Haskell: the best imperative programming language

Sergei Trofimovich – slyfox@gentoo.org

MLUG

Apr 26, 2014

Hia!

Warmup poll:

- Ever heard of Haskell?

Warmup poll:

- Ever heard of Haskell?
- Ever seen the code in Haskell?

Warmup poll:

- Ever heard of Haskell?
- Ever seen the code in Haskell?
- Do you know an M-word? :]

Who is expected to use it?

As defined by initial Haskell Committee the language should:

As defined by initial Haskell Committee the language should:

- be suitable for teaching, research, and applications, including building large systems.

As defined by initial Haskell Committee the language should:

- be suitable for teaching, research, and applications, including building large systems.
- be completely described via the publication of a formal syntax and semantics.

As defined by initial Haskell Committee the language should:

- be suitable for teaching, research, and applications, including building large systems.
- be completely described via the publication of a formal syntax and semantics.
- be freely available. Anyone should be permitted to implement the language and distribute it to whomever they please.

As defined by initial Haskell Committee the language should:

- be suitable for teaching, research, and applications, including building large systems.
- be completely described via the publication of a formal syntax and semantics.
- be freely available. Anyone should be permitted to implement the language and distribute it to whomever they please.
- be based on ideas that enjoy a wide consensus.

As defined by initial Haskell Committee the language should:

- be suitable for teaching, research, and applications, including building large systems.
- be completely described via the publication of a formal syntax and semantics.
- be freely available. Anyone should be permitted to implement the language and distribute it to whomever they please.
- be based on ideas that enjoy a wide consensus.
- reduce unnecessary diversity in functional programming languages.

Nice, but why should you care?

Some of The Haskell Features:

Some of The Haskell Features:

- complete value immutability (values are also called "bindings", created by construction)

Some of The Haskell Features:

- complete value immutability (values are also called "bindings", created by construction)
- referential transparency (no implicit side-effects)

Some of The Haskell Features:

- complete value immutability (values are also called "bindings", created by construction)
- referential transparency (no implicit side-effects)
- rich strong type safety (+ ML-style type inference)

Some of The Haskell Features:

- complete value immutability (values are also called "bindings", created by construction)
- referential transparency (no implicit side-effects)
- rich strong type safety (+ ML-style type inference)
- lazy evaluation model (or more precisely call-by-need semantics)

Some of The Haskell Features:

- complete value immutability (values are also called "bindings", created by construction)
- referential transparency (no implicit side-effects)
- rich strong type safety (+ ML-style type inference)
- lazy evaluation model (or more precisely call-by-need semantics)
- small but powerful core of the language

Some of The Haskell Features:

- complete value immutability (values are also called "bindings", created by construction)
- referential transparency (no implicit side-effects)
- rich strong type safety (+ ML-style type inference)
- lazy evaluation model (or more precisely call-by-need semantics)
- small but powerful core of the language
- great developer community!

Some of The Haskell Features:

- complete value immutability (values are also called "bindings", created by construction)
- referential transparency (no implicit side-effects)
- rich strong type safety (+ ML-style type inference)
- lazy evaluation model (or more precisely call-by-need semantics)
- small but powerful core of the language
- great developer community!
- great package sharing infrastructure (hackage)

Let's start from toy examples

Obligatory Hello World

00-hello.hs

```
main = print "Hello, World!"
```

Run as:

```
$ runhaskell 00-hello.hs
"Hello, World!"
```

Comments:

01-comments.hs

```
-- Single-line comment
--

{- Multi-line
   comment -}

{-| usually contains
    interesting documentation
  |-}

{- it also
   {- can be nested -} -}
```

Bindings (values):

