A good programming language is a *Functional* one

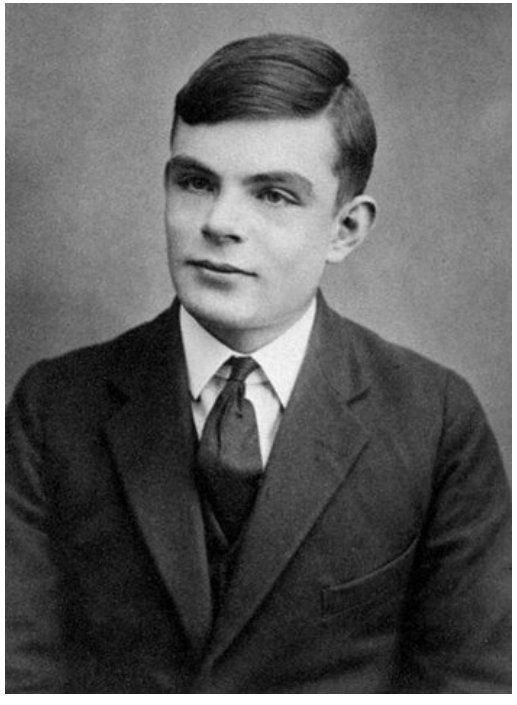
Artin Ghasivand

November 24, 2024

People

Alonzo Church

Kurt Gödel



Alan Turing

- Alonzo Church: Untyped Lambda-Calculus, 1935
- Kurt Gödel: General Recursive functions, 1935 (Stephen Kleene)
- Alan Turing: Turing machines, 1936

Untyped Lambda-Calculus

Syntax of the untyped lambda-calculus

var	$\ni$	$a, b, ab, ..$	Variable names
$term$	$::=$	var	Variable
	$\mid$	$\lambda var.term$	Abstraction
	$\mid$	$term_1\ term_2$	Application

Untyped Lambda-Calculus extended with natural numbers

Syntax of untyped lambda-calculus

num	$\ni$	$0, 1...$	Natural numbers
var	$\ni$	$a, b, ab, ..$	Variable names
$term$	$::=$	var	Variable
	$\mid$	$\lambda var.term$	Abstraction
	$\mid$	$term_1\ term_2$	Application

Assume the plus operator $+$ is a builtin function:

$$
\begin{aligned}
&y \\
&\lambda x.\ x \\
&\lambda x.\ x + 2 \\
&\lambda x.\ \lambda y.\ x \\
&(\lambda x.x + 2)\ 2 \\
&(\lambda x.x)\ 2
\end{aligned}
$$

Untyped Lambda-Calculus: Reduction

- The process of evaluating terms is called *reducing* them
- Reducing $(\lambda x.x + 2)\, 2$ will give us *4*
- The formal name for this process is called *beta-reduction*

Untyped Lambda-Calculus: Currying

- Lambda abstractions can only take *one* argument
- Multiple argument functions are encoded as functions that return *functions*
- Application is *left associative*: $\mathit{fun}\ \mathit{arg}_1\ \mathit{arg}_2$ is the same as $(\mathit{fun}\ \mathit{arg}_1)\ \mathit{arg}_2$

Example: $f(x, y) = x + y$ is encoded as $\lambda x.\,(\lambda y.\,x + y)$.

- $(\lambda x.\,(\lambda y.\,x + y))5$ reduces to $\lambda y.\,5 + y$
- $(\lambda x.\,(\lambda y.\,x + y))5 : 9$ reduces to $5 + 9$

Untyped Lambda-Calculus: Meaningless terms

What is the meaning of the following examples?

- 3 3 (we are applying 3 to 3)
- 9 $\lambda x.\, x$
- $(\lambda x.\, x)(\lambda y.\, y)$

Answer: They **Don't** have one!
In a programming language like Python, such terms result in a **runtime error**!

