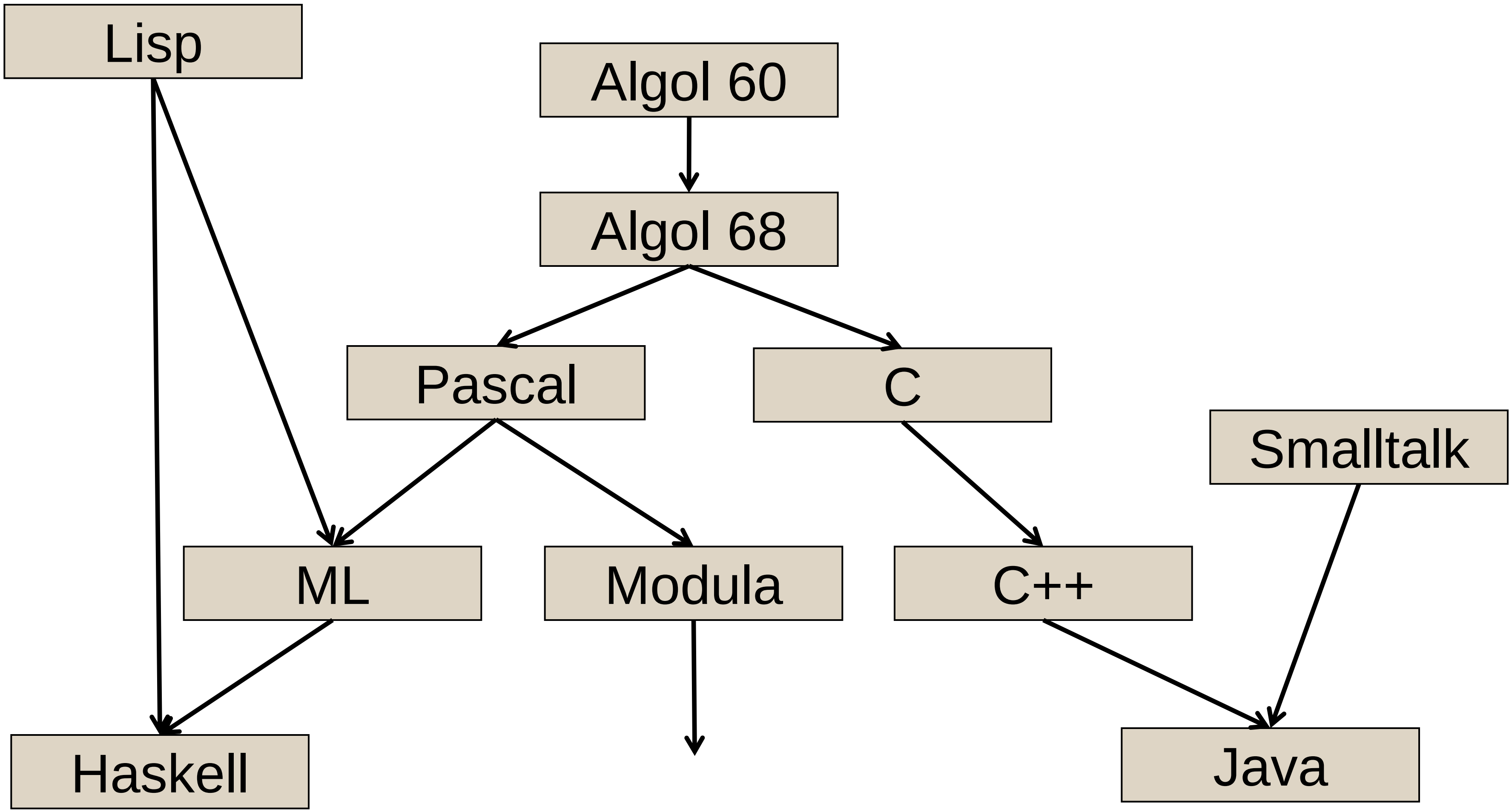
CS 242 Autumn 2012

Introduction to Haskell

(slides modified from those created by John Mitchell and Kathleen Fisher)

Language Evolution



Many others: Algol 58, Algol W, Scheme, EL1, Mesa (PARC), Modula-2, Oberon, Modula-3, Fortran, Ada, Perl, Python, Ruby, C#, Javascript, F#…

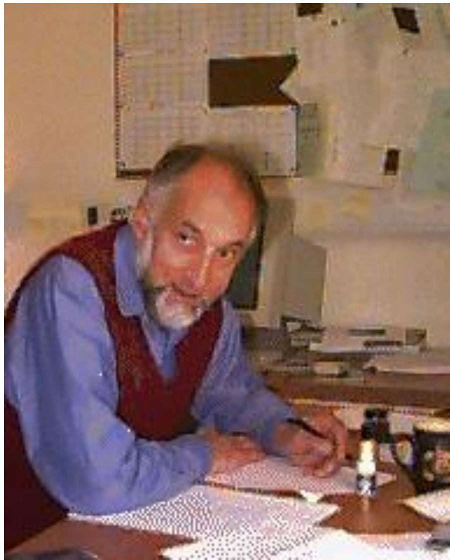
C Programming Language

Dennis Ritchie, ACM Turing Award for Unix

- Statically typed, general purpose systems programming language
- Computational model reflects underlying machine
- Relationship between arrays and pointers
  - An array is treated as a pointer to first element
  - E1[E2] is equivalent to ptr dereference: *((E1)+(E2))
  - Pointer arithmetic is not common in other languages
- Not statically type safe
  - If variable has type float, no guarantee value is floating pt
- Ritchie quote
  - "C is quirky, flawed, and a tremendous success"