02-basics.hs

```
a :: Int -- ':' denotes a type annotation
a = 1          -- value '1', or nullary function
my_sin :: Double -> Double
my_sin = sin -- another name of "sin", function
             -- 'sin' is already defined in Prelude
same x = x -- unmodified value
square x = x * x -- single-argument function
mul a b = a * b -- multiargument one

main = do
    print a                     -- 1
    print (my_sin 2)            -- 0.9092974268256817
    print (square 3.5)          -- 12.25
    print (mul 500 10^30)       -- 931322574615478515625000...
    print (mul 500.5 10^30)     -- 9.596711839997224e110
    print (same 12, same 'c')-- (12, 'c')
```

if and case keywords:

03-basics.hs

```haskell
-- primitive control structures
f x = if x < 3 then 24 else 42

g x = case x^x of
          1 -> "one"; 4 -> "two, right?"
          x | x >= 27 && x <= 256
            -> "around 3-4 i guess"
          _ -> "gah, that much!"
main = do
    print (f 1, f 4) -- (24, 42)
    print (g 1) -- "one"
    print (g 3) -- "around 3-4 i guess"
    print (g 6) -- "gah, that much!"
```

let and where keywords:

02-basics-let.hs

```
f1 t = let x = 4
           y = 8; z = 9
       in t - x^y / z

f2 t = t - x^y / z
    where x = 4; y = 8
          z = 9

main = do
    print (f1 42) -- -7239.777777777777
    print (f2 42) -- -7239.777777777777
```

All functions in Haskell are one-argument!

10-functions.hs

```haskell
-- All functions below have _the_same_ definition
f1 :: Double -> Double -> Double
f1 x y =  x / y

f2 :: Double -> (Double -> Double)
f2 x = \y -> x / y

f3 :: (Double -> (Double -> Double))
f3 = \x -> (\y -> x / y)

f4 :: Double -> Double -> Double
f4 = \x y -> x / y

main = do
    print (f1 2 3) -- 0.6666666666666666
    print ((f1 2) 3) -- 0.6666666666666666
```

High-Order-Function syntax is not very different from value syntax:

10-HOFs.hs

```haskell
apply :: (Double -> Double) -> Double -> Double
apply f x = f x

twice :: (Double -> Double) -> Double -> Double
twice f x = f (f x)

main = do
    print (apply sin 1) -- 0.8414709848078965
    print (twice sin 1) -- 0.7456241416655579

    print (apply (100 +) 1) -- 101.0
    print (twice (100 +) 1) -- 201.0
```

Operators are fun!

02-operators.hs

```
f x y = (x, y)
v1 = f 3 'c'
-- backticks make a function al infix operator
v2 = 3 `f` 'c' `f` 3.5

-- Operator Snail!
infixr ^\@. -- a op b op c = a op (b op c)
infixl .@/^ -- a op b op c = (a op b) op c

a ^\@. b = a ^ b -- this is also valid:
a .@/^ b = a ^ b -- (.@/^) a b = a ^ b

main = do
    print v2                    -- ((3,'c'),3.5)
    print (2 ^\@. 3 ^\@. 4) -- 2417851639229258349412352
    print (2 .@/^ 3 .@/^ 4) -- 4096
```

02-operators-are-handy.hs

```haskell
-- Prelude> :info (£)
-- (£) :: (a -> b) -> a -> b        -- Defined in ‘GHC.Base
-- infixr 0 £
-- f £ x = f x

infixr 0 .$
f .$ x = f x

-- lithpy parens
v1 x = f (g (h (i (j x))))
v2 x = f.$g.$h.$i.$j.$x -- no need to close trailing braces

v3 x = f $ g $ h $ i $ j $ x -- more common

f, g, h, i, j :: a
(f, g, h, i, j) = undefined
```

02-operators-are-handy-comp.hs

```haskell
import Data.Char (toUpper)

-- exactly as in 'Prelude.(.)'
(f <.> g) x = f (g x)

-- easy to build insane trains
v = takeWhile (< 90)
  . dropWhile (< 70)
  . filter even
  . map (100 +) $
    [ -50 .. 50]

main = do
    print $ map (succ <.> toUpper) "abcd" -- "BCDE"
    print $ map (succ . toUpper) "abcd"   -- "BCDE"
    print v -- [70,72,74,76,78,80,82,84,86,88]
```

