I

Functional programming: user enters expressions and the system evaluates those expressions.

vb. Module: (Haskell)

```
square :: Integer -> Integer
square x = x * x
```

no punctuation

Evaluation: replacing equals by equals

```
square (3 + 4)
   ↓
square ( 7 )
   ↓
7 . 7
   ↓
49
```

Expressions that can't be reduced any futher

↓

normal form

Values: in FP, the sole purpose of an expression is to denote a value

↓

Evaluator prints the canonical representation for a value (42)

↳ have no " " functions

Functions: functions transform one or more arguments into a result.

always deterministic
sometimes partial

eg. f :: Integer -> Integer

Application:

f x (f applied to x)
f(x + 3)
↑
operator in function parameter

Lambda expressions: notation for anonymous functions

\x -> x * x equals square

Haskell supports two different styles:

Declaration style:

eg. quad :: Integer -> Integer
quad x = square x * square x

Expression style:

eg. quad :: Integer -> Integer
quad = \x -> square x * square x

Two functions are equal if they give an equal outcome for all arguments.

Prefix: f 4 3 → infix: 4 `f` 3
Infix: 4 + 3 → Prefix: (+) 4 3

Composition:

$$(g \circ h)\, x = g(h\, x)$$

Typing Defining values:

```
name :: String
name = "Ralf"
```

Conditional definitions:

```
smaller :: (Integer, Integer) -> Integer
smaller (x,y) = if x ≤ y then x else y
```

or

```
smaller :: (Integer, Integer) -> Integer
smaller (x,y)
  | x ≤ y   = x
  | otherwise = y
```

Wildcards: _ acts as wildcard

```
Eg.  day :: Integer -> String
     day 1 = "Sunday"
     day 2 = "Saturday"
     day _ = "Weekday"
```

↑ evaluated in order untill a proper solution is found

Local definitions

where can be applied for short declarations:

```
Eg.  demo :: Integer -> Integer -> Integer
     demo x y = (a+1) * (b+2)
       where  a = x - y
              b = x + y
```

This can also be replaced by let ... in :

```
Eg.  demo :: Integer -> Integer -> Integer
     demo x y = let a = x - y
                    b = x + y
                in (a+1) * (b+2)
```

II

Haskell is strongly typed : every expression has a unique type

Haskell is statically typed : every type checking is done before runtime

Booleans : True or False

Operators :

&& : and
|| : or

Comparison :

== : is
=/= : is not
< : smaller than
...

Solving a function with Bool :

```
f :: Bool -> S

f False = ...      (solve for False)
f True  = ...      (solve for True)
```

Characters : type Char

↓

written in single quotes 'a'

→ can be converted to Int and back with:

```
ord :: Char -> Int
chr :: Int -> Char
```

Strings : type String

↓

Constants in double quotes "Hello"

Concatination with ++

Numbers : Ord
- fixed precision : Int
- arbitrary precision : Integer
- single- and double precision floats : Float, Double

Normal math operations can be done on these types.

↓

are overloaded

Enumerations: declaration of new types

Eg. data Day = Mon | Tue | Wed | ...

Functions 4

X -> Y -> Z o = X -> (Y -> Z)
\x y -> z = \x -> \y -> z
f x y = \x -> (\y -> z)
= (f x) y

Tuples: pairing types

Eg. (Char, Integer)

↓

order matters (can also be triples or nested)

Polymorphic functions are defined on any type:

Eg. fst :: (a,b) -> a

↓

type variables

Type synonyms: alternative names for types

↓

for documentation or clarity

- type Card = (Rank, Suit)

- Types can't be recursive

Type classes: a type class can contain several types

↓

For defining class specific functions (such as +)

- Num is a set of numeric types such as Integer, Float, etc.