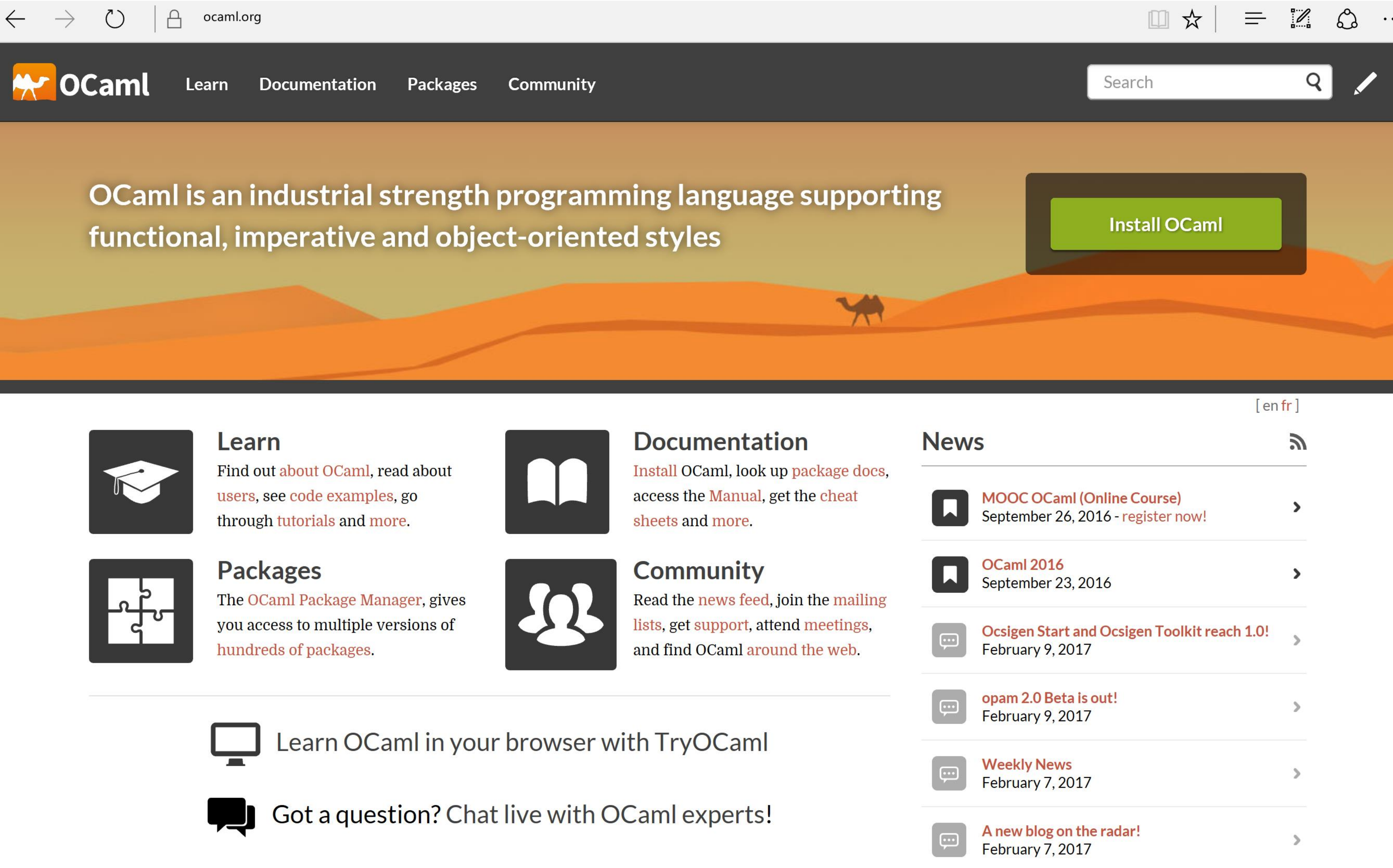
ML programming language

- Statically typed, general-purpose programming language
  - "Meta-Language" of the LCF theorem proving system
- Type safe, with formal semantics
- Compiled language, but intended for interactive use
- Combination of Lisp and Algol-like features
  - Expression-oriented
  - Higher-order functions
  - Garbage collection
  - Abstract data types
  - Module system
  - Exceptions
- Used in printed textbook as example language



Robin Milner, ACM Turing-Award for ML, LCF Theorem Prover, …

OCaml



ocaml.org
OCaml
Learn
Documentation
Packages
Community
Search
OCaml is an industrial strength programming language supporting functional, imperative and object-oriented styles
Install OCaml
[en fr]
Learn
Find out about OCaml, read about users, see code examples, go through tutorials and more.
Documentation
Install OCaml, look up package docs, access the Manual, get the cheat sheets and more.
Packages
The OCaml Package Manager, gives you access to multiple versions of hundreds of packages.
Community
Read the news feed, join the mailing lists, get support, attend meetings, and find OCaml around the web.
Learn OCaml in your browser with TryOCaml
Got a question? Chat live with OCaml experts!
News
MOOC OCaml (Online Course)
September 26, 2016 - register now!
OCaml 2016
September 23, 2016
Ocsigen Start and Ocsigen Toolkit reach 1.0!
February 9, 2017
opam 2.0 Beta is out!
February 9, 2017
Weekly News
February 7, 2017
A new blog on the radar!
February 7, 2017

Haskell

- Haskell programming language is
  - Similar to ML: general-purpose, strongly typed, higher-order, functional, supports type inference, interactive and compiled use
  - Different from ML: lazy evaluation, purely functional core, 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

Haskell B Curry

- Combinatory logic
  - Influenced by Russell and Whitehead
  - Developed combinators to represent substitution
  - Alternate form of lambda calculus that has been used in implementation structures
- Type inference
  - Devised by Curry and Feys
  - Extended by Hindley, Milner

Although “Currying” and “Curried functions” are named after Curry, the idea was invented by Schoenfinkel earlier