Simply Typed Lambda-Calculus

The syntax of Simply Typed Lambda-Calculus extended with natural numbers

tyvar	$\ni$	$\alpha, \beta, ...$	Type variable
num	$\ni$	$0, 1...$	Natural numbers
var	$\ni$	$a, b, ab, ..$	Variable names
Type	$::=$	*tyvar*	Type variable
	$\mid$	$\mathbb{N}$	Natural number
	$\mid$	$Type \rightarrow Type$	Function
term	$::=$	*var*	Variable
	$\mid$	$\lambda var : Type.\ term$	Abstraction
	$\mid$	$term_1\ term_2$	Application
	$\mid$	*num*	Natural number

Simply Typed Lambda-Calculus

For our purposes, you can think of a *type* as classifier for *terms*. But remember a term *can't have multiple types!*. Examples:

- All the terms of type $A \to B$ are *functions* that given something of type A would return something of type B
- All the terms of type $\mathbb{N}$ are *numbers*

Rules:

- Abstraction: if $x : A$ and $term : B$, then $\lambda x : A.\ term : A \to B$
- Application: if $fun : A \to B$ and $arg : A$, then $fun\ arg : B$

Laws:

- Function arrow is right associative: $A \to B \to C$ is the same as $A \to (B \to C)$

Example: If $\lambda x : \mathbb{N}.\ x + 2 : \mathbb{N} \to \mathbb{N}$ and $4 : \mathbb{N}$, then $(\lambda x : \mathbb{N}.\ x + 2 : \mathbb{N} \to \mathbb{N})\ 4 : \mathbb{N}$

The essence of functional programming

- Higher-order functions (first-class functions)
- Recursion over loops
- Lambdas (anonymous functions)
- Prioritizing purity and separating effectful computations
- Avoiding hidden state
- Persistant data structures
- Declarative style
- Composability

Nice to have features

- Static typing
- Type inference
- Pattern matching
- Guards
- Algebraic datatypes
- Read-Eval-Print Loop

Purity

A *pure* function is a function in the mathematical sense of the word. i.e.

- Given identical inputs, the function returns identical outputs.
- No side effects

Benefits of pure functions:

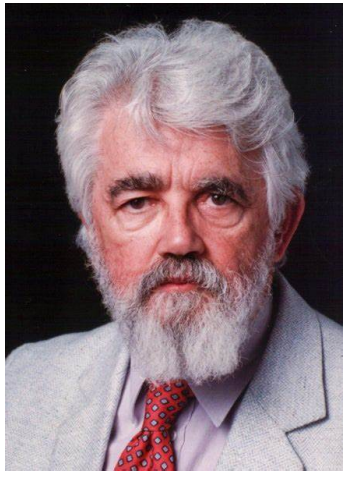
- Referential transparency
- No hidden state or action
- Easier to debug
- Easier to test

Laziness

- The arguments of a function are not evaluated until needed
- Laziness *forces* purity
- Defining control-flow as functions instead of primitives or macros
- Infinite data structures

Example in pseudo-code: if True then return 9 else return VeryBigDataStructure!!!
In this case VeryBigDataStructure is never evaluated!

Lisp



John McCarthy, Lisp

Steeve Russel

- Family of Meta-programming languages, first specified in 1960 by John McCarthy
- Short for *List Processing*
- Macros (meta-programming)
- Introduced garbage collection
- Introduced Read-Eval-Print Loop, i.e. REPL

Lisp

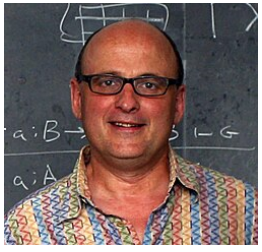
Guy Steele, Common Lisp, Scheme

Matthias Felleisen, Scheme, Racket

- Introduced dynamic typing
- Introduced conditionals
- Introduced higher-order functions
- Introduced tail-call optimization

ML

Robin Milner, type system, ML



Robert Harper, module system, SML

Mads Tofte, formal definition, SML

David MacQueen, core language, SML

- Short for *Meta Language*
- Family of strict and impure functional languages. Some implementations: SML, OCaml, F#
- Introduced type inference with the Hindley-Milner type system
- Algebraic datatypes and pattern matching
- ML modules

Erlang


Joe Armstrong


Robert Virding


Mike Williams

Erlang

- Strong support for concurrency
- Strict and dynamically typedx
- Developed in 1986 at Ericsson for telecommuting purposes
- Named after mathematician Agner Krarup Erlang and short for *Ericson Language*
- Hot code reloading
- BEAM VM
- REPL