Simple list syntax:

04-lists.hs

```haskell
-- Lists

el = []  -- empty
l1 = [1] -- single element
l2 = [1, 2, 3] -- some elements
l3 = [1 .. 10] -- range
l4 = [1.0, 1.3 .. 3] -- range with user defined step

main = do
    print (el :: [Int]) -- []
    print l1 -- [1]
    print l2 -- [1,2,3]
    print l3 -- [1,2,3,4,5,6,7,8,9,10]
    print l4 -- [1.0,1.3,1.6,1.9000000000000001,2.2,2.5,2.8
```

List comprehensions:

04-list-comprehensions.hs

```
l1 = [ x | x <- [1..10], even x]
l2 = [ (x, y) | x <- [1..10], y <- [1..10], x * y == 42]
l3 = [ (x, y)
     | x <- [1..10]
     , y <- [1..x]
     , x == 3 * y
     , even y
     ]

main = do
    print l1 -- [2,4,6,8,10]
    print l2 -- [(6,7),(7,6)]
    print l3 -- [(6,2)]
```

Every expression in haskell has a type:

05-types.hs

```
-- from 02-basics.hs

a :: Integer
a = 1          -- value
my_sin :: Double -> Double
my_sin = sin -- another name of "sin", function
same :: a -> a
same x = x
square :: Num a => a -> a
square x = x * x -- single-argument function
mul :: Num a => a -> a -> a
mul a b = a * b -- multiargument one
```

Some standard types:

06-standard-types.hs

```
i = 42 :: Int
bi = 42424242424242424242424242424242 :: Integer
fl = 42.5 :: Float
d = 42.5 :: Double
c = 'z' :: Char
s = "hello" :: String
b = True :: Bool

tup = (1, 3.5) :: (Int, Float)
lt = [(1, 'a'), (2, 'b')] :: [(Int, Char)]

f = (\x -> x + 10) :: Double -> Double
```

User defined types:

06-user-defined-types.hs

```haskell
data MyT = V1 | V2 | V3 -- enum-like, sum type
list_of_t :: [MyT]
list_of_t = [V1, V1, V3, V2]

data MyT2 = IV Int | CV Char | DV Double
          | T2 | OV Int Char Double
--          IV :: Int -> MyT2
--          OV :: Int -> Char -> Double -> MyT2
list_of_t2 :: [MyT2]
list_of_t2 = [IV 42, CV 'z', DV 3.5, T2, OV 15 'c' 3.5]
l :: Int
l = length list_of_t2

main = do
    print l -- 5
```

More user defined types:

07-more-user-defined-types.hs

```
pair = (17, "seventeen") -- 2-tuple
triple = ("three of", 'u', 's') -- 3-tuple

data MyT = V    Int    Char    String -- product-type
         | Recurse MyT
--         V :: Int -> Char -> String -> MyT

v = V 42 'a' "sneaky"

s_to_t :: String -> MyT
s_to_t s = V 0 'z' s

i_am_also_valid = Recurse (
                    Recurse (
                     Recurse (V 1 'a' "threefold")))
```

Inspecting a value:

07-deconstruct.hs

```haskell
import Data.Char (toUpper)
data MyT = V    Int    Char    String
         | Recurse MyT

name :: MyT -> String
name (V 0 _ n) = n ++ " 0 value"
name (V _ _ n) = n ++ " value"
name (Recurse nested) = name nested

v1 = V 32 'c' (map toUpper "shallow")
v2 = Recurse $ Recurse $ V 0 '!' ("de" ++ "ep")

main = do
    print $ name v1 -- "SHALLOW value"
    print $ name v2 -- "deep 0 value"
```

Let's grow some trees:

08-a-tree.hs

```
data IntTree = ILeaf Int | ITree [IntTree]
data DoubleTree = DLeaf Double | DTree [DoubleTree]

itree :: IntTree
itree = ITree [ITree [ ILeaf 4
                     , ILeaf 5
                     ]
              , ILeaf 8
              ]

dtree :: DoubleTree
dtree = DTree [DTree [ DLeaf 4.5
                     , DLeaf 5.1
                     ]
              , DLeaf 8.0
              ]
```