Miranda homepage
miranda.org.uk
Amazon
Camp Sierra
CellarTracker
Chase
Craigslist
Facebook
Fidelity.com
Google
Netflix
United
Other bookmarks
λ
Miranda™
A Non-strict Polymorphic Functional Language
Miranda is a pure, non-strict, polymorphic, higher order functional programming language designed by David Turner in 1983-6. The language was widely taken up, both for research and for teaching, and had a strong influence on the subsequent development of the field, influencing in particular the design of Haskell, to which it has many similarities. Miranda is however a simpler language. Here is a short description. For more detail you can read the Overview paper below, look at these examples of Miranda scripts, or read the definitions in the Miranda standard environment.
DOWNLOADS includes Linux, Windows(Cygwin), Intel/Solaris, SUN/Solaris & Mac versions. To be informed of new versions add yourself to the mailing list.
Why the name Miranda?
Browsable version of Miranda system manual
Browsable version of UNIX manual page mira.1
en.wikipedia.org/wiki/Miranda_(programming_language)

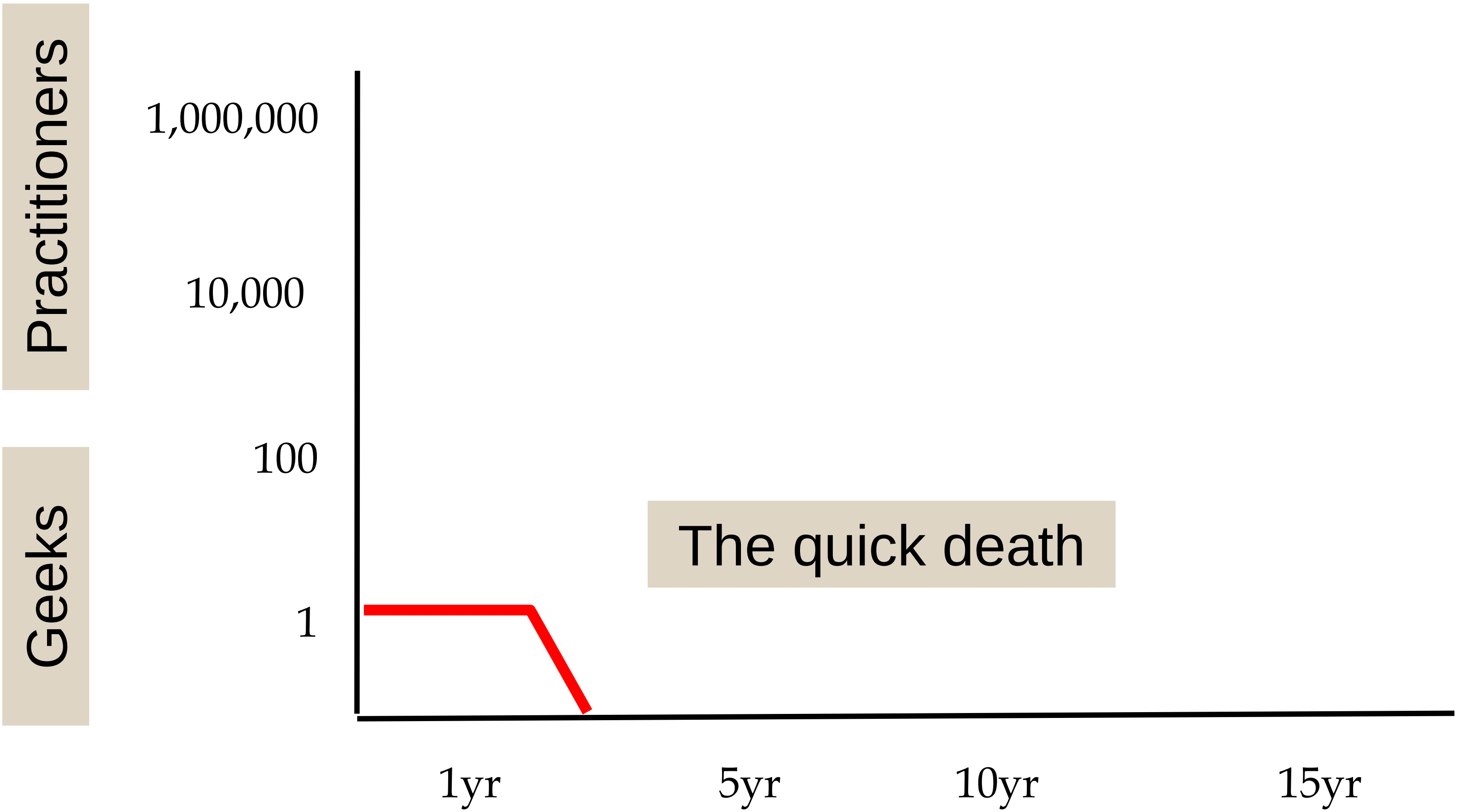
Why Study Haskell?

- Good vehicle for studying language concepts
- Types and type checking
  - General issues in static and dynamic typing
  - Type inference
  - Parametric polymorphism
  - Ad hoc polymorphism (aka, overloading)
- Control
  - Lazy vs. eager evaluation
  - Tail recursion and continuations
  - Precise management of effects

