Nothing stops you from making a C interpreter.

- Interpreter
  - Interactive
  - Slow

- Compiler
  - Fast due to optimizations

Stalin Compiler: Full ~~pro~~ ~~prots~~ program optimizer

Scheme to C compiler.

GHC → Glasgow Haskell compiler

Hugs → Interpreter.

ghci — REPL: read eval print loop

```
ghci > 1+2
ghci > 'c'
ghci > :type 'c'        , :t
module                  # to create a module
:info curry             #                    :info Num.
```

ghc --make hello.hs

./ hello

→ Kabal

- kabal build

Functional Programming

▷ There is no distinction b/w ~~a~~ a value and functions.

~~SS~~

Declarative ~~by~~ Style : Describe "WHAT" not "HOW"

"Douglas" := advancement of civilization.

words :: String → [string]

↓

list of strings

words "that is it" → ["That", "is", "it"]

length :: [a] → Int polymorphic length function.

wc str = length (words str)

31-1-24

words :: String → [String]

length :: [a] → Int

↑ type variable

wc :: String → Int

words "That is it"

["That", "is", "it"]

wc str = length (words str)

wc = length . words

[don't forget to leave space before and after the dot]

Haskell: → Strongly typed, Statically typed, } Types matter.

~~Paramet~~ allows Parametric Polymorphism.

principled Adhoc polymorphism.

[Parametric polymorphism should be actually called "Polymorphism"!!

Overloading is called ~~ad-hoc~~ "adhoc Polymorphism". Overloading is case by case. Polymorphism is generic.]

(1) Function Notations

$f(x)$ $\quad$ $f\ x$

(2) All functions are unary.
Then how to write a 2 argument function.

2.1 (Int, Bool) → String

2.2 Int → (Bool → String)

Types:

(1) Int, Char, Bool

[Type names start with capital letters.
Variables start with small letters.]

(2) If τ is a type then $[\tau]$ is a type.
↑ list of τ.

(3) τ_1 and τ_2 are types then $\tau_1 \to \tau_2$ is a type.
(minus . gt)

(4) If τ_1 and τ_2 are types then (τ_1, τ_2) is a type
(τ_1, τ_2, τ_3)
⋮
Cartesian product of τ_1 and τ_2.

elements are ordered pairs (x, y) where $x :: \tau_1$
$y :: \tau_2$.

Convention is Haskell is to use Camel Case. Eg: "fooBar"

```
x = 42
```
This equation defines x

```
foo y = y + 42
```
this equation defines foo and does not define y.

```
foo z = z + 42
```
" "

z

(Int, String)

(1, "Hello")

[Int]

[2,3,5]

[]

2 : 3 : 5 : []

if xs :: [a]
x :: a

then
x : xs :: [a]

([Int], String)

2-2-24

Basic Types Int, Bool, ...

"Compound" types If τ_1 and τ_2 are ~~then~~ types then (τ_1, τ_2)

List types [a] - list of a's.

Introduction Rule

If $x_1 :: \tau_1$ and $x_2 :: \tau_2$ then $(x_1, x_2) :: (\tau_1, \tau_2)$

$$\frac{x_1 :: \tau_1 \qquad x_2 :: \tau_2}{(x_1, x_2) :: (\tau_1, \tau_2)}$$

Intro rule for products.

Intro rule for List

nil rule $\dfrac{}{[\,] :: [a]}$

cons rule $\dfrac{x :: a \qquad xs :: [a]}{(x : xs) :: [a]}$

when you see $[1, 2, 3]$ it is a syntactic sugar.

This actually is $1 : 2 : 3 : []$

fst :: $(a, b) \to a$

snd :: $(a, b) \to b$

Q. Always ask the question

fst $(x_1, x_2) = x_1$

snd $(x_1, x_2) = x_2$

Pattern matching

— Will take to ~~show~~ only ~~error~~ for non-exhaustive definition. to accept only exhaustive definitions

length :: [a] → Int

length [] = 0

length (x:xs) = 1 + length xs

sum :: [Int] → Int

sum [] = 0

sum (x:xs) = x + sum xs

~~is Empt~~

isEmpty :: [a] → Bool

isEmpty [] = True

isEmpty x:xs = False

wildcard '_' (underscore)

fst $(x_1, \_) = x_1$

snd $(\_, x_2) = x_2$

isEmpty [] = True

isEmpty _ = False.

If P_1 and P_2 are patters then (P_1, P_2)

~~$P_1 : P_2$~~ $P_1 : P_2$

variable is a pattern
wildcard is a pattern.

