Haskell
Functional Programming

http://igm.univ-mlv.fr/~vialette/?section=teaching

Stéphane Vialette

LIGM, Université Paris-Est Marne-la-Vallée

September 19, 2019

Everybody's talking about functional programming

Erlang

Erlang (`https://www.erlang.org/`) is a general-purpose, concurrent, functional programming language, as well as a garbage-collected runtime system.

Everybody's talking about functional programming

Elixir

Elixir (`https://elixir-lang.org/`) is a functional, concurrent, general-purpose programming language that runs on the Erlang virtual machine (BEAM).

Everybody's talking about functional programming

F#

F# (`http://fsharp.org/` is a strongly typed, multi-paradigm programming language that encompasses functional, imperative, and object-oriented programming methods. It is being developed at Microsoft Developer Division and is being distributed as a fully supported language in the .NET framework.

Everybody's talking about functional programming

Ocaml

Ocaml (`http://ocaml.org/` originally named Objective Caml, is the main implementation of the programming language Caml. OCaml's toolset includes an interactive top-level interpreter, a bytecode compiler, a reversible debugger, a package manager (OPAM), and an optimizing native code compiler.

Everybody's talking about functional programming

Lisp

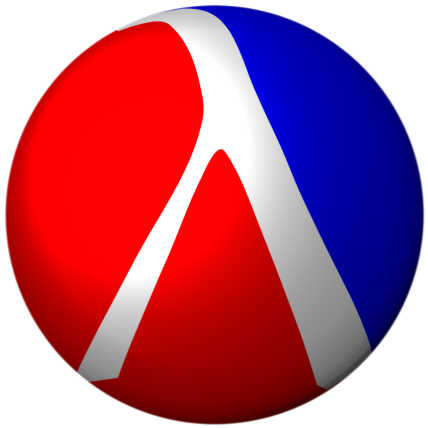
Lisp (historically, LISP) is a family of computer programming languages with a long history and a distinctive, fully parenthesized prefix notation. Originally specified in 1958, Lisp is the second-oldest high-level programming language in widespread use today. (Only Fortran is older, by one year.)

Everybody's talking about functional programming

Clojure

Clojure (`https://clojure.org/`) is a dialect of the Lisp programming language. Clojure is a general-purpose programming language with an emphasis on functional programming. It runs on the Java virtual machine and the Common Language Runtime.

Everybody's talking about functional programming



Racket

Racket (`http://racket-lang.org/`), formerly PLT Scheme, is a general purpose, multi-paradigm programming language in the Lisp-Scheme family. One of its design goals is to serve as a platform for language creation, design, and implementation

Everybody's talking about functional programming

Elm

Elm (`http://elm-lang.org/`) is a domain-specific programming language for declaratively creating web browser-based graphical user interfaces. Elm is purely functional, and is developed with emphasis on usability, performance, and robustness.

Everybody's talking about functional programming

Scala

Scala (`https://www.scala-lang.org/`) is a general-purpose programming language providing support for functional programming and a strong static type system. Designed to be concise, many of Scala's design decisions aimed to address criticisms of Java.

Everybody's talking about functional programming

Haskell

Haskell (`https://www.haskell.org/`) is a standardized, general-purpose purely functional programming language, with non-strict semantics and strong static typing. The latest standard of Haskell is Haskell 2010. As of May 2016, a group is working on the next version, Haskell 2020.

Everybody's talking about functional programming

Why functional programming matters:

1. FP offers concurrency/parallelism with tears.
2. FP has succint, concise and understandable syntax.
3. FP offers a different programming perspective.
4. FP is becoming more accessible.

FP is fun!

FP offers concurrency/parallelism with tears

Moore's law has held up for years but it is starting to reach its limits due to physical constraints. Chips aren't getting much faster but multi-core, hyper-threaded, etc machines are becoming far more commonplace.

If you want to take advantages of your machine's full processing power, you can no longer rely on continuous chip advances alone. You need to really start thinking about concurrency, parallelism and multi-threaded if you wish to better performance and use all available CPUs.

Of course, these are not easily implemented concepts so coders need to start considering ways (like FP!) to make these approaches more available and practical.

FP has succint, concise and understandable syntax

The abstract nature of FP leads to considerably simpler programs. It also supports a number of powerful new ways to structure and reason about programs.

`x = x+1;` We understand this syntax because we often resort to telling the computer what to do, but this equation really makes no sense at all!

Ask, don't tell.

FP offers a different programming perspective

For me, the most important thing about FP isn't that functional languages have some particular useful language features, but that it allows to think differently and simply about problems that you encouter when designing and writing applications. This is much more important than understanding any new technology or a programming language.