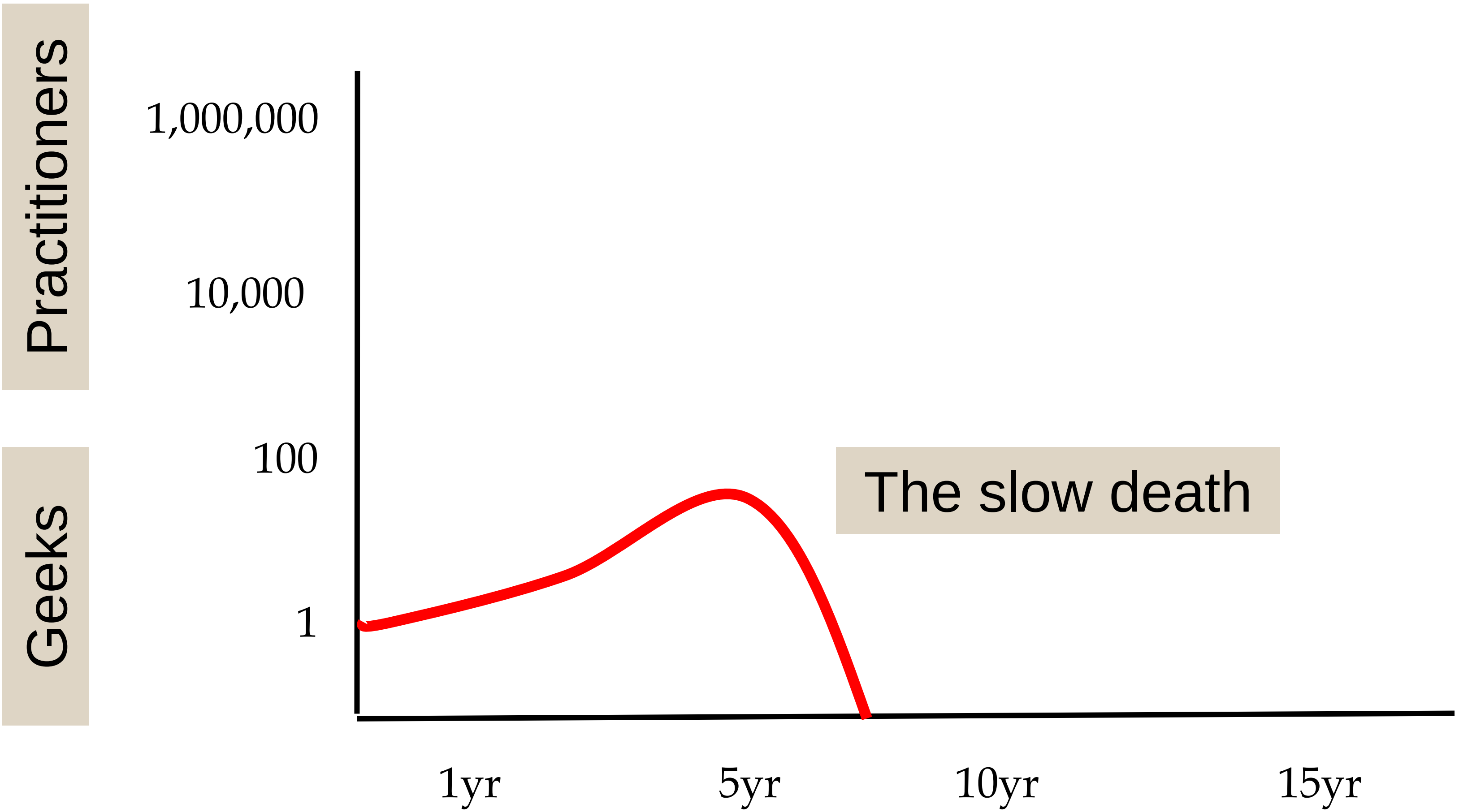
Why Study Haskell?

- Functional programming will make you think differently about programming.
  - Mainstream languages are all about state
  - Functional programming is all about values
- Haskell is “cutting edge”
  - A lot of current research is done using Haskell
  - Rise of multi-core, parallel programming likely to make minimizing state much more important
- New ideas can help make you a better programmer, in any language