Haskell also has parametric polymorphism:

09-polymorphic.hs

```
data PolyTree a = Leaf a
                | Tree [PolyTree a]
-- tree of 'Int's
itree :: PolyTree Int
itree = Tree [ Tree [ Leaf 4, Leaf 5 ]
             , Leaf 8
             ]
-- tree of 'Doubles's
dtree :: PolyTree Double
dtree = Tree [ Tree [Leaf 4.5 , Leaf 5.1]
             , Leaf 8.0
             ]
-- or even tree of trees of 'Int's
ttree :: PolyTree (PolyTree Int)
ttree = Leaf (Leaf 42)
```

And the Poly order can grow as well:

10-polymorphic.hs

```haskell
-- 2 type arguments: a - arg_t, r - result_t
data F1or2 a r = F1 (a -> r)
               | F2 (a -> a -> r)
f :: F1or2 Double Double
f = F1 cos
g :: F1or2 Double Double
g = F2 (+) -- (+) :: Num a => a -> a -> a

app_zeroes :: Num a => F1or2 a a -> a
-- pattern match on all constructors
app_zeroes (F1 f') = f' 0
app_zeroes (F2 g') = g' 0 0

main = do
    print (app_zeroes f) -- 1.0
    print (app_zeroes g) -- 0.0
```

Let's look at this example a bit closer:

11-polymorphic-error.hs

```
plus1 :: a -> a
plus1 x = x + 1
{- The compiler error is:

    No instance for (Num a) arising from a use of ‘+’
    Possible fix:
      add (Num a) to the context of
        the type signature for plus1 :: a -> a
    In the expression: x + 1
    In an equation for ‘plus1’: plus1 x = x + 1
-}
correct_plus1 :: Num a => a -> a
correct_plus1 x = x + 1
{- this one is fine -}
```

Not every type fits there. It's a type error!

How many function definitions you can write for a following function type:

12-polyquiz.hs

```
f :: a -> a
f x = ???
```

How many function definitions you can write for a following function type:

12-polyquiz.hs

```
f :: a -> a
f x = ???
```

- `f x = x`

How many function definitions you can write for a following function type:

12-polyquiz.hs

```
f :: a -> a
f x = ???
```

- `f x = x`
- `f x = f x`

How many function definitions you can write for a following function type:

12-polyquiz.hs

```
f :: a -> a
f x = ???
```

- `f x = x`
- `f x = f x`
- `f x = f (f x)`

How many function definitions you can write for a following function type:

12-polyquiz.hs

```
f :: a -> a
f x = ???
```

- `f x = x`
- `f x = f x`
- `f x = f (f x)`
- `f x = f (f ... (f x) ... )`

How many function definitions you can write for a following function type:

12-polyquiz.hs

```
f :: a -> a
f x = ???
```

- `f x = x`
- `f x = f x`
- `f x = f (f x)`
- `f x = f (f ... (f x) ... )`
- `f x = x `seq` f x`

Now suppose we would need to write a function that would return default value for a given type (different for each type):

13-typeclasses.hs

```
def :: ???
def = ???

data MyT = MyV deriving Show

main = do
    print (def :: Int) -- we want 42 here
    print (def :: Char) -- and '!' here
    print (def :: Double) -- and 13.5 here
    print (def :: MyT) -- and MyV here
```

It is known as an ad-hoc polymorphism:

14-ad-hoc-polymorphism.hs

```haskell
def :: Def a => a
def = getDefault
data MyT = MyV deriving Show
main = do
    print (def :: Int) -- we want 42 here
    print (def :: Char) -- and '!' here
    print (def :: Double) -- and 13.5 here
    print (def :: MyT) -- and MyV here

class Def a where
    getDefault :: a
instance Def Int where
    getDefault = 42
instance Def Char   where getDefault = '!'
instance Def Double where getDefault = 13.5
instance Def MyT    where getDefault = MyV
```

Typeclasses are open. It means you can make any type an instance of any other class you like.