Tomas Petricek
`http://tomasp.net/blog/`

Quicksort

Pseuso-code

1. Pick an element, called a pivot, from the array.
2. Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the *partition operation*.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

Quicksort

Erlang

```
-module(quicksort).

-export([qsort/1]).

qsort([]) -> [];
qsort([X|Xs]) ->
  qsort([Y || Y <- Xs, Y < X]) ++ [X] ++ qsort([Y || Y <- Xs, Y >= X]).
```

Quicksort

Elixir

```
defmodule Sort do
  def qsort([]), do: []
  def qsort([h | t]) do
    {lesser, greater} = Enum.split_with(t, &(&1 < h))
    qsort(lesser) ++ [h] ++ qsort(greater)
  end
end
```

Quicksort

F#

```
let rec qsort = function
    hd :: tl ->
        let less, greater = List.partition ((>=) hd) tl
        List.concat [qsort less; [hd]; qsort greater]
    | _ -> []
```

Quicksort

OCaml

```
let rec quicksort gt = function
  | [] -> []
  | x::xs ->
      let ys, zs = List.partition (gt x) xs in
      (quicksort gt ys) @ (x :: (quicksort gt zs))
```

Quicksort

Lisp

```
(defun quicksort (list &aux (pivot (car list)) )
  (if (cdr list)
      (nconc (quicksort (remove-if-not #'(lambda (x) (< x pivot)) list)
             (remove-if-not #'(lambda (x) (= x pivot)) list)
             (quicksort (remove-if-not #'(lambda (x) (> x pivot)) list)
      list))
```

Quicksort

Clojure

```
(defn qsort [L]
  (if (empty? L)
      '()
      (let [[pivot & L2] L]
           (lazy-cat (qsort (for [y L2 :when (<  y pivot)] y))
                     (list pivot)
                     (qsort (for [y L2 :when (>= y pivot)] y))))))
```

Quicksort

Racket

```
#lang racket
(define (quicksort < l)
  (match l
    ['() '()]
    [(cons x xs)
     (let-values ([(xs-gte xs-lt) (partition (curry < x) xs)])
       (append (quicksort < xs-lt)
               (list x)
               (quicksort < xs-gte)))]))
```

Quicksort

Scala

```scala
def sort(xs: List[Int]): List[Int] = xs match {
  case Nil => Nil
  case head :: tail =>
    val (less, notLess) = tail.partition(_ < head)
    sort(less) ++ (head :: sort(notLess))  // Sort each half
}
```

Quicksort

Haskell

```
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y < x] ++
               [x]                        ++
               qsort [y | y <- xs, y >= x]
```

Quicksort

Haskell

```haskell
import Data.List (partition)

qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort ys ++ x : qsort zs
  where
    (ys, zs) = partition (< x) xs
```

FP is becoming more accessible

More language options.

Tooling, IDEs.

Supports.

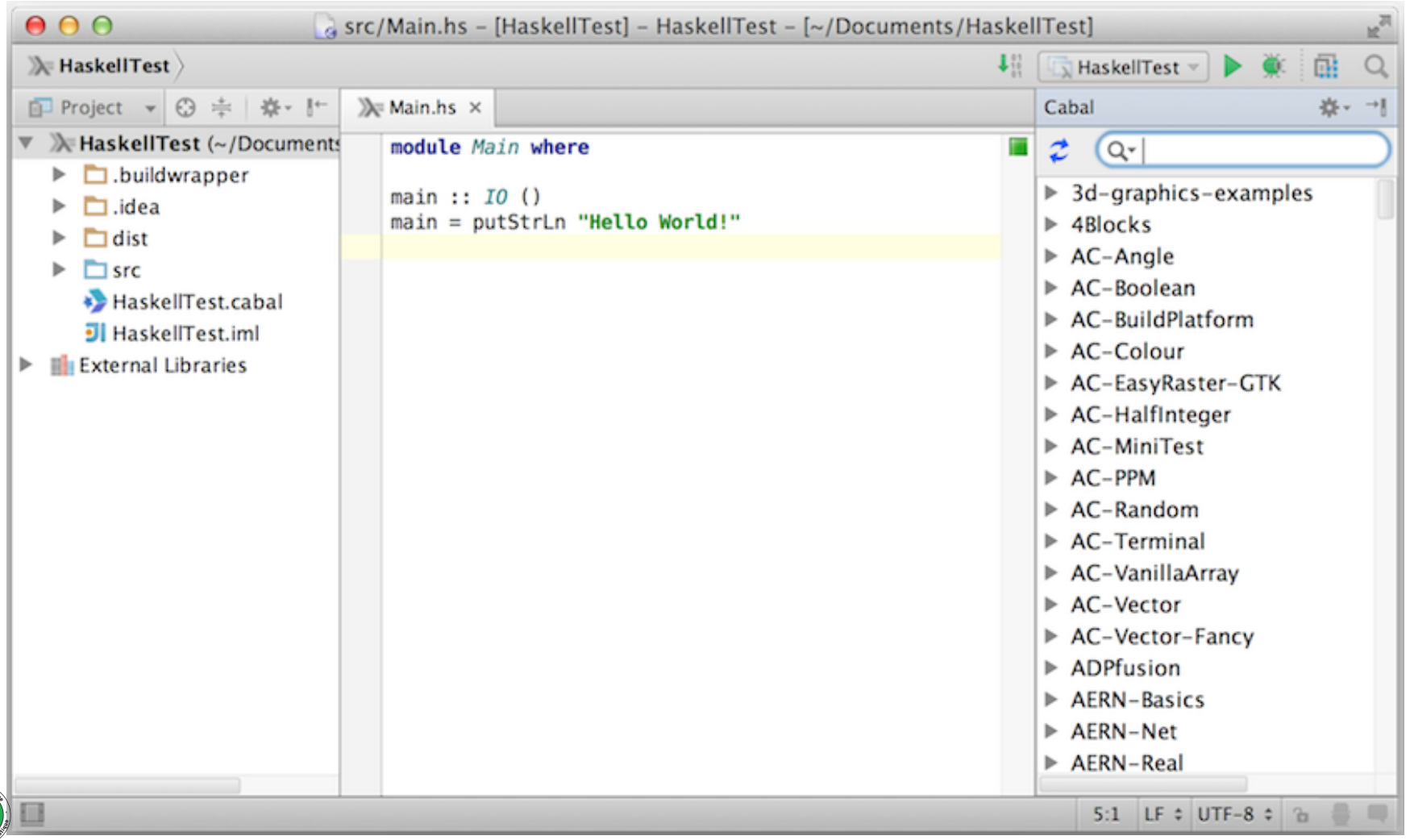
Books.