Clink ? ~~Hlinte~~ Hhint
Clinte will give you suggestions for your code.

```
take :: Int → [a] → [a]

take  n  []  =  []

take  n  [x:xs]  =  if  n > 0
                    then  x:(take(n-1) xs)
                    else  []
```

function application associates towards the left and arrow associates towards the right

* Operators have less precedence than function application.

$(x, y) = (10, "Hello")$

3-2-24

→ Pattern matching based - definition.

→ Patterns are essentially subclass of expression

(1) They are linear, no variable is repeated twice

(2) There is the wild card pattern.

(3) It should

[We can't arbitrarily pattern match on function applications
eg: foo (f x) we can't pattern match like this.]

Maps and folds

map

(1) I have a function from $a \to b$

(2) I have a list of a's

(3) I should compute the list obtained by applying f on all values of the list on (2)

```
map :: (a->b) -> [a] -> [b]
```

~~map f xs =~~

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

Foldr, Foldl

$[a_1, a_2 \cdots a_m] :: [a]$

```
o :: a -> s -> s
```

foldr : $= (a_1 \circ (a_2 \circ (\cdots (a_m \circ s_0))))$

```
prod :: [Int] -> Int
sum :: [Int] -> Int
```

prod $[a_0\ a_2 \quad a_n] = a_1 \times a_2 \cdots a_{n+1}$

sum $[a_1\ a_2 \quad a_n] = a_1 + a_2$ &c.

Foldl

$$(((a_1 \circ s_0) \circ a_2) \circ a_3) \cdots \circ a_n)$$

foldr :: $(a \to s \to s) \to s \to [a] \to s$

```
foldr f s [] = s
foldr f s (x:xs) = f x (foldr f s xs)
```

~~foldl f s (x:xs) = foldl f (f x s) xs~~

foldl :: $(s \to a \to s)$

$\to s$

$\to [a]$

$\to s$

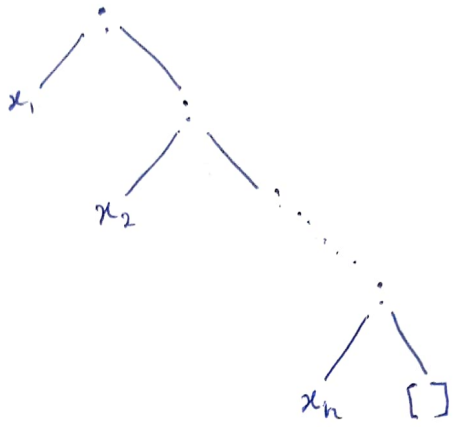
foldl f s_0 x_s = ___

foldl f s_0 [] = s_0

foldl f s_0 (x:xs) = foldl f (f s x) xs

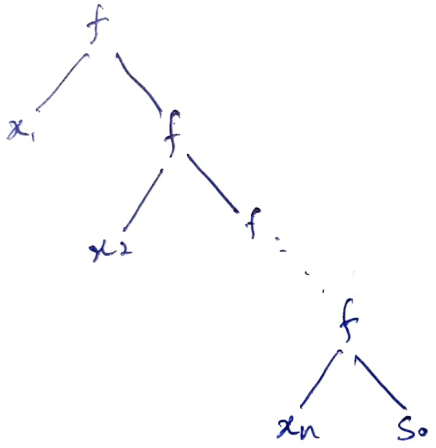
$[x_1, x_2 \ldots x_n] :: [a]$

$x_1 : (x_2 : \ldots (x_n : [\,]))$



foldr f s_0

→



converting operator to a function: (+) : Int → Int → Int

converting function to a operator: mod :: Int → Int → Int

ex: 5 `mod` 2

Any function f :: a → b → c can be converted to operator using `f`

(+2) : is a function that takes x and produces x+2

(2+) : " " x and produces 2+x

map (+1) : is a function which increments in every element of the list.

List

(1) [] :: [a]

(2) (:) :: a → [a] → [a]

we want to summarise [a] to S.

1.1 s :: S

2.1 cf :: a → S → S

HW write foldl as ff in terms of foldr.

foldr (:) [] → you get back the list

(:[]) :: a → [a] (takes a value and create a list with it)

(2:) :: [Int] → [Int] (takes a list and put 2 in front of it)

Lazy Evaluation

e_1 e_2 : Eager evaluation
(f u)

Haskell uses lazy evaluation.

for x y = if x>y then e_1
else y_1

where
y_1 = Big Computation.

ones = 1 : ones

y = take 100 ones

2-24

Lazy Evaluation

for x = let
y = Big
in
e

zipWith :: $(a \to b \to c) \to [a] \to [b] \to [c]$

$[a_1, a_2 \ldots a_n]$
$[b_1, b_2 \ldots b_n]$
↓
$[f(a_1, b_1), f(a_2, b_2) \ldots]$