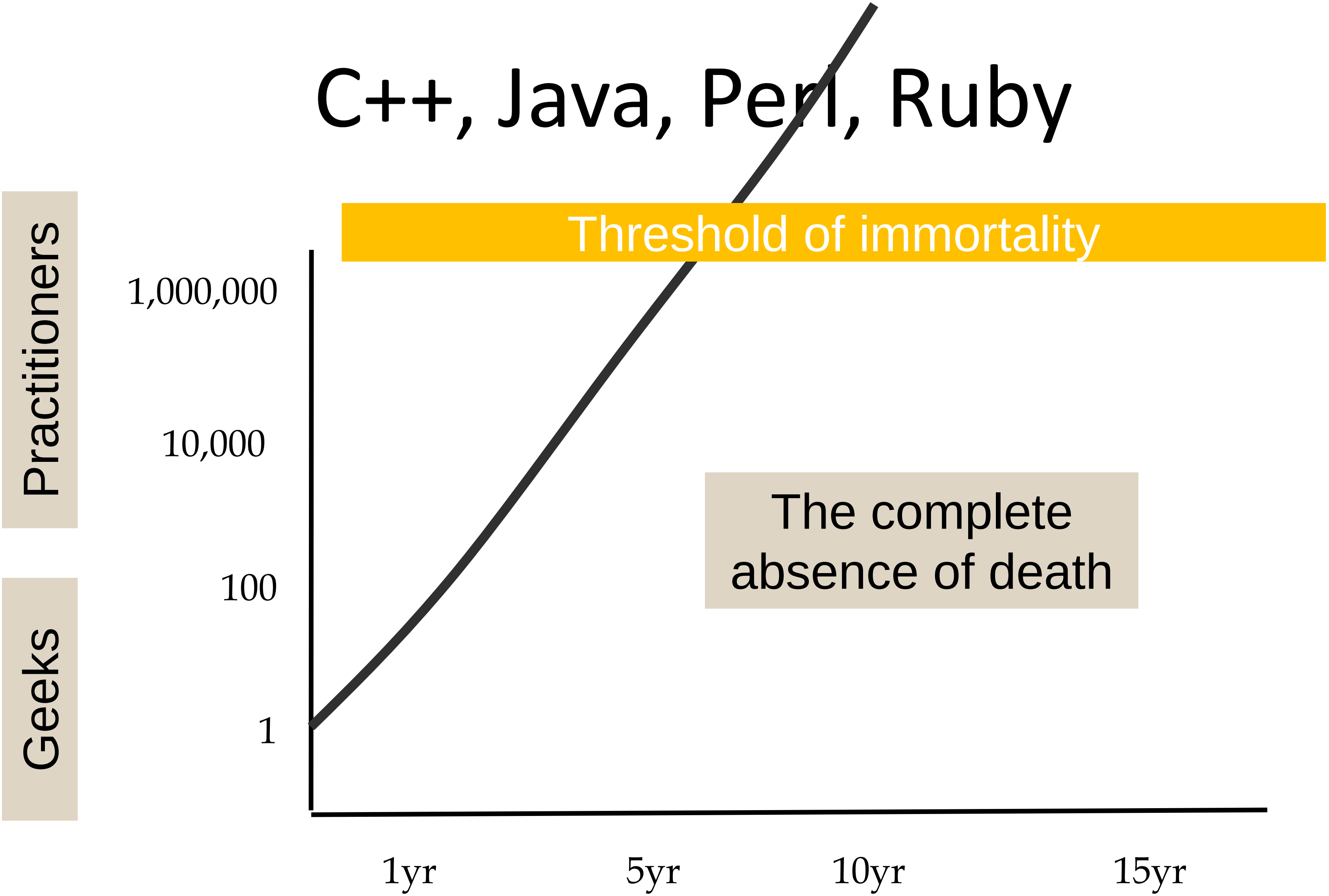
Most Research Languages



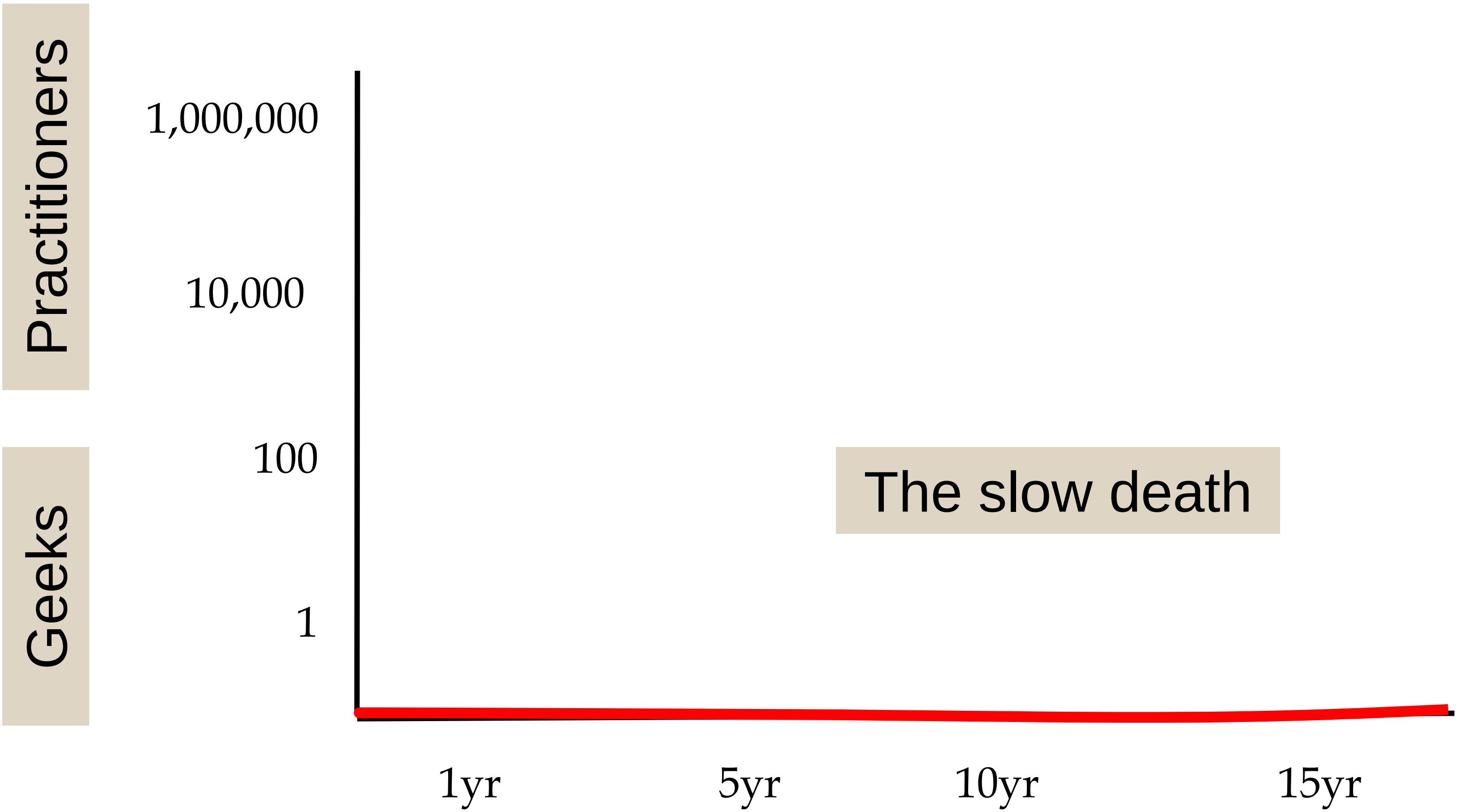
Successful Research Languages



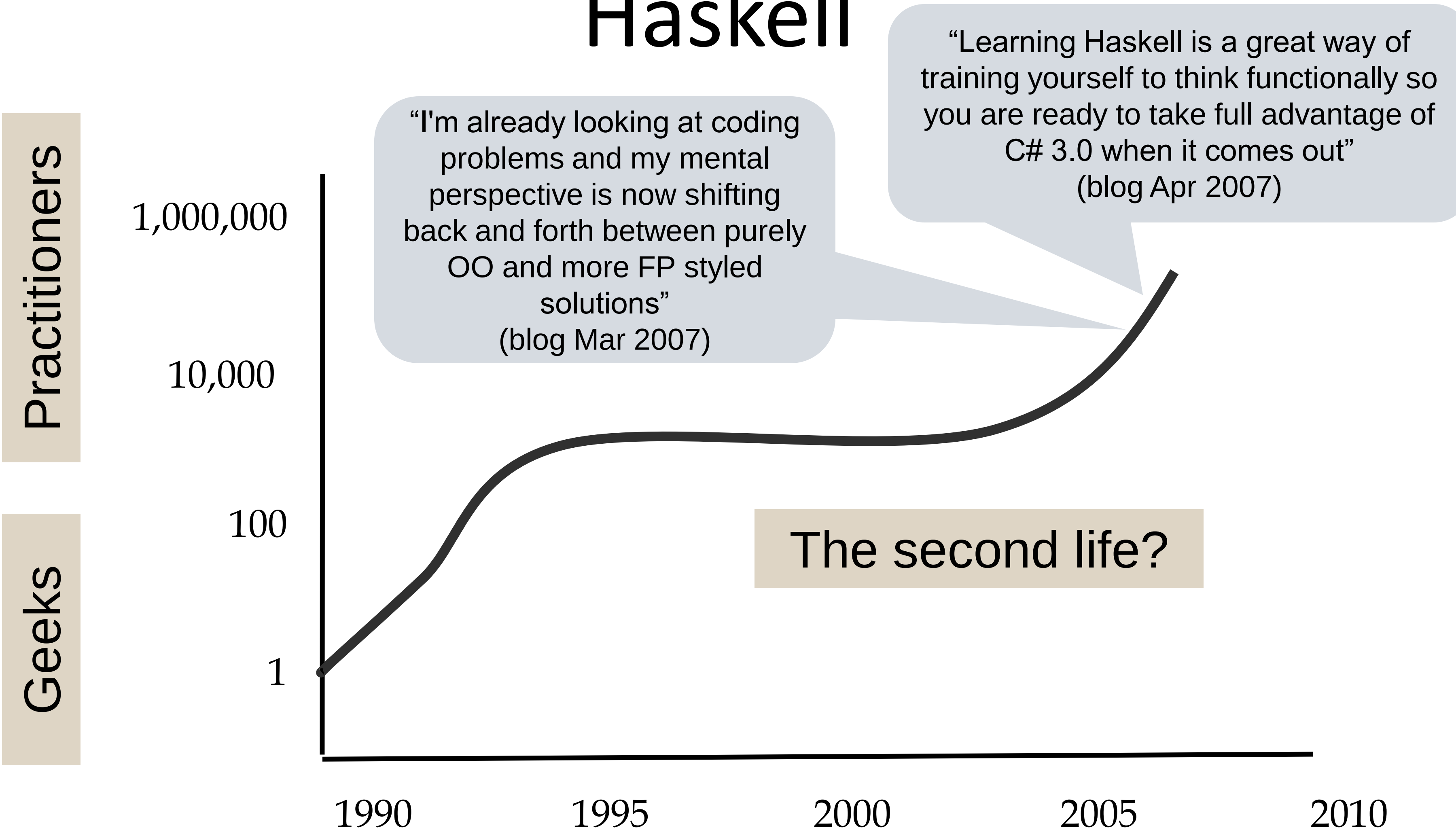
C++, Java, Perl, Ruby



Committee languages



Haskell



Practitioners
Geeks
1,000,000
10,000
100
1
1990
1995
2000
2005
2010
“I'm already looking at coding problems and my mental perspective is now shifting back and forth between purely OO and more FP styled solutions” (blog Mar 2007)
“Learning Haskell is a great way of training yourself to think functionally so you are ready to take full advantage of C# 3.0 when it comes out” (blog Apr 2007)
The second life?

Haskell   Downloads   Community   Documentation   News

Haskell Platform

Haskell with batteries included

A multi-OS distribution

designed to get you up and running quickly, making it easy to focus on using Haskell. You get:

- the Glasgow Haskell Compiler
- the Cabal build system
- the Stack tool for developing projects
- support for profiling and code coverage analysis
- 35 core & widely-used packages

Prior releases of the Platform are also available.

Let's get started

Note: the stack tool has been evolving relatively rapidly. Users who wish to ensure they are running the latest version may want to consider running "stack update" and ensuring the proper path for stack-installed binaries is in their environment.