Blogs, podcasts and screencasts.

Conferences and user groups.

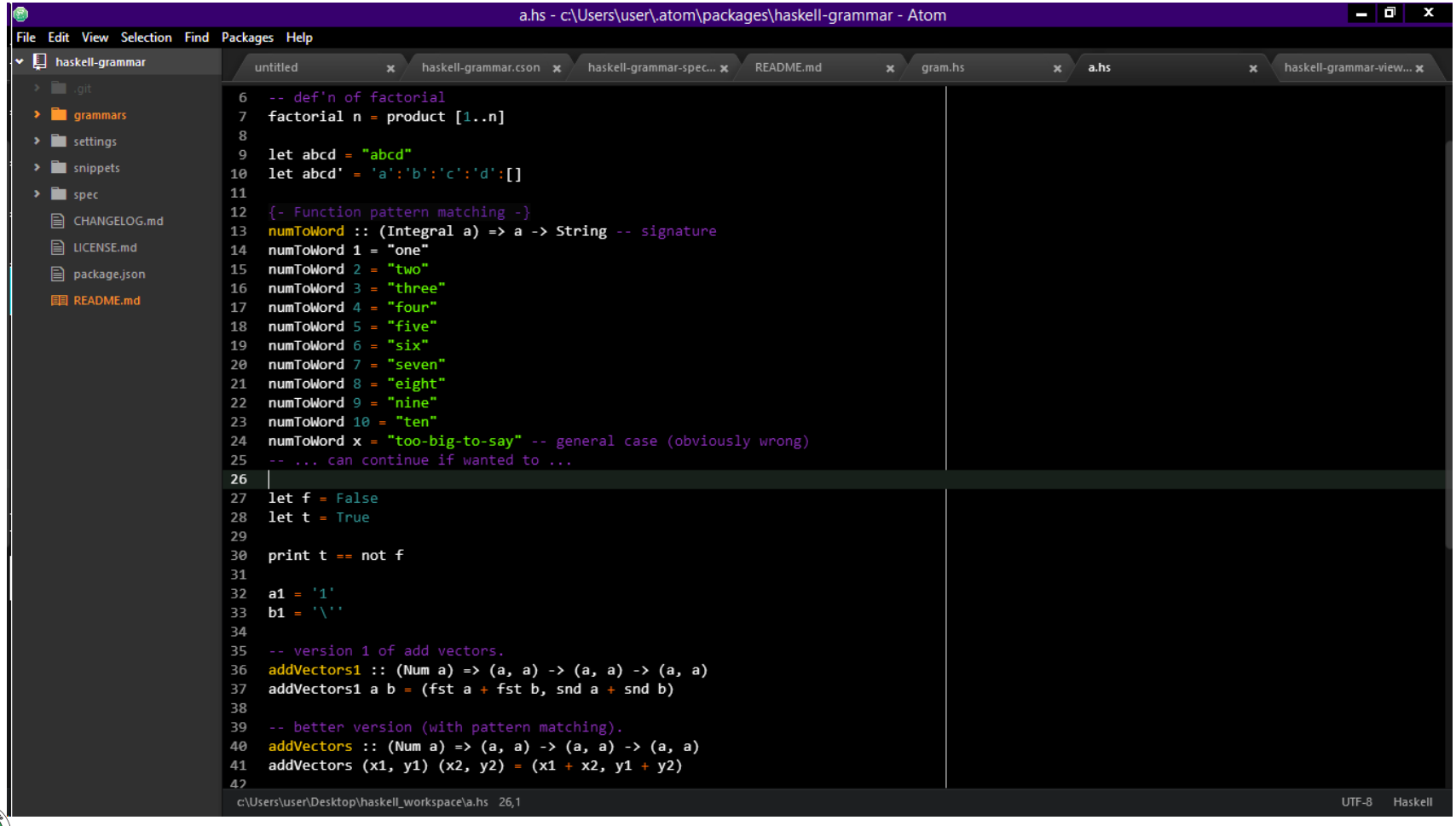
Haskell is becoming more accessible

IntelliJ IDEA



Haskell is becoming more accessible

Atom



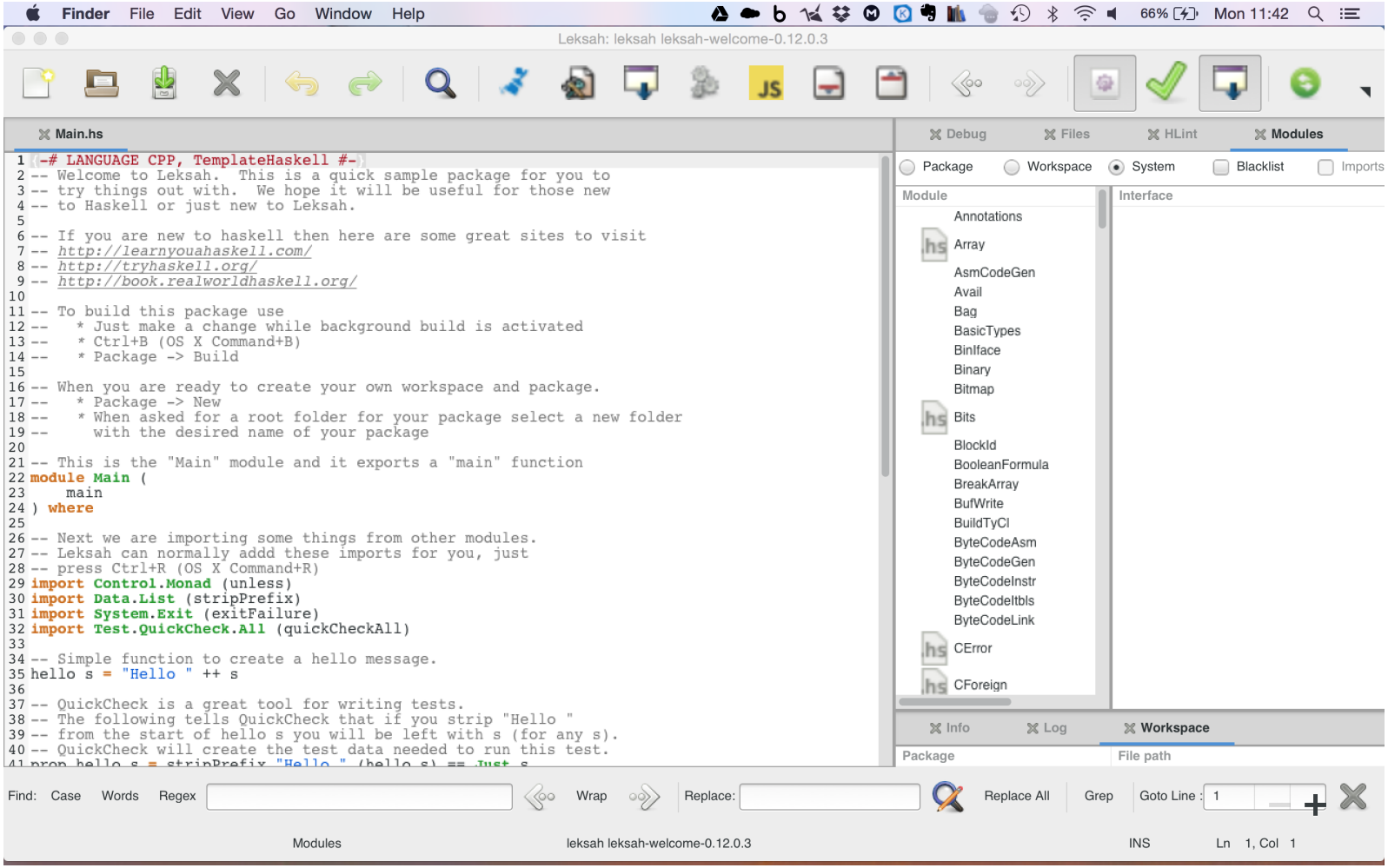
Haskell is becoming more accessible

Emacs

```
File Edit Options Buffers Tools Haskell Help

import qualified Data.Hashable as Hash
import Data.Time

-- | Task priority
data Priority
    = L -- ^ low priority
    | M -- ^ medium priority
    | H -- ^ high priority
      deriving (Eq, Ord, Show, Read, Bounded, Enum)

-- | A single task
data Task = Task {
      tId      :: Int          , -- ^ task id, might change after display
      tHashID  :: Int          , -- ^ unique hash, never changes
      tDesc    :: String       , -- ^ task description
      tCreated :: UTCTime      , -- ^ creation time (October 23, 4004 BC?)
      tDue     :: Maybe UTCTime  , -- ^ task due time
      tPri     :: Maybe Priority , -- ^ task priority
      tProj    :: Maybe String     -- ^ assocaiated project
    } deriving (Show, Read)

-- | Constructs a new Task. tHashID of a task is calculated
-- automatically based on the description and creation time
createTask :: Int            -- ^ id of a new task
           -> String         -- ^ task description
           -> UTCTime        -- ^ creation time
           -> Maybe UTCTime  -- ^ task due time
           -> Maybe Priority -- ^ task priority
           -> Maybe String   -- ^ project to assign a task to
           -> Task           -- ^ the new task
createTask taskId taskDesc taskCreated taskDue taskPri taskProj =
    Task taskId taskHashID taskDesc taskCreated taskDue taskPri taskProj
    where taskHashID = hashTask taskCreated taskDesc

-- | Create unique hash of a task based on creation time and description
-UU-:----F1  Tasks.hs        6% L15   Git-master  (Haskell WS Ind Doc)----------------------------------------------------------
data [context =>] simpletype = constrs [deriving]
```

Haskell is becoming more accessible

Leksah



Key Haskell concepts

High order functions, map, filter reduce (*i.e.*, fold).

Recursion.

Pattern matching.

Currying.

Lazy/eager evaluation.

Strict/non-strict semantics.

Type inference.

Monads.

Continuations.

Closures.

