Haskell

- Purely functional: Functions are like mathematical functions.
- Flexible static types: polymorphism and type classes.
- Easy definition of data types.
- Lazy: Don’t evaluate what is not needed.
- Fast and interactive.
Functional Hello World: The Fibonacci sequence

Functions are defined via equations:

```haskell
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```
Everything has a (possibly polymorphic) type:

```
fib :: Integer -> Integer
[1,2,3] :: [Int]
length :: [a] -> Int
length :: forall a. [a] -> Int
(:) :: a -> [a] -> [a]
(+) :: (Num a) => a -> a -> a
(\x -> 2*x) :: (Num a) => a -> a
```
Types are Inferred

\[
\begin{align*}
\text{lengthList} \; [] &= 0 \\
\text{lengthList} \; (x:xs) &= 1 + \text{lengthList} \; xs
\end{align*}
\]

Haskell infers the most general type:

*Main> :t lengthList
lengthList :: (Num t1) => [t] -> t1
Curried Functions

\[ \text{plus } x \ y = x + y \]

*Main> :t plus
plus :: (Num a) => a -> a -> a
plus :: (Num a) => a -> (a -> a)

Partial evaluation

*Main> let plus2 = plus 2
*Main> :t plus2
plus2 :: Integer -> Integer
*Main> plus2 13
15
data Tree a = Leaf a |
    Branch (Tree a) (Tree a)

deriving (Eq, Show)

tree = Branch (Leaf 2)
    (Branch (Leaf 3) (Leaf 4))

leaves (Leaf l) = [l]
leaves (Branch t1 t2) = (leaves t1) ++
    (leaves t2)
Laziness

\[
\text{naturals} = [1..] \\
\text{ones} = 1:\text{ones} \\
\text{primes} :: [\text{Integer}] \\
\text{primes} = \text{sieve} [2..] \\
\quad \text{where} \\
\quad \quad \text{sieve} (p:xs) = \\
\quad \quad \quad p : \text{sieve} [x|x <- xs, x \mod p > 0] \\
\text{fibs} :: [\text{Integer}] \\
\text{fibs} = 0:1: \text{zipWith} (+) \text{fibs} (\text{tail} \ \text{fibs})
\]
newton :: Double -> (Double -> Double) -> (Double -> Double) -> [Double]
newton x0 f f' = iterates
    where
        iterates =
            x0 : map (\x -> x - f x / f' x) iterates

limit :: [Double] -> Double
limit (x0:x1:xs) | abs (x0-x1) < 1e-8 = x1
                 | otherwise = limit (x1:xs)

Functions that produce data do not need to worry about termination conditions.
Functions that do Stuff

- Are called actions and have types of the form

```haskell
action :: a -> IO b
putStrLn :: String -> IO ()
getLine :: IO String
```

- Actions are sequenced in do blocks.

```haskell
main :: IO ()
main = do
  putStrLn "Enter a number."
  x <- getLine
  putStrLn ("Two times the number is: " ++ (show (2 * read x)) ++ ".")
```
### General do blocks

<table>
<thead>
<tr>
<th>triples = do</th>
</tr>
</thead>
<tbody>
<tr>
<td>x ← [1..20]</td>
</tr>
<tr>
<td>y ← [1..20]</td>
</tr>
<tr>
<td>z ← [1..30]</td>
</tr>
<tr>
<td>if (x^2 + y^2 == z^2)</td>
</tr>
<tr>
<td>then return (x, y, z)</td>
</tr>
<tr>
<td>else []</td>
</tr>
</tbody>
</table>

- Sequencing of operations in do blocks and is programmable: monads.
- There are monads for:
  - IO: Everything.
  - Lists: Operations over several lists.
  - Maybe: operations that can fail.
  - ST: Mutable variables and arrays.
  - More: monads for continuations, parsers, etc.