You appear to be using **Microsoft Windows**. See below for other operating systems.

https://www.haskell.org/platform/

Function Types in Haskell

In Haskell, $f :: A \to B$ 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 $A \to B$ can throw an exception or have other effects, but not in Haskell

Higher-Order Functions

- 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
```

Overview by Type

- Booleans

```
True, False :: Bool
if …  then … else …        --types must match
```

- Integers

```
0, 1, 2, … :: Integer
+, * , …   :: Integer  -> Integer -> Integer
```

- Strings

```
"Ron Weasley"
```

- Floats

```
1.0, 2, 3.14159, …  --type classes to disambiguate
```

Haskell Libraries

Simple Compound Types

- Tuples

```
(4, 5, “Griffendor”) :: (Integer, Integer, String)
```

- Lists

```
[] :: [a]                            -- polymorphic type
```

```
1 : [2, 3, 4] :: [Integer]    -- infix cons notation
```

- Records

```
data Person = Person {firstName :: String,
                      lastName  :: String}
hg = Person { firstName = “Hermione”,
              lastName  = “Granger”}
```

Patterns and Declarations

- Patterns can be used in place of variables

  <pat> ::= <var> | <tuple> | <cons> | <record> …
- Value declarations
  - General form: <pat> = <exp>
  - Examples