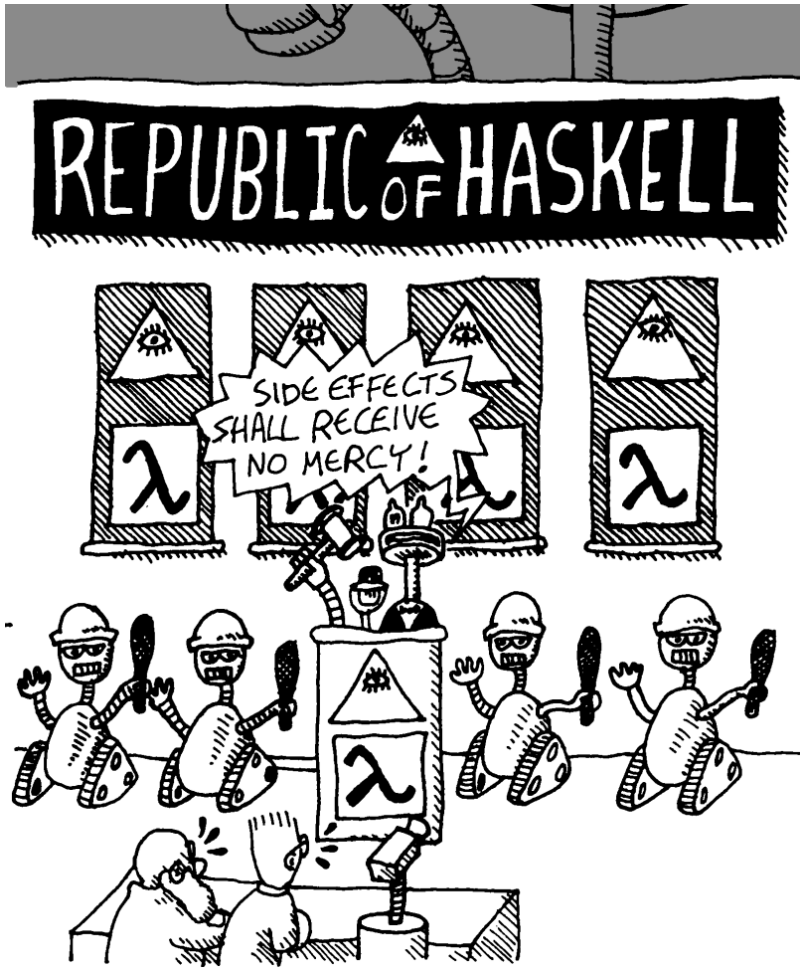
Haskell

Haskell

Haskell is a standardized, general-purpose purely functional programming language, with non-strict semantics and strong static typing.

It is named after logician Haskell Curry.

Haskell


REPUBLIC OF HASKELL
SIDE EFFECTS SHALL RECEIVE NO MERCY!
λ
λ
λ
λ
λ

What can Haskell offer the programmer?

Purity: Unlike some other functional programming languages Haskell is pure. It doesn't allow any side-effects. This is probably the most important feature of Haskell.

Laziness: Haskell is lazy (technically speaking, it's "non-strict"). This means that nothing is evaluated until it has to be evaluated.

Strong typing: Haskell is strongly typed, this means just what it sounds like. It's impossible to inadvertently convert a `Double` to an `Int`, or follow a null pointer. Unlike other strongly typed languages types in Haskell are automatically inferred.

Elegance: Another property of Haskell that is very important to the programmer, even though it doesn't mean as much in terms of stability or performance, is the elegance of Haskell. To put it simply: stuff just works like you'd expect it to.

Haskell and bugs

Pure. There are no side effects.

Strongly typed. There can be no dubious use of types. And No Core Dumps!

