Haskell

- Haskell is a programming language that is
  - **Similar to** ML: general-purpose, strongly typed, higher-order, functional, supports type inference, supports interactive and compiled use
  - **Different from** ML: lazy evaluation, purely functional, rapidly evolving type system.

- Designed by committee in 80’s and 90’s to unify research efforts in lazy languages.
  - Haskell 1.0 in 1990, Haskell ’98, Haskell’ ongoing.
  - “**A history of Haskell: Being lazy with class**” HOPL 3

Paul Hudak

John Hughes

Simon Peyton Jones

Phil Wadler
In Haskell, \( f :: A \rightarrow B \) means for every \( x \in A \),

\[
f(x) = \begin{cases} 
\text{some element } y = f(x) \in B \\
\text{run forever}
\end{cases}
\]

In words, “if \( f(x) \) terminates, then \( f(x) \in B \).”

In ML, functions with type \( A \rightarrow B \) can throw an exception, but not in Haskell.
Functions that take other functions as arguments or return as a result are higher-order functions.

Common Examples:
- **Map**: applies argument function to each element in a collection.
- **Reduce**: takes a collection, an initial value, and a function, and combines the elements in the collection according to the function.

```
list = [1,2,3]
r = foldl (\accumulator i -> i + accumulator) 0 list
```

Google uses Map/Reduce to parallelize and distribute massive data processing tasks. (Dean & Ghemawat, OSDI 2004)
Basic Overview of Haskell

- Interactive Interpreter (ghci): read-eval-print
  - ghci infers type before compiling or executing
    Type system does not allow casts or other loopholes!

- Examples
  Prelude> (5+3)-2
  6
  it :: Integer
  Prelude> if 5>3 then "Harry" else "Hermione"
  "Harry"
  it :: [Char]  -- String is equivalent to [Char]
  Prelude> 5==4
  False
  it :: Bool
Simple Compound Types

- **Tuples**
  - $(4, 5, \text{"Griffendor"}) :: (\text{Integer}, \text{Integer}, \text{String})$

- **Lists**
  - $[] :: [a]$  
    - polymorphic type
  - $1 : [2, 3, 4] :: [\text{Integer}]$  
    - infix cons notation

- **Records**
  - `data Person = Person {firstName :: String, lastName :: String}`
  - `hg = Person { firstName = \text{"Hermione"}, lastName = \text{"Granger"}}`
Patterns and Declarations

- Patterns can be used in place of variables
  \[ \text{pat} ::= \text{var} \mid \text{tuple} \mid \text{cons} \mid \text{record} \ldots \]

- Value declarations
  - General form
    \[ \text{pat} = \text{exp} \]
  - Examples
    myTuple = ("Flitwick", "Snape")
    (x,y) = myTuple
    myList = [1, 2, 3, 4]
    zs = myList
    z:zs = myList
  - Local declarations
    let (x,y) = (2, "Snape") in x * 4
Functions and Pattern Matching

- Anonymous function
  - \(x \rightarrow x+1\) like Lisp lambda, function (...) in JS

- Declaration form
  - \(<\text{name}> <\text{pat}_1> = <\text{exp}_1>\)
  - \(<\text{name}> <\text{pat}_2> = <\text{exp}_2> \ldots\)
  - \(<\text{name}> <\text{pat}_n> = <\text{exp}_n> \ldots\)

- Examples
  - \(f (x,y) = x+y\) actual parameter must match pattern \((x,y)\)
  - length [] = 0
  - length \((x:s)\) = 1 + length\(s\)
Datatype Declarations

- **Examples**
  - `data Color = Red | Yellow | Blue`
    - elements are Red, Yellow, Blue
  - `data Atom = Atom String | Number Int`
    - elements are Atom “A”, Atom “B”, ..., Number 0, ...
  - `data List = Nil | Cons (Atom, List)`
    - elements are Nil, Cons(Atom “A”, Nil), ...
    - Cons(Number 2, Cons(Atom(“Bill”), Nil)), ...

- **General form**
  - `data <name> = <clause> | ... | <clause>`
  - `<clause> ::= <constructor> |<constructor> <type>`
  - Type name and constructors must be Capitalized.
No side effects. At all.

- A call to `reverse` returns a new list; the old one is unaffected.

  ```haskell
  reverse :: [w] -> [w]
  prop_RevRev l = reverse(reverse l) == l
  ```

- A variable `'l'` stands for an immutable `value`, not for a `location` whose value can change.
Things to Notice

Purity makes the interface explicit.

- Takes a list, and returns a list; that's all.

- Takes a list; may modify it; may modify other persistent state; may do I/O.

reverse:: [w] -> [w] -- Haskell

void reverse( list l ) /* C */
Haskell is a **lazy** language

Functions and data constructors don’t evaluate their arguments until they need them.

Programmers can write control-flow operators that have to be built-in in eager languages.

```
cond :: Bool -> a -> a -> a
cond True  t e = t
cond False t e = e

(||) :: Bool -> Bool -> Bool
True || x = True
False || x = x
```
A Lazy Paradigm

- Generate all solutions (an enormous tree)
- Walk the tree to find the solution you want

```haskell
nextMove :: Board -> Move
nextMove b = selectMove allMoves
  where
    allMoves = allMovesFrom b
```

A gigantic (perhaps infinite) tree of possible moves
datatype 'a tree = Leaf of 'a | Node of ('a tree * 'a tree)

fun fringe (Leaf a) = [a]
  | fringe (Node (x, y)) = (fringe x) @ (fringe y)

fun samefringe t1 t2 = (fringe t1) = (fringe t2)

data Tree a = Leaf a | Node (Tree a) (Tree a)

fringe (Leaf a) = [a]
fringe (Node x y) = (fringe x) ++ (fringe y)

samefringe t1 t2 = (fringe t1) == (fringe t2)
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) = x : y : (merge xs ys)

a = [ x^3 | x <- [0..]]
b = [ 3^x | x <- [0..]]
double = [ x^2 | x<- [0..]]
c = merge [0..] double
d = [ -x | x <-[1..]]

add [] [] = []
add (x:xs) (y:ys) = (x+y) : add xs ys

fib = 0 : 1 : (add fib (tail fib))