```
myTuple = ("Flitwick", "Snape")
(x,y)  = myTuple
myList = [1, 2, 3, 4]
z:zs  = myList
```

  - Local declarations

```
let (x,y) = (2, "Snape") in x * 4
```

Functions and Pattern Matching

- Anonymous function

```
\x -> x+1     --like Lisp lambda, function (…) in JS
```

- Function declaration form

  <name> <pat$_1$> = <exp$_1$>

  <name> <pat$_2$> = <exp$_2$> …

  <name> <pat$_n$> = <exp$_n$> …

- Examples

```
f (x,y) = x+y     --argument must match pattern (x,y)
length [] = 0
length (x:s) = 1 + length(s)
```

Map Function on Lists

- Apply function to every element of list

```
map f [] = []
map f (x:xs) = f x : map f xs
```

```
map (\x -> x+1) [1,2,3]                [2,3,4]
```

- Compare to Lisp

```
(define map
    (lambda (f  xs)
      (if   (eq? xs ())   ()
         (cons (f   (car xs))   (map f   (cdr xs)))
   )))
```

More Functions on Lists

- Append lists

```
append ([], ys) = ys
append (x:xs, ys) = x : append (xs, ys)
```

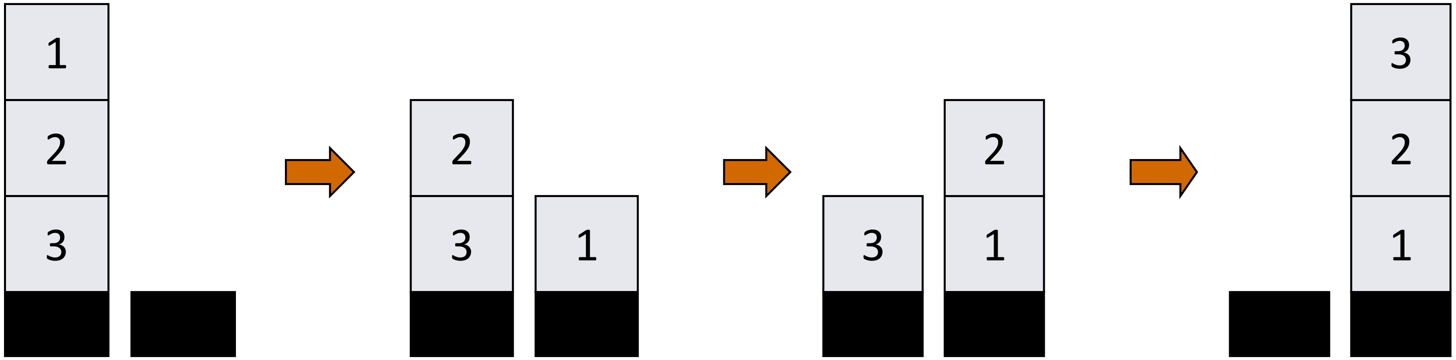
- Reverse a list

```
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
```

- Questions
  - How efficient is reverse?
  - Can it be done with only one pass through list?

More Efficient Reverse

```
reverse xs =
    let rev ( [], accum ) = accum
         rev ( y:ys, accum ) = rev ( ys, y:accum )
    in rev ( xs, [] )
```



List Comprehensions

- Notation for constructing new lists from old:

```
myData = [1,2,3,4,5,6,7]

twiceData = [2 * x | x <- myData]
-- [2,4,6,8,10,12,14]

twiceEvenData = [2 * x| x <- myData, x `mod` 2 == 0]
-- [4,8,12]
```

- Similar to “set comprehension”

$$\{ x \mid x \in Odd \wedge x > 6 \}$$

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> | <contructor> <type>
```

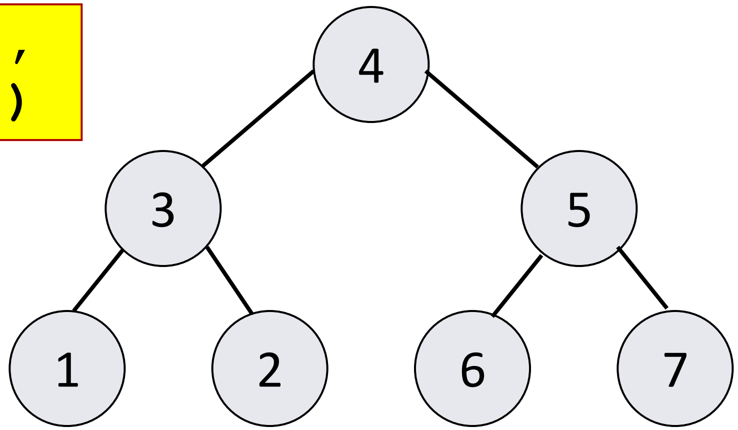
– Type name and constructors must be Capitalized.

Datatypes and Pattern Matching

- Recursively defined data structure

```
data Tree = Leaf Int | Node (Int, Tree, Tree)
```

```
Node(4, Node(3, Leaf 1, Leaf 2),
        Node(5, Leaf 6, Leaf 7))
```



- Recursive function

```
sum (Leaf n) = n
sum (Node(n,t1,t2)) = n + sum(t1) + sum(t2)
```

Example: Evaluating Expressions

- Define datatype of expressions

```
data Exp = Var Int | Const Int | Plus (Exp, Exp)
```

write (x+3)+ y as Plus(Plus(Var 1, Const 3), Var 2)

- Evaluation function

```
ev(Var n) = Var n
ev(Const n ) = Const n
ev(Plus(e1,e2)) =  …
```

- Examples

```
ev(Plus(Const 3, Const 2))          ⇒          Const 5
```

```
ev(Plus(Var 1, Plus(Const 2, Const 3)))   ⇒
                                   Plus(Var 1, Const 5)
```

Case Expression

- Datatype

```
data Exp = Var Int | Const Int | Plus (Exp, Exp)
```

- Case expression

```
case e of
     Var n ->  …
     Const n -> …
     Plus(e1,e2) -> …
```

Indentation matters in case statements in Haskell.