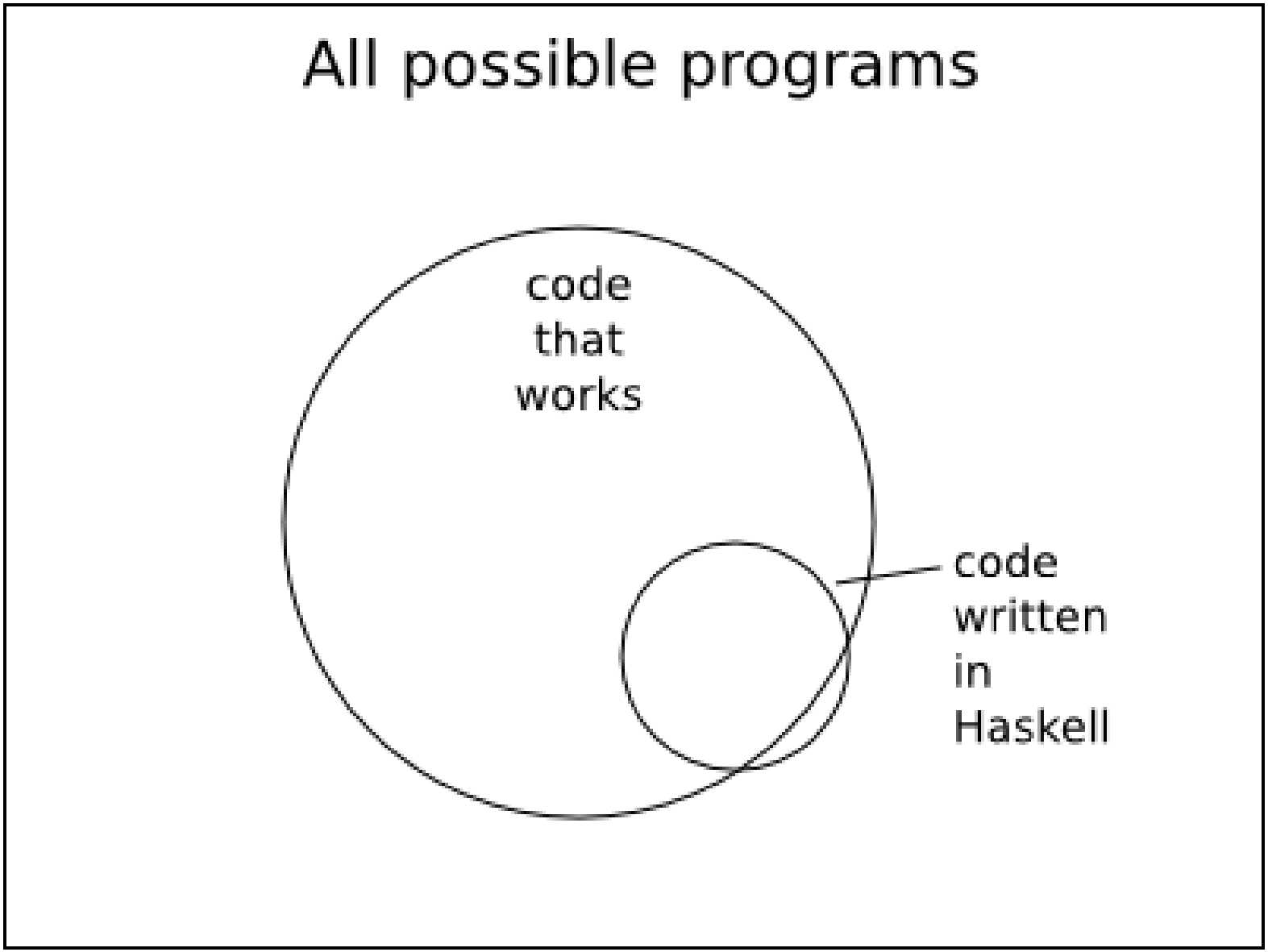
Concise. Programs are shorter which make it easier to look at a function and "take it all in" at once, convincing yourself that it's correct.

High level. Haskell programs most often reads out almost exactly like the algorithm description. Which makes it easier to verify that the function does what the algorithm states.

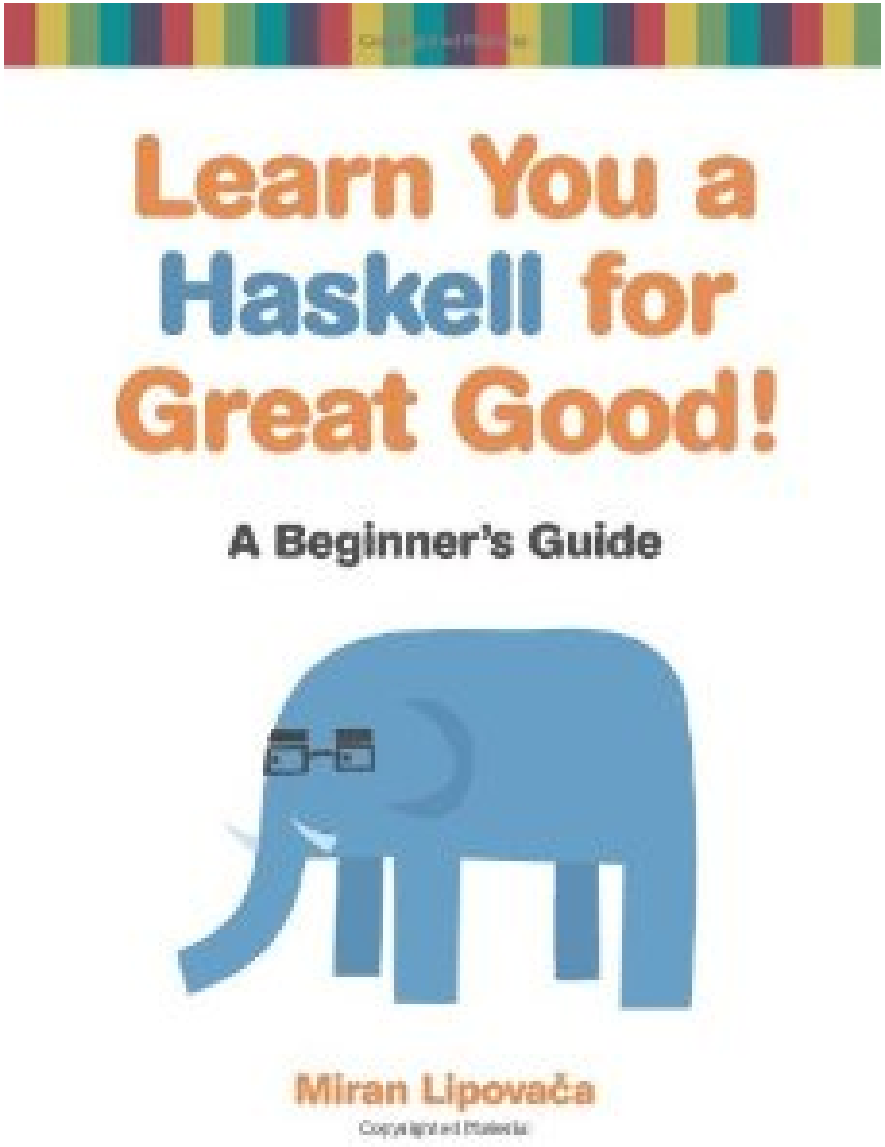
Memory managed. There's no worrying about dangling pointers, the Garbage Collector takes care of all that.