~~zipWith f xs ys = r~~

```
zipWith f [] ys = []
zipWith f _ [] = []
zipWith f (x:xs) (y:ys) = f(x,y) : (zipWith f xs ys)
```

Definition Better:

```
zipWith f (x:xs) (y:xs) = f(x,y) : (zipWith f xs ys
zipWith _ _ _           = []
```

```
tl :: [a] -> [a]
tl (x:xs) = xs
```

```
fib = 1 : 1 : zipwith (+) fib (tl fib)
```

[1 1 2 3 5 8
[1 2 3 5 8 13
(tail of fib)

```
fib  = 1 : 1 : fib2
fib2 = zipWith (+) fib fib1
fib1 = tl fib
```

"Steer by wire, Latest Tesla Cybertruck"

~~sieve~~

2 3 ~~4~~ ~~5~~ ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~
11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~

```
primes = sieve [2..]
```

(this will be a list 2, 3 . . . .)

```
sieve :: [Integer] -> [Integer]
sieve (x:xs) = x : sieve (strikeOff x xs)
sieve (x:xs) = x : sieve (strikeOff x xs)
strikeOff x xs = remove (multiples x) xs
multiples x = [k * x | k <- [2..]]
```

[we can also add conditions like
k <- [2..], odd k]

```
remove rm@(r:rs) f@(y:ys) | y > r = remove rs (y:ys)
                          | y = r = remove rs ys
                          | otherwise = remove rm ys
```

Use f@(y:ys) to do pattern matching and

| → guarded

string : [char]

[[string]]

type Table = [[String]] Type aliasing

or we can define it like

```
type Row = [String]
type Table = [Row]
```

-- We have to pad the table with , such that each colum aligns

~~padTable padding Table = map (padCols padding) tab~~

~~padCols padding el col = zipWith~~

```
alignTable
padTable padding tab = map (padRow padding) tab
alignRow
padRow padding row = zipWith align padding row
```

```
align n s = s ++ repeat ' ' (n - length s)
```

```
zeros = 0 : zeros
findPadRow r = map length r ++ zeros
padTable = foldr combFn zeros
combFn :: [String] Row -> [Int] -> [Int]
combFn = zipWith                    -- complete this
```

for Input (1) a padding list paddings

(2) The table tab

↓

(alignTable paddingTab, padTable tab)

```
let (paddedTable, pad) = foo pad tab
in
  paddedTable
```

1-2-24

Lazy Evaluation : Pretty printing tables, one traversal over the structure.

NixOs based on ~~mix~~ nix package manager.

Problems of Laziness

foldr and foldl

```
foldr :: (a -> s -> s) -> [a] -> s -> s.

foldr f [ ] s0 = s0

foldr f x:xs s0 = let rec = foldr xs s0
                  in
                    f x rec
```

~~foldl :: (s -> a -> s) -> [a] -> s -> s~~

```
foldl :: (s -> a -> s) -> s -> [a] -> s

foldl f s0 [ ] = s0

foldl f s0 x:xs = ~~foldl f (f s0 x) xs~~
                = foldl f (f s0 x) xs

                or
                = let s1 = f s0 x
                  in foldl f s1 x
```

~~thunks or thunk~~ thunks ?

foldl (+) 0

let a1 = (+ 0 x1)

let a1 = (+ 0 x1)

let a2 = (+ a1 x2)

:

let an = (+ a(n-1) xn)

in an

Douglas Adams

If a value is not forced it will not be computed.

foo (x : xs) = ——

* When you call "foo ys". It will only check if ys is in the form x : xs

~~when~~ you call

foo x : xs = x + 1

when you call 'foo ys'. only the x part is evaluated to ~~use~~ use it in 'x + 1'. The rest of the list is not evaluated.

[foldl ? ~~foldl~~ ~~foldl~~ is more effient than foldl in a lazy evaluation]

data Tuple ~~a~~ a b = T a b

Tuple a b — new datatype with a single constructor

T :: a → b → Tuple a b

fstT :: Tuple a b → a

fstT (T a _) = a

sndT (T _ b) = b

17-2-23

data Tuple a b = T a b

(1) A new type Tuple a b for types a, b (Polymorphi

(2) A constructor T: a → b → Tuple a b

fst of Tuple (T x _) = x

snd of Tuple (T _ y) = y

data List a = Nil
| Cons a (List a)

NOTE: All types are lazy, even the ones we create.

So we can even defines

ones = Cons 1 ones using our defined List.

