$

$(.)$:: $(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$

`show` :: `Stuff` \rightarrow `String`

`length` :: `String` \rightarrow `Int`

`length` . `show` :: `Stuff` \rightarrow `Int`

```
map    :: (a → b) → [a] → [b]
filter :: (a → Bool) → [a] → [a]
($)    :: (a → b) → a → b
(.)    :: (b → c) → (a → b) → (a → c)
```

```
cat input | grep token | sed stuff | tee output
```

Quizz with a vengeance!

`foldl` :: $(a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a$

Quizz with a vengeance!

`foldl` :: $(a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a$

$(a \rightarrow b \rightarrow a)$ combines accumulator and value

a initial accumulator

$[b]$ list of values

a result

Quizz with a vengeance!

`foldl` :: $(a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a$

$(a \rightarrow b \rightarrow a)$ combines accumulator and value

a initial accumulator

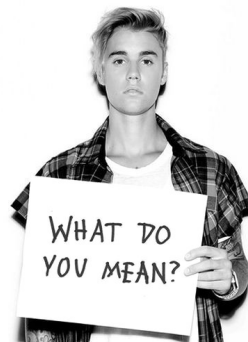
$[b]$ list of values

a result

"reduce"

- ▶ Type composition
- ▶ Product and sum types
- ▶ Cardinality expressions

- ▶ Type composition
- ▶ Product and sum types
- ▶ Cardinality expressions



```
type Point    = (Int, Int)  -- tuple
```

```
type Point    = (Int, Int)  -- tuple  
type Polygon = [Point]      -- list
```

```
type Point    = (Int, Int)  -- tuple  
  
type Polygon = [Point]      -- list  
  
type Map k v = [(k, v)]     -- type parameters
```



▶ No methods...



- ▶ No methods...
- ▶ No modifiers...



- ▶ No methods...
- ▶ No modifiers...
- ▶ No private members...



- ▶ No methods...
- ▶ No modifiers...
- ▶ No private members...

What's left?



- ▶ No methods...
- ▶ No modifiers...
- ▶ No private members...

What's left? Constructors!



The almighty "data" keyword

```
data None = None
```

```
None :: None
```

The almighty "data" keyword

```
data None      = None
```

```
data Minutes = Minutes Int
```

```
Minutes      :: Int → Minutes  
Minutes 42 ::      Minutes
```

The almighty "data" keyword

```
data None      = None
```

```
data Minutes = Minutes Int
```

```
data Bool      = False | True
```

```
True  :: Bool
```

```
False :: Bool
```


The almighty "data" keyword

```
data None      = None
```

```
data Minutes = Minutes Int
```

```
data Bool      = False | True
```

```
data Maybe a = Nothing | Just a
```

```
Nothing :: Maybe a
```

```
Just     :: a → Maybe a
```

```
Just 42  :: Maybe Int
```

The almighty "data" keyword

```
data None      = None
```

```
data Minutes = Minutes Int
```

```
data Bool      = False | True
```

```
data Maybe a = Nothing | Just a
```

```
data List a = Nil | Cell a (List a)
```

```
Nil :: List a
```

```
Cell :: a → List a → List a
```

```
Cell 0 (Cell 1 (Nil)) :: List Int
```

The almighty "data" keyword

```
data None      = None
```

```
data Minutes = Minutes Int
```

```
data Bool      = False | True
```

```
data Maybe a = Nothing | Just a
```

```
data List a = Nil | Cell a (List a)
```

```
Nil :: List a
```

```
Cell :: a → List a → List a
```

```
Cell 0 $ Cell 1 $ Nil :: List Int
```

The almighty "data" keyword

```
data None      = None
```

```
data Minutes = Minutes Int
```

```
data Bool      = False | True
```

```
data Maybe a = Nothing | Just a
```

```
data [a]      = [] | (a:[a])
```

```
    [] :: [a]
```

```
    (:) :: a → [a] → [a]
```

```
0:1:[] :: [Int]
```

The almighty "data" keyword

```
data None      = None
```

```
data Minutes = Minutes Int
```

```
data Bool      = False | True
```

```
data Maybe a = Nothing | Just a
```

```
data [a]      = [] | (a:[a])
```

```
[] :: [a]
```

```
(:) :: a → [a] → [a]
```

```
[0,1] :: [Int]
```

```
data User = User String Int
```

```
User      :: String → Int → User
```

```
data User = User {  
    userName :: String,  
    userAge  :: Int  
}
```

```
User      :: String → Int → User
```

```
data User = User {  
    userName :: String,  
    userAge  :: Int  
}
```

```
User      :: String → Int → User
```

```
userName  :: User → String
```

```
userAge   :: User → Int
```



```
not :: Bool → Bool  
not x = ???
```

```
not :: Bool → Bool  
not x = if x then False else True
```

```
not :: Bool → Bool  
not True  = False  
not False = True
```

#PatternMatching

```
(&&) :: Bool → Bool → Bool  
x && y = ???
```

```
(&&) :: Bool → Bool → Bool  
x && y = if x  
        then (if y then True else False)  
        else False
```

$(\&\&) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$

$\text{False} \ \&\& \ \text{False} = \text{False}$

$\text{False} \ \&\& \ \text{True} = \text{False}$

$\text{True} \ \&\& \ \text{False} = \text{False}$

$\text{True} \ \&\& \ \text{True} = \text{True}$

$(\&\&) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$

$\text{True} \ \&\& \ \text{True} = \text{True}$

$x \ \&\& \ y = \text{False}$

$(\&\&) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$

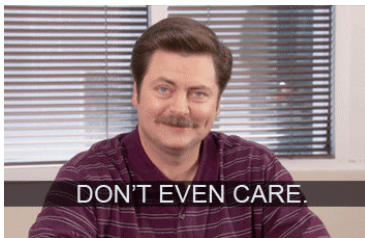
$\text{True} \ \&\& \ y = y$

$x \ \&\& \ y = \text{False}$

$(\&\&) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$

$\text{True} \ \&\& \ y = y$

$_ \ \&\& \ _ = \text{False}$



Deconstructors?

```
data Minutes = Minutes Int
```

```
add :: Minutes → Minutes → Minutes
```

```
add mx my = ???
```

Deconstructors?

```
data Minutes = Minutes Int
```

```
add :: Minutes → Minutes → Minutes
```

```
add mx my = mx + my
```

```
data Minutes = Minutes Int
```

```
add :: Minutes → Minutes → Minutes  
add (Minutes x) (Minutes y) = ???
```

```
data Minutes = Minutes Int
```

```
add :: Minutes → Minutes → Minutes
```

```
add (Minutes x) (Minutes y) = Minutes (x + y)
```

```
data Minutes = Minutes Int
```

```
add :: Minutes → Minutes → Minutes
```

```
add (Minutes x) (Minutes y) = Minutes $ x + y
```

```
data [a] = [] | (a:[a])
```

```
length :: [a] → Int
```

```
length l = ???
```

```
data [a] = [] | (a:[a])
```

```
length :: [a] → Int
```

```
length [] = ???
```

```
length (x:xs) = ???
```


Length

```
data [a] = [] | (a:[a])
```

```
length :: [a] → Int
```

```
length [] = 0
```

```
length (x:xs) = ???
```

```
data [a] = [] | (a:[a])
```

```
length :: [a] → Int
```

```
length [] = 0
```

```
length (_,xs) = 1 + length xs
```

#Recursion

The end!



- ▶ tryhaskell.org
- ▶ learnyouahaskell.com
- ▶ book.realworldhaskell.org
- ▶ haskellbook.com
- ▶ haskell.org/hoogle/