Miranda

David Turner

Miranda

- Created in 1985 by David Turner
- Lazy and pure
- Algebraic datatypes
- Hindley-Milner type inference
- List comprehensions
- Beautiful syntax
- Proprietary

Haskell: summary

Haskell Curry

Haskell

- Lazy and pure
- Similar syntax to Miranda
- Named after logician Haskell Brooks Curry
- List comprehension
- Kind system
- Type inference
- Introduced type classes (ad-hoc polymorphism) and Monads

Haskell: history

Working group 2.8

- Devised by a committee in 1998
- Greatly inspired by Miranda
- Developed to have a single standard for *pure* and *lazy* functional languages
- Later standards Haskell98 and Haskell2010

Haskell: The original Haskell98 committee

- Simon Peyton Jones, Microsoft Research, Cambridge
- Lennart Augustsson, Sandburst Corporation
- Dave Barton, Intermetrics
- Brian Boutel, Victoria University of Wellington
- Warren Burton, Simon Fraser University
- Joseph Fasel, Los Alamos National Laboratory
- Kevin Hammond, University of St. Andrews
- Ralf Hinze, University of Bonn
- Paul Hudak, Yale University
- John Hughes, Chalmers University of Technology
- Thomas Johnsson, Chalmers University of Technology
- Mark Jones, Oregon Graduate Institute
- John Launchbury, Oregon Graduate Institute
- Erik Meijer, Microsoft Corporation
- John Peterson, Yale University
- Alastair Reid, University of Utah
- Colin Runciman, York University
- Philip Wadler, Avaya Labs

Haskell: Some of the influential people throughout the years

Simon Peyton Jones, Philip Wadler, Stephenie Weirich, Richard Eisenberg, Arvind, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Manuel M T Chakravarty, Duncan Coutts, Jon Fairbairn,Joseph Fasel, John Goerzen,Andy Gordon, Maria Guzman, Kevin Hammond, Bastiaan Heeren, Ralf Hinze, Paul Hudak, John Hughes, Thomas Johnsson, Isaac Jones, Mark Jones, Dick Kieburtz, John Launchbury, Andres Löh, Ian Lynagh, Simon Marlow, John Meacham, Erik Meijer, Ravi Nanavati, Rishiyur Nikhil, Henrik Nilsson, Ross Paterson, John Peterson, Mike Reeve, Alastair Reid, Vladislav Zaviolav, Ryan Scott, Colin Runciman, Don Stewart, Martin Sulzmann, Audrey Tang, Simon Thompson, Malcolm Wallace, David Wise, Jonathan Young, and many more!

Haskell: implementations

Haskell has had many implementation, but currently GHC is the most performant and featurful implementaion.

- GHC
- hbc
- hugs
- Yale Haskell
- UHC
- LHC

GHC Haskell

Simon Peyton Jones

Simon Marlow

Figure: Original authors of GHC

Some of the features of GHC: Impredicativity, GADTs, Existential types, Type families, Type abstractions, Required type arguments ,Pattern synonyms, Higher-Rank types, Kind polymorphism, Dependent kinds and ...!

GHCi: GHC's REPL

```
artin@Batcomputer:~/ > ghci
GHCi, version 9.10.1: https://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /Users/artin/.ghc/ghci.conf
λx. "Hello!"
"Hello!"
λx. :t \x -> x
\x -> x :: p -> p
λx.
```

- GHC has a REPL called GHCi that let's you compile Haskell to byte code and run it interactively.
- You can ask for the type of something in GHCi by using the :t command

Defining functions

```haskell
two :: Integer
two = 2

plusTwo :: Integer -> Integer
plusTwo = \x -> x + 2

plusTwo' :: Integer -> Integer
plusTwo' x = x + 2

const :: a -> b -> a
const x y = x
```

- The part with :: is called the *type signature* of the function
- The part with = is called the *function defintion* or the *function equation*

Using functions

Just like in Lambda-calculus, applying a function to its argument is done without putting them inside parenthesis.
Applying the function f to arguments x and y:

- Python: f(x,y)
- Haskell: f x y
- Lambda-Calculus: f x y

Operators are functions too

```
(+++) :: String -> String -> String
str1 +++ str2 = concat str1 str2

-- s1 +++ s2 +++ s3 == (s1 +++ s2) +++ s3
--                  == s1 +++ (s2 +++ s3)
```