15-standard-typeclasses.hs

```
class Eq a where
  (==) :: a -> a -> Bool

class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  (-) :: a -> a -> a
  abs :: a -> a
  fromInteger :: Integer -> a
-- There are many other standard typeclasses:
-- Ord, Bounded, Read, Show, ...
```

23-quickcheck.hs

```
import qualified Test.QuickCheck as QC
import           Test.QuickCheck ((==>))

prop_plus_minus :: Int -> Bool
prop_plus_minus a =
    a == (a + 1) - 1

prop_div x y =
  (y /= (0 :: Double)) ==>
    (x == x / y * y)

main = do
    QC.quickCheck prop_plus_minus -- +++ OK, passed 100 tes
    QC.quickCheck prop_div -- *** Failed! Falsifiable (aft
                           --           tests and 1116 shri
                           -- 5.0e-324
                           -- 1.500002125805702
```

Sometimes it's handy to name some type in a shorter form, or give it more explicit name.

16-type-aliases.hs

```haskell
import Text.Printf

-- from now on 'Voltage' and 'Double'
-- can be used interchangeably
type Voltage = Double

eu_voltage :: Voltage
eu_voltage = 220

peak_voltage :: Voltage -> Voltage
peak_voltage rms = rms * sqrt 2

main = do
    printf "%.2f\n" (peak_voltage eu_voltage) -- 311.13
```

It's interesting to look at what type does main have:

17-main.hs

```haskell
-- It's some polymorphic 'IO' bit applied to '()'
main :: IO ()
main = print "heh"

-- '()' is a value of type '()'
-- like 0-tuple. Pronounced as 'unit'
l :: [()]
l = [(), (), ()]
ll = length l -- 3

-- IO is defined in `GHC.Types'
type IO a = RealWorld# -> (RealWorld#, a)
```

So the secret of Creation is just a function, accepting existing state of real world, and building new state of the world. No more, no less.

There is way more, that that.
How about looking at more real examples?

Copying a file, cp style:

20-file-copy.hs

```
import qualified Data.ByteString as B
import System.Environment (getArgs)
import System.Exit
main :: IO ()
main = do
    args <- getArgs
    case args of
        [source, destination]
            -> do file_data <- B.readFile source
                  B.writeFile destination file_data
        _   -> usage

usage :: IO ()
usage = do
    putStrLn "usage file_copy <source> <dest>"
    exitFailure
```

Copying a file, cp style, desugared do-notation:

20-file-copy-desugared.hs

```
import qualified Data.ByteString as B
import System.Environment (getArgs)
import System.Exit
main :: IO ()
main =
    getArgs >>= \args ->
    case args of
        [source, destination]
            -> B.readFile source >>= \file_data ->
                B.writeFile destination file_data
        _   -> usage

usage :: IO ()
usage =
    putStrLn "usage file_copy <source> <dest>" >>= \_ ->
    exitFailure
```

20-file-copy-types.hs

```haskell
-- Data.ByteString
readFile  :: FilePath -> IO ByteString
writeFile :: FilePath -> ByteString -> IO ()

-- System.Environment
getArgs :: IO [String]

-- Prelude
putStrLn :: String -> IO ()
```

do-notation has nothing to do with 'IO a'!

