Haskell 101

nicuveo@

January 24, 2019

g2g

Today’s menu

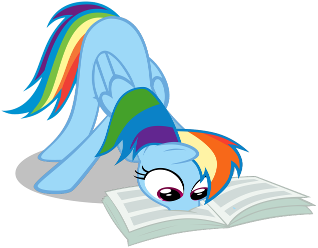
- 101
  - Concepts and generalities
  - Syntax overview
  - Data structures
  - Declaring functions

g2g

Not today

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

Prerequisites



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

GHCI is your new best friend

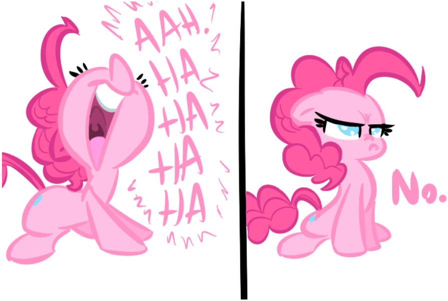
- Type expressions, get result.
- Test and debug your code.
- `:t`

Haskell is...

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

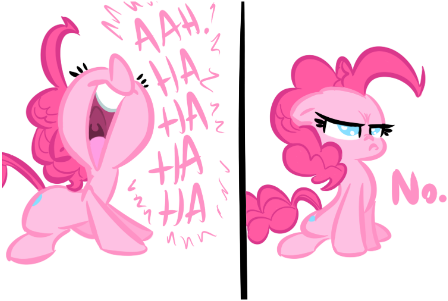
Haskell is NOT...

- A silver bullet
- For category theorists



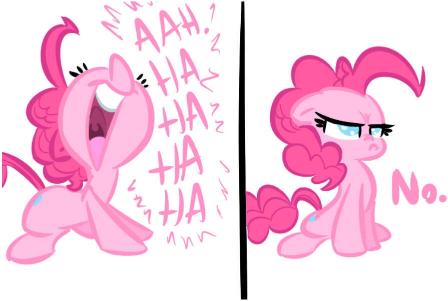
Haskell is NOT...

- A silver bullet
- For category theorists
- Hard!



Haskell is NOT...

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



g2g

Purely functional

- Everything is a function

```
f :: Int → Int
f x = x + 1
```

```
f : Z → Z
f(x) = x + 1
```

Purely functional

- Everything is a function
- Everything is immutable

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

Purely functional

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

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

Purely functional

- 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)
```

g2g

Purely functional

- 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
```

Purely functional

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

```
foo :: Int → String
```

Purely functional

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

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

g2g

Purity...

- All side effects are in IO

Purity...

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

g2g

Purity...

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

...and corruption

- $\exists$ `f :: a → IO a` (from pure to impure)
- $\nexists$ `f :: IO a → a` (from impure to pure)

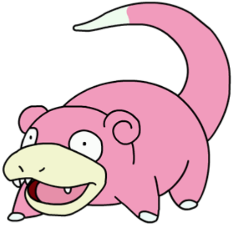
g2g

...and corruption

- $\exists$ `f :: a → IO a` (from pure to impure)
- $\nexists$ `f :: IO a → a` (from impure to pure)

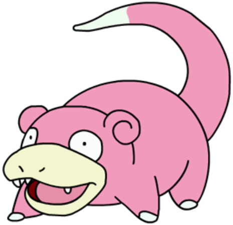
IO corrupts.

Lazy



- Deferred expression evaluation
- Not used ⟹ not computed

Lazy



- Deferred expression evaluation
- Not used ⟹ not computed

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

Reduction steps

- Strict evaluation: inner to outer

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

Reduction steps

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

add(12 + 8 , 20 + 2)

12 + 8 + 20 + 2

42

Laziness pros and cons

- Memory pitfalls

Delayed computations (but escape hatches)

Laziness pros and cons

- Memory pitfalls
- IO and parallelism pitfalls

Delayed computations (but escape hatches)

Laziness pros and cons

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

Equation reduction and short-circuiting

Laziness pros and cons

- 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]
```

Curried functions, partial application

```
f        ::       Int → Int → [Int]
```

Curried functions, partial application

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

Curried functions, partial application

```
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]
```

Curried functions, partial application

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

g2g

Quizz!