- Easier to chain together
- Easier to show algebraic laws like associativity

Function composition

- Just another function!
- Takes two functions and a value. First applies the second argument and then the first
- The . operator is amongst the most used operators!

```haskell
compose :: (b -> c) -> (a -> b) -> a -> c
compose g f x = g (f x)

(.) :: (b -> c) -> (a -> b) -> a -> c
(.) g f x = g (f x)

-- h . g . f = h . (g . f)
```

Function composition: example

$\backslash x \rightarrow x * 3$ is the same thing as $(* 3)$.

```
compositionExample = (\y -> y^9 ) . (\x -> x * 3) . plusTwo . const 4
```

```
compositionExample2 = (^ 9) . (* 3) . plusTwo . const 4
```

```
λ> compositionExample 2
19835929036 8
λ> compositionExample2 2
19835929036 8
```

List and laziness

- The empty list is represented using []
- We use the cons operator (:) to add an element to the start of a list

```
λ> emptyList = []
λ> oneAndTwo = 1 : 2 : []
λ> oneAndTwo' = [1,2]
λ> oneAndTwo == oneAndTwo'
True
λ> 3 : oneAndTwo
[3,1,2]
```

```
λ> const "Hello!" [0..]
"Hello!"
```

Algebraic Datatypes

Algebraic Data Types are created by the combination of:

- Sum types i.e. better enums
- Product types i.e. better structs

Examples:

- Sum type: Bool is *either* True or False
- Product type: The 2-tuple (Int,String) needs both an Int and a String. (3, "Hi!"")

Algebraic Datatypes: Food type

```haskell
type Size = Int

data HasRice = Yes | No

data Food = Pizza Size | Kebab Size HasRice
```

```
λ> myPizza = Pizza 5
λ> myKebab = Kebab 8 Yes
λ> :t myPizza
myPizza :: Food
λ> :t myKebab
myKebab :: Food
```

Food has two *data constructors*:

- Pizza: In order to make Pizza you need to give it a value of type Size
- Kebab: In order to make Kebab you need to give it a value of type Size **and** a value of type HasRice

Pattern matching

You can *match* and *deconstruct* patterns on the left hand side of an equation.

```
isKebabWithRice :: Food -> HasRice
isKebabWithRice (Kebab _ hasRice) = hasRice
isKebabWithRice _                 = No

getFoodSize :: Food -> Size
getFoodSize (Pizza size)   = size
getFoodSize (Kebab size _) = size

isPizzaOfSizeSeven :: Food -> Bool
isPizzaOfSizeSeven (Pizza 7) = True
isPizzaOfSizeSeven _         = False
```

- Patterns can *deconstruct* data constructors, *introduce* variables, or always match by having a *wildcard*
- Patterns can be as nested as you want!

The map function

The map function applies a function to all of the elements of a list
Examples

```
λ> map (\x -> x + 2) [1,2,3]
[3,4,5]
```

Example 1

```
λ> map (\n -> say n "H") [1,2,3]
["H","HH","HHH"]
```

Example 2

The map function in action

```
map :: (a -> b) -> List a -> List b
map f [] = []
map f (x : xs) = f x : map f xs
```

Important Note: Because of laziness, Haskell doesn't actually reduce the applications until it's needed!

Let's evaluate map (+2) [1,2,3]

1. map (+2) [1,2,3]

The map function in action

```
map :: (a -> b) -> List a -> List b
map f [] = []
map f (x : xs) = f x : map f xs
```

Important Note: Because of laziness, Haskell doesn't actually reduce the applications until it's needed!

Let's evaluate map (+2) [1,2,3]

1. map (+2) [1,2,3]
2. (1 + 2) : map (+2) [2,3]

The map function in action

```
map :: (a -> b) -> List a -> List b
map f [] = []
map f (x : xs) = f x : map f xs
```

Important Note: Because of laziness, Haskell doesn't actually reduce the applications until it's needed!