Modular. Haskell offers stronger and more "glue" to compose your program from already developed modules.

So what !?


All possible programs
code
that
works
code
written
in
Haskell

Reference book



Hello, World!

```haskell
module Main where

main :: IO ()
main = putStrLn "Hello, World!"
```

Hello, World!: Compile to native code

```
barbalala: ghc -o Hello Hello.hs
[1 of 1] Compiling Main                ( Hello.hs, Hello.o )
Linking Hello ...
barbalala: ./Hello
Hello, World!
barbalala:
```

Hello, World!: Interpreter

```
barbalala: ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/
:? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :load "Hello"
[1 of 1] Compiling Main ( Hello.hs, interpreted )
Ok, modules loaded: Main.
*Main> main
Hello, World!
*Main>
```

Quicksort in Haskell

```
quicksort :: Ord a => [a] -> [a]
quicksort []     = []
quicksort (p:xs) = quicksort lesser ++
                   [p]              ++
                   quicksort greater
    where
        lesser  = filter (< p)  xs
        greater = filter (>= p) xs
```

The Fibonacci sequence

```haskell
fib :: (Eq a, Num a, Num b) => a -> b
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

or

```haskell
fib :: (Integral b, Integral a) => a -> b
fib n = round $ phi ** fromIntegral n / sq5
  where
    sq5 = sqrt 5 :: Double
    phi = (1 + sq5) / 2
```

or

```haskell
fibs :: Num a => [a]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

or ...

Implementations

The Glasgow Haskell Compiler (GHC) compiles to native code on a number of different architectures. GHC has become the de facto standard Haskell dialect. There are libraries (e.g. bindings to OpenGL) that will work only with GHC. GHC is also distributed along with the Haskell platform.

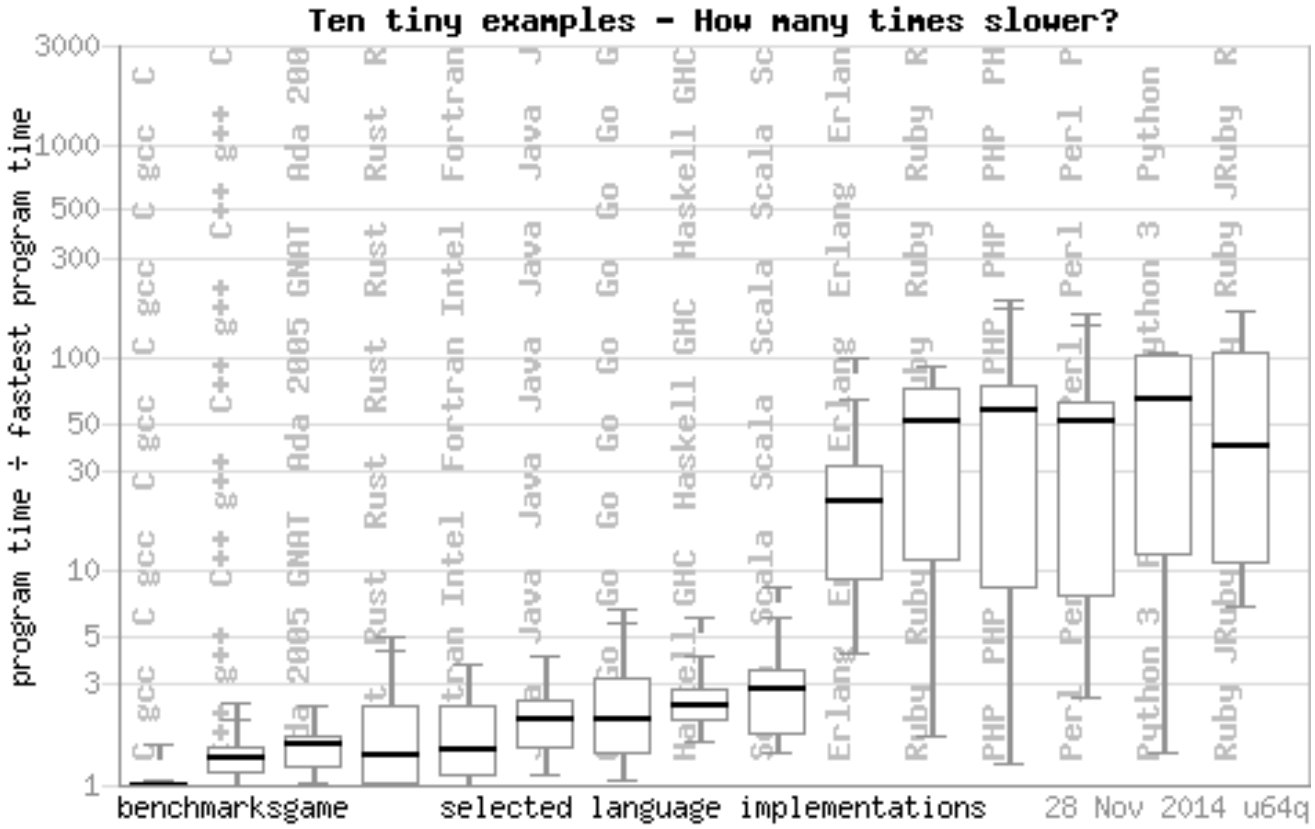
The Utrecht Haskell Compiler (UHC) is a Haskell implementation from Utrecht University. UHC supports almost all Haskell 98 features plus many experimental extensions.