How to Define a binary tree

1) Empty tree

2) Node with left and right children.

```
data Tree a = Empty
            | Node (Tree a) a (Tree a)
```

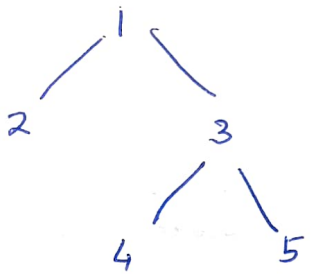
See this definition of Tree a as an expression, i.e. that Tree is either Empty or a Node with a value of type 'a', Left node and right node

Can we define a type without a base case

```
binTreeOnes = Node (binTreeOnes) 1 (binTreeOnes)

isEmptyTree :: Tree a → Bool

isEmptyTree Empty = True

isEmptyTree (Node L a R) = False

or [ isEmptyTree _ = False ]

singleton x = Node Empty x Empty   -- singleton :: a → Tree a

inorder :: Tree a → [a]
```

```
inorder Empty = []
inorder (Node ~~Empty~~ l a r) = inorder(l) ++ [a] ++ inorder (r)
                               = (inorder (l)) ++ [a] ++ (inorder [r])
```

~~inorder~~



- Node (singleton 2) 1 (Node (Singleton 4) 3 (Singleton 5))

```
depth :: Tree a → Int

depth Empty = 0

depth (Node lt _ rt) = 1 + ~~max(lt, rt)~~
                           max(depth lt, depth rt)
```

19-2-23

```
data Type a = Empty
            | Node (Tree a) a (Tree a)
```

Empty :: Tree a

lt :: Tree a
x :: a
rt :: Tree a

Node lt x rt :: Tree a

(1) A new type Tree a is Born

(2) Creates the constructors

(3) Expands the pattern matching rules.

P ::$_{pat}$ τ p is a pattern of type τ.

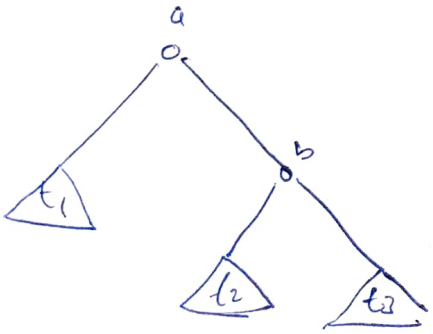
Empty ::$_{pat}$ Tree a

pl ::$_{pat}$ tree a

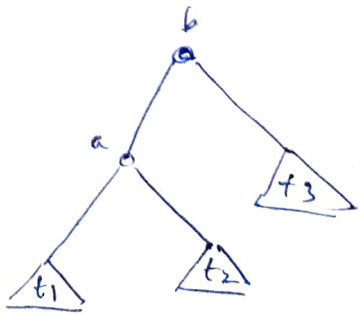
p ::$_{pat}$ a

pr ::$_{pat}$ Tree a

Node pl p pr ::$_{pat}$ Tree a



rotation →



rotateAC :: Tree a ⟶ Tree a

rotateAC (Node t_1 a (Node t_2 b t_3)) =

(Node (Node t_1 a t_2) b t_3)

rotateAC t = t

All datatypes are lazily evaluated.

HW

Functions, takes a tree of integers, ~~it takes~~ and outputs a tree with ~~max~~ all value as max value of the tree.

Do it in one pass.

Nothing :: Maybe a

---

x : a

Just x :: Maybe a

datatype Maybe a = Just a

| Nothing

data Either a b = Left a

| Right a

```
hd :: [a] → a
hd (x : _) = x
-- Unsafe

hds :: [a] → Maybe a
hds [] = Nothing
hds (x : _) = Just x.
```

-- This function is never safe because a is a type variable and it can be any thing.

But a function of type

[Int] → Int is

potentially safe because we can write some stupid functions.

So only safe function of type a → a is only the identity function.

```
root :: Tree a → Maybe a
root Empty = Nothing
root Node _ x _ = Just x
```

21-2-24

(1) data Maybe a = Just a | Nothing } 1 + a

(2) data Either a b = Left a | Right a } a + b

hd : [a] → Maybe a

hd : [a] → a ✗ Not possible.

Ex: Parse :: String → Either ParseError a.

Type classes: A principled approach to ad-hoc polymorphism.

```
foo : xs = Just y
  then for save y ∈ xs
```

Polymorphism is not only for convenience but it is also for correctness.