Let's evaluate map (+2) [1,2,3]

1. map (+2) [1,2,3]
2. (1 + 2) : map (+2) [2,3]
3. (1 + 2) : (2 + 2) : map (+2) [3]

The map function in action

```
map :: (a -> b) -> List a -> List b
map f [] = []
map f (x : xs) = f x : map f xs
```

Important Note: Because of laziness, Haskell doesn't actually reduce the applications until it's needed!

Let's evaluate map (+2) [1,2,3]

1. map (+2) [1,2,3]
2. (1 + 2) : map (+2) [2,3]
3. (1 + 2) : (2 + 2) : map (+2) [3]
4. (1 + 2) : (2 + 2) : (3 + 3) : map (+2) []

The map function in action

```
map :: (a -> b) -> List a -> List b
map f [] = []
map f (x : xs) = f x : map f xs
```

Important Note: Because of laziness, Haskell doesn't actually reduce the applications until it's needed!

Let's evaluate map (+2) [1,2,3]

1. map (+2) [1,2,3]
2. (1 + 2) : map (+2) [2,3]
3. (1 + 2) : (2 + 2) : map (+2) [3]
4. (1 + 2) : (2 + 2) : (3 + 3) : map (+2) []
5. (1 + 2) : (2 + 2) : (3 + 2) : []

The map function in action

```
map :: (a -> b) -> List a -> List b
map f [] = []
map f (x : xs) = f x : map f xs
```

Important Note: Because of laziness, Haskell doesn't actually reduce the applications until it's needed!

Let's evaluate map (+2) [1,2,3]

1. map (+2) [1,2,3]
2. (1 + 2) : map (+2) [2,3]
3. (1 + 2) : (2 + 2) : map (+2) [3]
4. (1 + 2) : (2 + 2) : (3 + 3) : map (+2) []
5. (1 + 2) : (2 + 2) : (3 + 2) : []
6. 3 : 4 : 5 : []

The map function in action

```
map :: (a -> b) -> List a -> List b
map f [] = []
map f (x : xs) = f x : map f xs
```

Important Note: Because of laziness, Haskell doesn't actually reduce the applications until it's needed!

Let's evaluate map (+2) [1,2,3]

1. map (+2) [1,2,3]
2. (1 + 2) : map (+2) [2,3]
3. (1 + 2) : (2 + 2) : map (+2) [3]
4. (1 + 2) : (2 + 2) : (3 + 3) : map (+2) []
5. (1 + 2) : (2 + 2) : (3 + 2) : []
6. 3 : 4 : 5 : []
7. [3,4,5]

The Maybe type and its usage

```haskell
data Maybe a = Just a
             | Nothing

(/?) :: Integer -> Integer -> Maybe Integer
x /? 0 = Nothing
x /? y = Just (division x y)
```

Why is this better than throwing an error?

- Errors are thrown when we *run* the program. We want the compiler to catch the bug during *compilation*!
- By reading the type of $(/?)$ i.e. $Integer \to Integer \to MaybeInteger$, we find out that this function could *fail*

The Maybe type and its usage

Important note: GHCi compiles the code before running it

```
λ> 2 + (2 /? 0)
<interactive>:189:6-11: error: [GHC-83865]
    • Couldn't match expected type ‘Integer’
                  with actual type ‘Maybe Integer’
    • In the second argument of ‘(+)’, namely ‘(2 /? 0)’
      In the expression: 2 + (2 /? 0)
      In an equation for ‘it’: it = 2 + (2 /? 0)
```

Type classes: Ad-hoc polymorphism

Type classes give us access to ad-hoc polymorphism.
As an example, The Show type class lets us convert values into strings.

```haskell
class Show a where
  show :: a -> String

instance Show Bool where
  show True  = "True"
  show False = "False"
```

```
λ> :t show
show :: Show a => a -> String
λ> show 2
"2"
λ> show True
"True"
λ> show [ [True, True], [False, False] ]
"[[True,True],[False,False]]"
```

Type classes: Deriving

Haskell can write *most* of the type class instances for you!

```haskell
data Day = Saturday
         | Sunday
         | Monday
         | Tuesday
         | Wednesday
         | Thursday
         | Friday
         deriving (Show, Eq)
```

Day type

```
λ> show Saturday
"Saturday"
λ> Saturday == Saturday
True
```

Examples in GHCi

Datatypes as contexts

- List: Order matters
- Tree: Easy navigation and lookup
- Maybe: Computation could fail

All of these have one thing in common: They all hold values inside them, and you can see the type of that value from outside.