```
???  ::  (a → b) → [a] → [b]
```

Quizz!

```
???  ::  (a → b) → [a] → [b]
```

Lowercase letter: type parameter

Quizz!

```
???  ::  (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

Quizz!

```
map  ::  (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

Quizz!

```
   map  ::  (a → b) → [a] → [b]
?????? ::  (a → Bool) → [a] → [a]
```

Quizz!

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

Quizz!

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

Quizz!

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

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

Quizz!

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

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

Quizz!

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

Quizz!

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

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

Quizz!

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

```
           show :: Stuff → String
length          ::         String → Int
length  .  show :: Stuff          → Int
```

Quizz!

```
   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 → b → a) → a → [b] → a
```

Quizz with a vengeance!

```
foldl  ::  (a → b → a) → a → [b] → a
```

`(a → b → a)`	combines accumulator and value
`a`	initial accumulator
`[b]`	list of values
`a`	result

Quizz with a vengeance!

```
foldl  ::  (a → b → a) → a → [b] → a
```

`(a → b → a)`	combines accumulator and value
`a`	initial accumulator
`[b]`	list of values
`a`	result

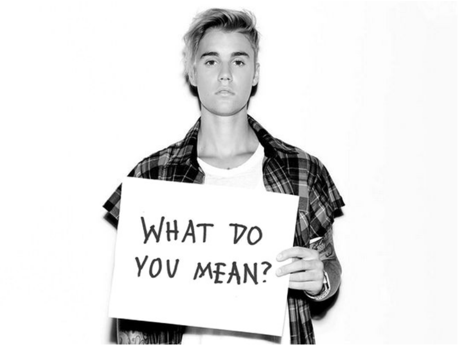
”reduce”

Algebraic data types

- Type composition
- Product and sum types
- Cardinality expressions

Algebraic data types

- Type composition
- Product and sum types
- Cardinality expressions



Type synonyms

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

Type synonyms

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

Type synonyms

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

Immutable data structures...

Immutable data structures…

- No methods…

Immutable data structures...

- No methods...
- No modifiers...

g2g

Immutable data structures...

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

Immutable data structures...

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

What's left?

g2g

Immutable data structures...

- 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
```

g2g

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]
```

g2g

Record syntax

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

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

Record syntax

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

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

Record syntax

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

```
User     :: String → Int → User

userName :: User → String
userAge  :: User → Int
```

g2g

Not

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

Not

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

Not

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

#PatternMatching

And

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

And

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

And

```
(&&) :: Bool → Bool → Bool
False && False = False
False && True  = False
True  && False = False
True  && True  = True
```

And

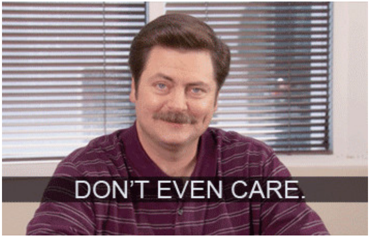
```
(&&) :: Bool → Bool → Bool
True && True = True
x    && y    = False
```

And

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

And

```
(&&) :: Bool → Bool → Bool
True && y = y
_    && _ = False
```



g2g

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
```

Deconstructors?

```
data Minutes = Minutes Int

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

Deconstructors?

```
data Minutes = Minutes Int

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

Deconstructors?

```
data Minutes = Minutes Int

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

Length

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

length :: [a] → Int
length l = ???
```

Length

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

length :: [a] → Int
length []     = ???
length (x:xs) = ???
```

Length

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

length :: [a] → Int
length []     = 0
length (x:xs) = ???
```

g2g

Length

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

length :: [a] → Int
length []     = 0
length (_:xs) = 1 + length xs
```

#Recursion

g2g

The end!

g2g

Links

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