Parametric Polymorphism → real polymorphism [Eg: general function of Hd]

Adhoc Polymorphism → Overloading [Eg: '+']

[you use the same ~~it~~ name for different things. we decide functionality ~~on~~ on different ~~types~~ types looking at the type]

In ~~of poly~~ parametric polymorphism. ~~Diff~~ for any type we do the ~~same~~ ~~it~~ something.

eq :: a → a → Bool

"Zyada dimag math lagana"

class Eq a where {

(==)
eq :: a → a → Bool

instance Eq Bool where

(==) True True = True

(==) False False = True

(==) — — = False

instance Eq Int where

(==) : - - -

- - -

type (==) :: Eq a ⇒ a → a → Bool.

foo x y = if x == y then "equal"

else "not equal"

type foo Eq a ⇒ a → a → String

instance (Eq a, Eq b) ⇒ Eq (a, b) where

where

(==) (a1, b1) (a2, b2) = a1 == a2 && b1 == b2

Info Eq instance (Eq a, Eq b) ⇒ Eq (a,b)

instance Eq Bool

instance Eq Int.

How:

```
instance Eq a ⇒ Eq [a] where
  (==) :: [a] → [a] → Bool
  (==) x:xs  y:ys  =  (x == y) && (xs == ys)
  (==) []    []    =  True
  (==) _     _     =  False
```

23-2-24

```
class Eq a where
  (==) :: a → a → Bool

Instance Eq Bool where
{- (==) :: Bool → Bool → Bool -}
  (==) True  True  = True
  (==) False False = True
  _    _     = _
  _    _     = _
```

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    (/=) x y = not (x == y)   -- default defn.
    (==) x y = not (x /= y)   -- default defn.
```

-- If the default definition is not given then in the instance we have to give definition of both (==) and (/=)

-- ~~to~~ to override default definiton, we ~~can~~ can define (/=) again in the instance.

-- default definition enables us to get away with only defining either of (==) or (/=)

```
Hash is a type
instance Eq Hash.
hash :: String -> Hash

data HString = HString Hash String
```

-- if there is only one constructor, then we can use the same name as datatype

fromString :: String → Hstring

fromString str = Hstring (hash str) string

toString :: Hstring → String

~~toString hstr =~~

toString (Hstring hash str) = str

Instance Eq Hstring where

~~(==) :: Hstring~~

~~(==) hs1 hs2 =~~

(==) (Hstring h1 s1) (Hstring h2 s2) = (h1 == h2) && (s1 == s2)

(/=) (Hstring h1 s1) (Hstring h2 s2) = (h1 /= h2) || (s1 /= s2)

26-2-24

~~class Eq a class~~

class Eq a where

(==) :: a → a → ~~Equal~~ Bool

Instance Eq Bool where

Instance (Eq a, Eq b) ⇒ Eq (a, b) where

Instance Eq a ⇒ Eq [a] where

foo x y = if x == y then "equal"
else "not equal"

foo :: Eq ⇒ a → a → string

foo (x1, y1), z = if (x1, y1) = z then "equal"
else "not equal"

(Eq a1, Eq a2) ⇒ (a1, a2) → (a1, a2) → String

Class Ord

class Ord a where
(<=) :: a → a → Bool

class Eq ⇒ Ord a where
(<=) :: a → a → Bool.

ghci

```
instance Ord Boo where
-- (<=) :: Bool -> Bool -> Bool
(<=) : True False
```

```
instance Ord where
(<=)
```

```
instance Ord where
```

```
instance Ord Boo where
-- (<=) : Bool -> bool -> Bool
```

```
(<=) _ True = True
(<=)
```

```
instance Ord Boo where
(<=) True _ = False
(<=) _ _ = True
```

```
instance Ord (a,b) where
(<=) (a,b) = a <=
(<=) (x1,y1) (x2,y2) = x1 <= x2 ||
instance (Ord a, Ord b) => Ord (a,b) where
(<=) (x1,y1) (x2,y2) | x1 <= x2 = True
                     | x1 == x2 = y1 <= y2
                     | otherwise
```

instance (Ord a, Ord b) => Ord (a, b) where

(<=) (x1, y1) (x2, y2) | x1 == x2 = y1 <= y2

| x1 <= x2 = True

| otherwise = False

instance Ord a => Ord [a] where

(<=) (x :: xs) (y :: ys) | not (x == y) = xs <= y2

| x <= y = True

| otherwise = False

(<=) [] [] = True

(<=) [] _ = ~~False~~ True

(<=) _ _ = False.

~~So~~

~~sort :: Ord a => a -> a -> Bool~~

~~sort :: (a -> a -> Bool) -> a -> a~~

sort :: Ord a => [a] -> [a]

sort :: (a -> a -> Bool) -> [a] -> [a]