- Maybe String holds a value of type String
- List a holds values of type a
- Tree Food holds values of type Food

The Functor class

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor [] where
  fmap = map

instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap f Nothing = Nothing
```

Laws:

- $fmap\ (g \circ f)\ x \equiv (fmap\ g \circ fmap\ f)\ x$
- $fmap\ id\ x \equiv x$

Functor: two instances

Yes! the Functor instance of the List type is just the map function!

```haskell
instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap f Nothing  = Nothing

instance Functor List where
  fmap f (x : xs) = f x : fmap f xs
  fmap f []       = []
```

Functor: examples

```
λ> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f b
```

```
λ> fmap (\x -> x + 2) (Just 2)
Just 4
λ> fmap (\x -> x + 2) Nothing
Nothing
λ> fmap (\x -> x + 2) [1,2]
[3,4]
λ> fmap (\x -> x + 2) []
[]
```

Type Inference

```
and True x  = x
and False _ = False
```

```
λ> :type and
and :: Bool -> Bool -> Bool
λ>
```

Haskell is smart enough to *infer* the type of most functions by itself. And if you give it the wrong argument, it will tell you exactly what you did wrong!
e.g. test = and True 'c' will result in the following error message:

```
Main.hs:35:17: error: [GHC-83865]
    • Couldn't match expected type ‘Bool’ with actual type ‘Char’
    • In the second argument of ‘and’, namely ‘'c'’
      In the expression: and True 'c'
      In an equation for ‘test’: test = and True 'c'
   |
35 | test = and True 'c'
   |                 ^^^
```

Type Inference

```
and True x  = x
and False _ = False
```

```
λ> :type and
and :: Bool -> Bool -> Bool
λ>
```

Haskell is smart enough to *infer* the type of most functions by itself. And if you give it the wrong argument, it will tell you exactly what you did wrong!
e.g. test = and True 'c' will result in the following error message:

```
Main.hs:35:17: error: [GHC-83865]
    • Couldn't match expected type ‘Bool’ with actual type ‘Char’
    • In the second argument of ‘and’, namely ‘'c'’
      In the expression: and True 'c'
      In an equation for ‘test’: test = and True 'c'
   |
35 | test = and True 'c'
   |                 ^^^
```

Type Inference

Haskell's type system is clever enough that we didn't actually need to write *any* type signatures for the examples I showed you! To emphasis this, the type signatures of the upcoming examples are commented out.
Here are some good reasons for why you should write them though:

- Type signatures are like documentation, they tell us a lot about the function at hand
- Sometimes Haskell infers types that are *more* polymorphic than you'd want
- When refactoring, you might unknowingly change the type of functions

Example: factorial in Python

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Example: factorial in Haskell

```haskell
-- factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Example: factorial in Haskell

```haskell
-- factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Let's evaluate factorial 3:

1. factorial 3
2. 3 * factorial 2

Example: factorial in Haskell

```haskell
-- factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Let's evaluate factorial 3:

1. factorial 3
2. 3 * factorial 2
3. 3 * 2 * factorial 1

Example: factorial in Haskell

```haskell
-- factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Let's evaluate factorial 3:

1. factorial 3
2. 3 * factorial 2
3. 3 * 2 * factorial 1
4. 3 * 2 * 1 * factorial 0

Example: factorial in Haskell

```haskell
-- factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Let's evaluate factorial 3:

1. factorial 3
2. 3 * factorial 2
3. 3 * 2 * factorial 1
4. 3 * 2 * 1 * factorial 0
5. 3 * 2 * 1 * 1

Example: factorial in Haskell

```haskell
-- factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Let's evaluate factorial 3:

1. factorial 3
2. 3 * factorial 2
3. 3 * 2 * factorial 1
4. 3 * 2 * 1 * factorial 0
5. 3 * 2 * 1 * 1
6. 6 * 1 * 1

Example: factorial in Haskell

```haskell
-- factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Let's evaluate factorial 3:

1. factorial 3
2. 3 * factorial 2
3. 3 * 2 * factorial 1
4. 3 * 2 * 1 * factorial 0
5. 3 * 2 * 1 * 1
6. 6 * 1 * 1
7. 6 * 1