21-blaze-builder-do-notation.hs

```haskell
{-# LANGUAGE OverloadedStrings #-}
import qualified Text.Blaze.Html5 as H
import qualified Text.Blaze.Html.Renderer.Text as HR

hi :: String -> H.Html
hi user = H.html $ do
  H.head $
    H.title "Good morning!"
  H.body $ do
    H.h1 "Big text"
    H.p ("Hello " >> H.toHtml user >> "!")

main = do
    print $ HR.renderHtml (hi "you")
    -- "<html><head><title>Good morning!</title></head>
    -- <body><h1>Big text</h1><p>Hello you!</p></body>
    -- </html>"
```

22-do-desugaring.hs

```
do {e}                      =    e

do {e;stmts}                =    e >> do {stmts}

do {p <- e; stmts}          =    let ok p = do {stmts}
                                     ok _ = fail "..."
                                 in e >>= ok

do {let decls; stmts}       =    let decls in do {stmts}
```

24-concurrency.hs

```haskell
import              Control.Concurrent (forkIO, threadDelay)
import              Control.Monad (forever)

say_them n_secs phrase = do
    threadDelay (n_secs * 10^6) -- once in n seconds
    putStrLn phrase

main = do
    forkIO $
        forever $ say_them 1 "i am new thread!"
    forever     $ say_them 2 "i am main thread!"
```

24-chans-concurrency.hs

```
import           Control.Concurrent (forkIO)
import qualified Control.Concurrent.Chan as CCC
import           Control.Monad (forever, replicateM)
import qualified System.Random as R

data Something = VI Int | VS String | Done deriving Show
producer c = do
    i <- R.randomIO
    say (VI i) >> say (VS "seen int?")
    say Done
    putStrLn "I'm done, bye"
  where say = CCC.writeChan c
main = do
    c <- CCC.newChan
    replicateM 32 (forkIO (producer c))
    forever $
        CCC.readChan c >>= print
```

The real implementation of those scary things:

25-forever-and-when.hs

```
import qualified System.Random as R
import Data.Word

forever :: (Monad m) => m a -> m b
forever action = do
    action
    forever action

when :: (Monad m) => Bool -> m () -> m ()
when p s = if p then s else return ()

main :: IO ()
main = forever $ do
    i <- R.randomIO :: IO Word16
    when (i == 42) $
        putStrLn "Hah, 42!"
```

Interfacing with with C language is also trivial:

26-c-ffi.hs

```
import Foreign.C.Types
import Foreign.Ptr
import Foreign.Storable (peek)

-- 'int rand(void);'
foreign import ccall "stdlib.h rand" c_rand :: IO CInt
-- 'int nice(int);'
foreign import ccall "unistd.h nice" c_nice :: CInt -> IO C

main = do
    c_rand >>= print -- 1804289383
    c_rand >>= print -- 846930886
    c_nice 19 >>= print -- 19
    c_nice (-19) >>= print -- -1, failed
```

More advanced cases of passing pointers there and back:

27-more-c-ffi.hs

```
-- pointer to C function:
foreign import ccall "stdlib.h &free"
    p_free :: FunPtr (Ptr a -> IO ())

-- convert haskell function to stable C pointer
type Compare = Int -> Int -> Bool
foreign import ccall "wrapper"
    mkCompare :: Compare -> IO (FunPtr Compare)

-- convert function pointer to callable haskell function
type IntFunction = CInt -> IO ()
foreign import ccall "dynamic"
    mkFun :: FunPtr IntFunction -> IntFunction
```

One of the most controversial things in haskell is laziness:

28-laziness.hs

```
import qualified Data.ByteString.Lazy as B8
infinite_list = [1,2..]

main = do
    -- all the randomness I will ever have!
    ur <- B8.readFile "/dev/urandom"

    -- skip first megabyte, pick 10 bytes
    let p100 :: Integer
        p100 = product $ take 100 infinite_list
        a_bit :: B8.ByteString
        a_bit = B8.take 10 $ B8.drop (10^6) ur

    print p100  -- 933262154439441526816992388562667004907
    print a_bit -- "<\CAN\171n\a\167\228{o\ESC"
```

But lazines makes evaluation model quite intricate.

Haskell language is named after **Haskell Brooks Curry**.
An American mathematician and logician.

Haskell language is named after **Haskell Brooks Curry**.
An American mathematician and logician.

There are three languages named after his name.

Haskell language is named after **Haskell Brooks Curry**.
An American mathematician and logician.