Jhc is a Haskell compiler written by John Meacham emphasising speed and efficiency of generated programs as well as exploration of new program transformations.

Ajhc is a fork of Jhc.

The speed of Haskell

For most applications the difference in speed between C++ and Haskell is so small that it's utterly irrelevant



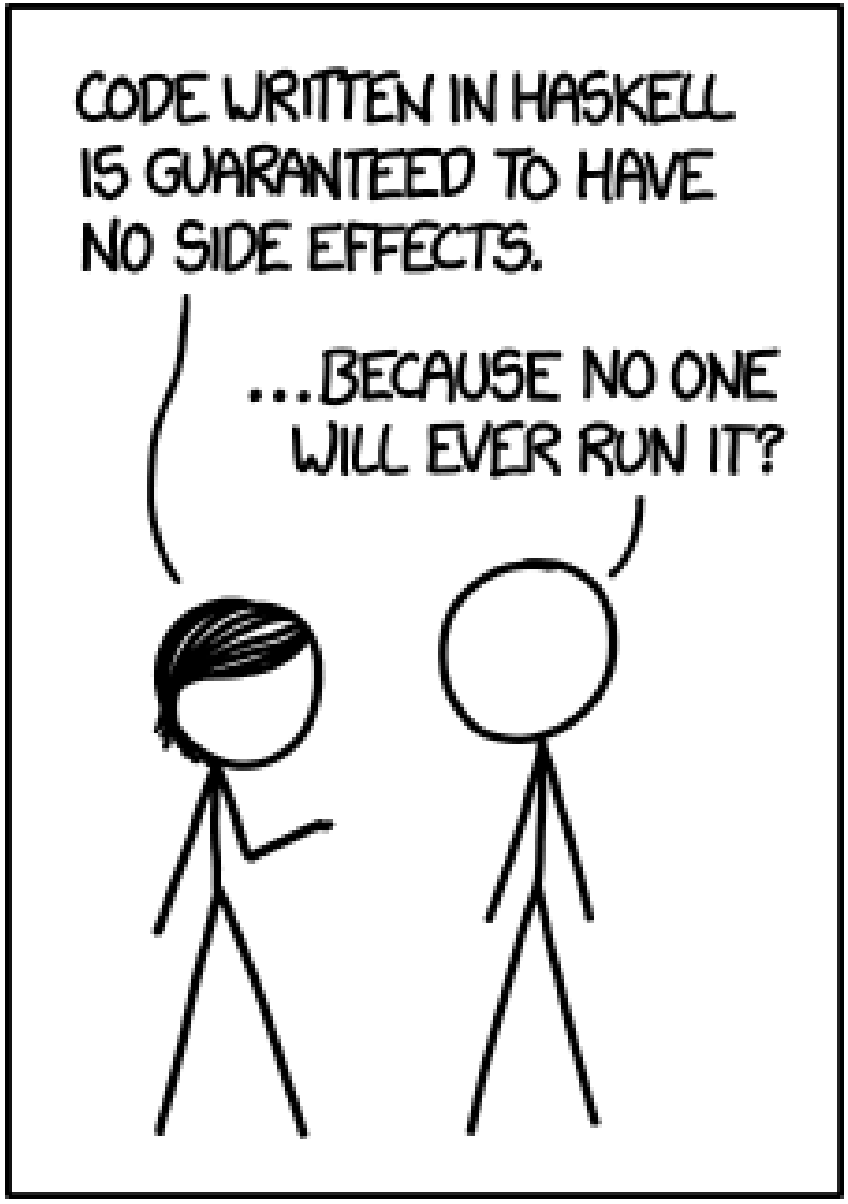
The speed of Haskell

There's an old rule in computer programming called the "*80/20 rule*". It states that 80% of the time is spent in 20% of the code. The consequence of this is that any given function in your system will likely be of minimal importance when it comes to optimizations for speed. There may be only a handful of functions important enough to optimize.

Remember that algorithmic optimization can give much better results than code optimization.

Last but not least, Haskell offers substantially increased programmer productivity (Ericsson measured an improvement factor of between 9 and 25 using Erlang, a functional programming language similar to Haskell, in one set of experiments on telephony software.)

Haskell in Industry



Why is Haskell not used in the software industry?

even though it is a popular functional programming language!

- Integration with the companies' existing codebase.
- There are not enough people with Haskell experience.
- Colleges and universities do little to popularize Haskell.
- Clojure and Scala are not purely functional but have done a lot to popularize functional programming.

  Using these languages, the management and programmers can claim to be trained in functional programming and yet know of nothing more than map, reduce and fold.

Haskell in Industry

- https://wiki.haskell.org/Haskell_in_industry
- http://industry.haskell.org/
- https://www.fpcomplete.com/hubfs/Haskell-User-Survey-Results.pdf
- https://github.com/erkmos/haskell-companies
- http://www.cs.tut.fi/~bitti/functional-seminar/RoleofHaskellintheSoftwareIndustry.pdf

Functional programming languages