Example: factorial in Haskell

```haskell
-- factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Let's evaluate factorial 3:

1. factorial 3
2. 3 * factorial 2
3. 3 * 2 * factorial 1
4. 3 * 2 * 1 * factorial 0
5. 3 * 2 * 1 * 1
6. 6 * 1 * 1
7. 6 * 1
8. 6

Example: say in Python

say is a function that given a number *n* and a string *string*, will repeat the *string* *n* times.
e.g. say(2,"Hi ") = "Hi Hi "

```
def say(n,string):
    i = 0
    final = ""
    while i < n:
        final = string + final
        i += 1
    return final
```

Example: say in Haskell

The ++ operator concatenates two strings together.

```haskell
-- say :: Int -> String -> String
say 0 _   = ""
say n str = str ++ say (n - 1) str
```

Example: say in Haskell

The ++ operator concatenates two strings together.

```haskell
-- say :: Int -> String -> String
say 0 _   = ""
say n str = str ++ say (n - 1) str
```

Let's evaluate say 2 "Hi ":

1. say 2 "Hi "
2. "Hi " ++ say 1 "Hi"

Example: say in Haskell

The ++ operator concatenates two strings together.

```haskell
-- say :: Int -> String -> String
say 0 _   = ""
say n str = str ++ say (n - 1) str
```

Let's evaluate say 2 "Hi ":

1. say 2 "Hi "
2. "Hi " ++ say 1 "Hi"
3. "Hi " ++ "Hi " ++ say 0 "Hi"

Example: say in Haskell

The ++ operator concatenates two strings together.

```haskell
-- say :: Int -> String -> String
say 0 _   = ""
say n str = str ++ say (n - 1) str
```

Let's evaluate say 2 "Hi ":

1. say 2 "Hi "
2. "Hi " ++ say 1 "Hi"
3. "Hi " ++ "Hi " ++ say 0 "Hi"
4. "Hi " ++ "Hi " ++ ""

Example: say in Haskell

The ++ operator concatenates two strings together.

```haskell
-- say :: Int -> String -> String
say 0 _   = ""
say n str = str ++ say (n - 1) str
```

Let's evaluate say 2 "Hi ":

1. say 2 "Hi "
2. "Hi " ++ say 1 "Hi"
3. "Hi " ++ "Hi " ++ say 0 "Hi"
4. "Hi " ++ "Hi " ++ ""
5. "Hi Hi " ++ ""

Example: say in Haskell

The ++ operator concatenates two strings together.

```haskell
-- say :: Int -> String -> String
say 0 _   = ""
say n str = str ++ say (n - 1) str
```

Let's evaluate say 2 "Hi ":

1. say 2 "Hi "
2. "Hi " ++ say 1 "Hi"
3. "Hi " ++ "Hi " ++ say 0 "Hi"
4. "Hi " ++ "Hi " ++ ""
5. "Hi Hi " ++ ""
6. "Hi Hi "

Example: hasEven in Python

hasEven is a function that given a character, *char* ,and a string *string*, will return *true* if the number of times *char* occurs in *string* is even.

```
def hasEven(char, string):
    i = 0
    for c in string:
        if c == char:
            i += 1
    return (True if (i % 2) == 0 else False)
```

Example: hasEven in Haskell

even is a built-in function that says whether its input is even or not.

```haskell
-- hasEven :: Char -> String -> Bool
hasEven char string = even (loop 0 string)
  where
    loop n ""             = n
    loop n (ch : chars) = if ch == char
                            then loop (n + 1) chars
                            else loop n chars
```

Example: hasEven in Haskell

```haskell
-- hasEven :: Char -> String -> Bool
hasEven char string = even (loop 0 string)
  where
    loop n ""             = n
    loop n (ch : chars) = if ch == char
                            then loop (n + 1) chars
                            else loop n chars
```

Example: hasEven in Haskell

```haskell
-- hasEven :: Char -> String -> Bool
hasEven char string = even (loop 0 string)
  where
    loop n ""             = n
    loop n (ch : chars) = if ch == char
                            then loop (n + 1) chars
                            else loop n chars
```

Let's evaluate hasEven 'e' "eyes":

1. even (loop 0 "eyes")
2. even (loop 1 "yes")

Example: hasEven in Haskell