Evaluation by Cases

```
data Exp = Var Int | Const Int | Plus (Exp, Exp)

ev ( Var n) = Var n
ev ( Const n ) = Const n
ev ( Plus ( e1,e2 ) ) =
    case ev e1 of
      Var n -> Plus( Var n, ev e2)
      Const n -> case ev e2 of
                   Var m -> Plus( Const n, Var m)
                   Const m -> Const (n+m)
                   Plus(e3,e4) -> Plus ( Const n,
                                         Plus ( e3, e4 ))
      Plus(e3, e4) -> Plus( Plus ( e3, e4 ), ev e2)
```

Laziness

- Haskell is a **lazy** language
- Functions and data constructors don't evaluate their arguments until they need them

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

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

Short-circuiting "or"

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

Using Laziness

```haskell
isSubString :: String -> String -> Bool
x `isSubString` s = or [ x `isPrefixOf` t
                       | t <- suffixes s ]
```

```haskell
suffixes:: String -> [String]
-- All suffixes of s
suffixes[]      = [[]]
suffixes(x:xs) = (x:xs) : suffixes xs
```

type String = [Char]

```haskell
or :: [Bool] -> Bool
-- (or bs) returns True if any of the bs is True
or []     = False
or (b:bs) = b || or bs
```

A Lazy Paradigm

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

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

A gigantic (perhaps infinite) tree of possible moves

Core Haskell

- Basic Types
  - Unit
  - Booleans
  - Integers
  - Strings
  - Reals
  - Tuples
  - Lists
  - Records
- Patterns
- Declarations
- Functions
- Polymorphism
- Type declarations
- Type Classes
- Monads
- Exceptions

Running Haskell

- Look for instructions on web site
  - Or use ghci from corn or myth
- Or, download: http://haskell.org/ghc
- Interactive:
  - ghci intro.hs
- Compiled:
  - ghc –make HaskellIntro.hs

Demo ghci

Testing

- It's good to write tests as you write code
- E.g. **`reverse`** undoes itself, etc.

```
reverse xs =
    let rev ( [], z ) = z
        rev ( y:ys, z ) = rev( ys, y:z )
    in rev( xs, [] )

-- Write properties in Haskell
type TS = [Int]      -- Test at this type

prop_RevRev :: TS -> Bool
prop_RevRev ls = reverse (reverse ls) == ls
```

Test Interactively

Test.QuickCheck is simply a Haskell library (not a "tool")

```
bash$ ghci intro.hs
Prelude> :m +Test.QuickCheck

Prelude Test.QuickCheck> quickCheck prop_RevRev
+++ OK, passed 100 tests
```

...with a strange-looking type

```
Prelude Test.QuickCheck> :t quickCheck
quickCheck :: Testable prop => prop -> IO ()
```

Demo QuickCheck

QuickCheck

- Generate random input based on type
  - Generators for values of type a has type Gen a
  - Have generators for many types
- Conditional properties
  - Have form <condition> ==> <property>
  - Example:

```
ordered xs = and (zipWith (<=) xs (drop 1 xs))
insert x xs = takeWhile (<x) xs++[x]++dropWhile (<x) xs
prop_Insert x xs =
      ordered xs ==> ordered (insert x xs)
where types = x::Int
```

QuickCheck

- QuickCheck output
  - When property succeeds:

    quickCheck prop_RevRev OK, passed 100 tests.
  - When a property fails, QuickCheck displays a counter-example.

    prop_RevId xs = reverse xs == xs where types = xs::[Int]

    quickCheck prop_RevId

    Falsifiable, after 1 tests: [-3,15]
- Conditional testing
  - Discards test cases which do not satisfy the condition.
  - Test case generation continues until
    - 100 cases which do satisfy the condition have been found, or
    - until an overall limit on the number of test cases is reached (to avoid looping if the condition never holds).

See : http://www.haskell.org/haskellwiki/Introduction_to_QuickCheck

Things to Notice

No side effects. At all.

```
reverse:: [w] -> [w]
```

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

```
prop_RevRev l = reverse(reverse l) == l
```

- A variable ‘l’ stands for an immutable value, not for a location whose value can change.
- Laziness forces this purity.

Things to Notice

- Purity makes the interface explicit.

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

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

```
void reverse( list l )         /* C */
```

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

Things to Notice

- Pure functions are easy to test.

```
prop_RevRev l = reverse(reverse l) == l
```

- In an imperative or OO language, you have to
  - set up the state of the object and the external state it reads or writes
  - make the call
  - inspect the state of the object and the external state
  - perhaps copy part of the object or global state, so that you can use it in the post condition

Things to Notice

Types are everywhere.

```
reverse:: [w] -> [w]
```

- In Haskell, types express high-level design, in the same way that UML diagrams do, with the advantage that the type signatures are machine-checked.
- Types are (almost always) optional: type inference fills them in if you leave them out.

More Info: haskell.org

- The Haskell wikibook
  - http://en.wikibooks.org/wiki/Haskell
- All the Haskell bloggers, sorted by topic
  - http://haskell.org/haskellwiki/Blog_articles
- Collected research papers about Haskell
  - http://haskell.org/haskellwiki/Research_papers
- Wiki articles, by category
  - http://haskell.org/haskellwiki/Category:Haskell
- Books and tutorials
  - http://haskell.org/haskellwiki/Books_and_tutorials
  - http://book.realworldhaskell.org

A list of functions that make up the Prelude package in Haskell

- http://www.haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html