There are three languages named after his name.

Guess how they are called.

- Haskell

Haskell language is named after **Haskell Brooks Curry**.
An American mathematician and logician.

There are three languages named after his name.

Guess how they are called.

- Haskell
- Brooks

Haskell language is named after **Haskell Brooks Curry**.
An American mathematician and logician.

There are three languages named after his name.

Guess how they are called.

- Haskell
- Brooks
- Curry

One of early Haskell compilers had a name hbc.

Haskell's brief history

Haskell's brief history

- **September 1987**: Initial meeting at **FPCA**, Portland, Oregon. Need for Lazy **FPL**.

Haskell's brief history

- **September 1987**: Initial meeting at **FPCA**, Portland, Oregon. Need for Lazy **FPL**.
- **January 1988**: A multi-day meeting at Yale University. ~~Curry~~Haskell name was chosen.

Haskell's brief history

- **September 1987**: Initial meeting at **FPCA**, Portland, Oregon. Need for Lazy **FPL**.
- **January 1988**: A multi-day meeting at Yale University. ~~Curry~~Haskell name was chosen.
- **1 April 1990**: The Haskell version 1.0 Report (125 pages), edited by Hudak and Wadler.

Haskell's brief history

- **September 1987**: Initial meeting at **FPCA**, Portland, Oregon. Need for Lazy **FPL**.
- **January 1988**: A multi-day meeting at Yale University. ~~Curry~~Haskell name was chosen.
- **1 April 1990**: The Haskell version 1.0 Report (125 pages), edited by Hudak and Wadler.
- A few more releases....

Haskell's brief history

- **September 1987**: Initial meeting at **FPCA**, Portland, Oregon. Need for Lazy **FPL**.
- **January 1988**: A multi-day meeting at Yale University. ~~Curry~~Haskell name was chosen.
- **1 April 1990**: The Haskell version 1.0 Report (125 pages), edited by Hudak and Wadler.
- A few more releases....
- **February 1999**: The Haskell 98 Report: Language and Libraries (150 + 89 pages), edited by Peyton Jones and Hughes.

Haskell's brief history

- **September 1987**: Initial meeting at **FPCA**, Portland, Oregon. Need for Lazy **FPL**.
- **January 1988**: A multi-day meeting at Yale University. ~~Curry~~Haskell name was chosen.
- **1 April 1990**: The Haskell version 1.0 Report (125 pages), edited by Hudak and Wadler.
- A few more releases....
- **February 1999**: The Haskell 98 Report: Language and Libraries (150 + 89 pages), edited by Peyton Jones and Hughes.
- **1999**: The Haskell Committee ceased to exist.

Haskell's brief history

- **September 1987**: Initial meeting at **FPCA**, Portland, Oregon. Need for Lazy **FPL**.
- **January 1988**: A multi-day meeting at Yale University. ~~Curry~~Haskell name was chosen.
- **1 April 1990**: The Haskell version 1.0 Report (125 pages), edited by Hudak and Wadler.
- A few more releases....
- **February 1999**: The Haskell 98 Report: Language and Libraries (150 + 89 pages), edited by Peyton Jones and Hughes.
- **1999**: The Haskell Committee ceased to exist.
- Various implementations appear with different extensions to the base language.

Haskell's brief history

- **September 1987**: Initial meeting at **FPCA**, Portland, Oregon. Need for Lazy **FPL**.
- **January 1988**: A multi-day meeting at Yale University. ~~Curry~~Haskell name was chosen.
- **1 April 1990**: The Haskell version 1.0 Report (125 pages), edited by Hudak and Wadler.
- A few more releases....
- **February 1999**: The Haskell 98 Report: Language and Libraries (150 + 89 pages), edited by Peyton Jones and Hughes.
- **1999**: The Haskell Committee ceased to exist.
- Various implementations appear with different extensions to the base language.
- **July 2010**: The Haskell 2010 Report: Language and Libraries (152 + 142 pages), edited by Simon Marlow.

Thank you!