```haskell
-- hasEven :: Char -> String -> Bool
hasEven char string = even (loop 0 string)
  where
    loop n ""             = n
    loop n (ch : chars) = if ch == char
                            then loop (n + 1) chars
                            else loop n chars
```

Let's evaluate hasEven 'e' "eyes":

1. even (loop 0 "eyes")
2. even (loop 1 "yes")
3. even (loop 1 "es")

Example: hasEven in Haskell

```haskell
-- hasEven :: Char -> String -> Bool
hasEven char string = even (loop 0 string)
  where
    loop n ""           = n
    loop n (ch : chars) = if ch == char
                          then loop (n + 1) chars
                          else loop n chars
```

Let's evaluate hasEven 'e' "eyes":

1. even (loop 0 "eyes")
2. even (loop 1 "yes")
3. even (loop 1 "es")
4. even (loop 2 "s")

Example: hasEven in Haskell

```haskell
-- hasEven :: Char -> String -> Bool
hasEven char string = even (loop 0 string)
  where
    loop n ""           = n
    loop n (ch : chars) = if ch == char
                          then loop (n + 1) chars
                          else loop n chars
```

Let's evaluate hasEven 'e' "eyes":

1. even (loop 0 "eyes")
2. even (loop 1 "yes")
3. even (loop 1 "es")
4. even (loop 2 "s")
5. even (loop 2 "")

Example: hasEven in Haskell

```haskell
-- hasEven :: Char -> String -> Bool
hasEven char string = even (loop 0 string)
  where
    loop n ""           = n
    loop n (ch : chars) = if ch == char
                          then loop (n + 1) chars
                          else loop n chars
```

Let's evaluate hasEven 'e' "eyes":

1. even (loop 0 "eyes")
2. even (loop 1 "yes")
3. even (loop 1 "es")
4. even (loop 2 "s")
5. even (loop 2 "")
6. even 2

Example: hasEven in Haskell

```haskell
-- hasEven :: Char -> String -> Bool
hasEven char string = even (loop 0 string)
  where
    loop n ""           = n
    loop n (ch : chars) = if ch == char
                          then loop (n + 1) chars
                          else loop n chars
```

Let's evaluate hasEven 'e' "eyes":

1. even (loop 0 "eyes")
2. even (loop 1 "yes")
3. even (loop 1 "es")
4. even (loop 2 "s")
5. even (loop 2 "")
6. even 2
7. True

Python

- Lambdas
- Dynamic Typing
- Garbage collection
- List comprehension

Excel

As unbelievable as it is, Excel has become turing complete! Simon Peyton Jones (Haskell) and Andy Gordon have added lambdas to Excel in 2021

Rust

- Anonymous functions
- Traits, which are the equivalents of Haskell's type classes
- Prioritizing immutability
- Pattern matching
- The type system itself is inspired on ML
- Macros which are the same as Scheme's macro system

Swift

- SwiftUI, apple's UI library has many functional philosophies
- Closures, which are close to lambdas
- Syntactic sugar for closures
- Enums and structs are algebraic datatypes!
- Pattern matching
- Optional chaining is based on the idea of Monads!

Java

- Lambdas
- Java's generics was designed by Philip Wadler (Haskell) himself!
- Pattern matching

React

React is heavily based on functional programming ideas like pure functions, immutability, avoiding mutation, and avoiding state!

In industry

- NASA' copilot and ogma projects are written in Haskell
- Meta's spam filter is written in Haskell
- JP Morgan uses Haskell
- IOHK cardano uses Haskell
- Siemens uses Haskell
- Jane Street uses OCaml
- Epic Games is developing Verse to use in its metaverse
- X's backend is written Scala
- Spotify's backend is written in Scala
- Whatsapp is written in Erlang
- Discord is written in Elixir
- Tesla uses Haskell to generate C code for its cars
- And many more!

More functional languages

- Idris2
- Flix
- Lean4
- OCaml
- Verse
- Racket
- Janet
- Common Lisp
- Scheme
- Clojure
- Gleam
- Scala
- Elixir
- OCaml
- F#