III

List notation:

[1,2,3] -> elements seperated by ","

Strings are lists of chars:

"Hello" = ['H', 'e', 'l', 'l', 'o']

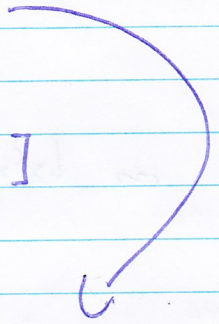
Elements can be of any type

Functions

vb. Remove newlines:

```
unwrap :: String -> String
unwrap = concat ∘ lines
```

```
lines  :: String -> [String]
concat :: [String] -> String
```



∘ 1 ∘ 2 : 1 after 2

↳ 2 gets executed first

List:

- empty []
- or consists of an element followed by a list x:xs

: and [] are called constructors

Defining functions over lists:

Two cases: `[]` and `[_:_]`

```
null :: [a] -> Bool
null []     = True
null (_:_)  = False
```

Case expression:

```
null :: [a] -> Bool
null xs = case xs of
            []    -> True
            (_:_) -> False
```

Operators on lists:
- map
- filter

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

filter :: (a -> Bool) -> ([a] -> [a])
filter _p []    = []
filter p (x:xs)
  | p x         = x : filter p xs
  | otherwise   = filter p xs
```

Comprehensions: list generation inside list maker

vb. [square x | x ∈ [1 ... 5], odd x]

generator is of the form: - x ⟵ xs
- boolean expression

IV

Product data types:

vb. type Name = String
type Age = Int
data Person = P Name Age

then P :: Name -> Age -> Person

Extracting data from a constructor function is easy:

vb. showPerson :: Person -> String
showPerson (P n a) = "Name = " ++ n ++ ", Age = " ++ show a

↓

Safe than type synonyms: type Person = (Name, Age)

Data types can have variants:

vb. data Suit = Spades | Hearts | Diamonds | Clubs
data Rank = Faceless Integer | Jack | Queen | King
data Card = Card Rank Suit | Joker

↓

Function

Parametric datatypes: data types may be parametric

↓

Constructors are then functions:

vb. `data Maybe a = Nothing | Just a`

then: `Just 13 :: Maybe Int`
`Nothing :: Maybe a, Just :: a -> Maybe a`

Datatypes can be recursive too:

`data Expr = Lit Integer | Add Expr Expr | Mul Expr Expr`

Constructor names may be operators starting with `:`

vb.
```
infixl 6 :+
infixl 7 :*

data Expr
  = Lit Integer
  | Expr :+ Expr
  | Expr :* Expr
  deriving (Show)
```

Then we can construct expressions like this:

```
expr :: Expr
expr = (Lit 4 :* Lit 7) :+ (Lit 11)
```

Design pattern for recursive data types

vb `f :: Expr -> S`

step 1: solve the problem for Literals (non-recursive options)

`f (Lit n) = ... n ...`

step 2: solve the problem for each recursive option:

`f (x :+ y) = ... x ... y ... f x ... f y`
`f (x :* y) = ... x ... y ... f x ... f y`

V

Functional programming is all about functions

↓

have all the rights of other types

- may be passed as arguments
- may be returned as result
- may be stored in data structures

Higher order functions: functions that manipulate functions
↳ (map or filter)

Function as argument:

```
quickSortBy :: (a -> a -> Bool) -> ([a] -> [a])
...
```

Functions as results:

```
data Op = Add | Sub | Mul | Div
op :: Op -> (Integer -> Integer -> Integer)
op Add = (+)
op Sub = (-)
op Mul = (*)
op Div = div
```

Partial application:

```
add' x y = x+y
```

type: Integer -> Integer -> Integer

↓

but this is a shorthand notation for:

Integer -> (Integer -> Integer)

→ The function thus takes an Integer and returns a function

↓

Partial application: we don't need to apply a function to all its arguments at once

In Haskell, every function takes <u>exactly one</u> argument.

↑

Currying

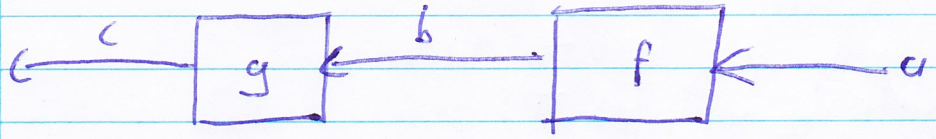
Transformation:

```
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f a b = f (a,b)

uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry f (a,b) = f a b
```

Composition: two functions can be glued together with ∘

$(g \circ f)\ a$:



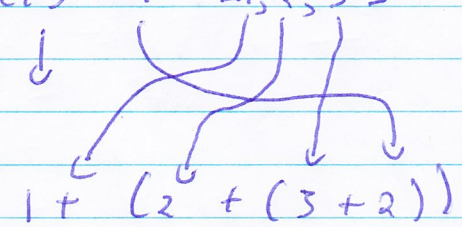
Fold right: (foldr)

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

Applies the function $(a \to b \to b)$ to first to its second element and the last element on the list and then to all next ones (from right to left)

```
foldr (▷) e = recurse
    where recurse [] = e
          recurse (x:xs) = x ▷ recurse xs
```

```
foldr (+) 2 [1,2,3]
```



Fold left: (foldl) → left to right computation

Scanl : foldl but applied to every initial segment:

scanl (◁) e [x, y, z] = [e, e◁x, (e◁x)◁y, ...]

→ also a scanr

VI

Overloading: using the same name for different but related functions.

↓

Type classes declare a group of identifiers as overloaded

Vb.
```
class Eq a where
(==) :: a -> a -> Bool
(=/=) :: a -> a -> Bool
```

↳ in the Eq class there exist a method == (or =/=) and for this class the type of this method is a -> a -> Bool

If an overloaded operator is used in another function, this function is now overloaded as well.

Instances of type classes have to be declared explicitly:

Vb.
```
data Gende = Female | Male
instance Eq Gende where
    Female == Female = True
    Female == Male   = False
    :
    x =/= y = not (x == y)
```

Classes can also be extended:

vb
```
class (Eq a) => Ord a where
    Compare :: a -> a -> Ordering
    :
```

Ord is a subclass of Eq

Monoid: function which is associative and has a unit element

```
class Monoid a where
    ε :: a
    (•) :: a -> a -> a
```

↓

$\epsilon \bullet x = x$

$x \bullet \epsilon = x$

$(x \bullet y) \bullet z = x \bullet (y \bullet z)$

We can now create + with •:

```
instance Monoid Additive where
    ε = Sum 0
    x • y = Sum (fromSum x + fromSum y)
```

Sequential evaluation of polynomials

A polynomial can be represented by a list of coefficients:

$[4,7,1,1] = 4 + 7x + x^2 + x^3$

We can create a function:

```
evaluate :: Integer -> [Integer] -> Integer
evaluate x = foldr (\a v -> a + x * v) 0
```

evaluate 2 [4,7,1,1] = 30

We can also evaluate this in parallel:

$$p(x) = x^0 \cdot (4+7 \cdot x) + x^2 \cdot (1+1 \cdot x)$$

Ub.
```
data Pdg = Pdg Integer Integer
  deriving (Show)

instance Monoid Pdg where
  ε = Pdg 1 0
  Pdg x u • Pdg y v = Pdg (x*y) (u+x*v)

evaluate :: Integer -> [Integer] -> Integer
evaluate x = reduce ∘ map (\a -> Pdg x a)

reduce :: (Monoid m) => [m] -> m
reduce [] = ε
reduce x:xs = x • reduce xs
```

Mapping functions:

Ub.
```
map :: (a -> b) -> ([a] -> [b])
map f [] = []
map f (x:xs) = fx : map f x
```

VII

We can prove that Haskell programs work by substitution and induction

Equational reasoning

Given:

curry f a b = f (a, b)
fst (a, b) = a
const a b = a

Prove that curry fst = const

```
    curry fst x y
=   { definition of curry }
    fst (x, y)
=   { definition of fst }
    x
=   { definition of const }
    const x y
```

Programming over recursive types usually require recursion:

```
sum []     = 0
sum (x:xs) = x + sum xs
```

} → explicit recursion

or

```
sum = foldr (+) 0
```

} → canned recursion

Proofs by induction:

Show that property P(xs) holds for every list xs

1. Show that the property holds for the empty list:

   P([]) → base case

2. Show that the property holds for non-empty lists P(x:xs), assume you already have established P(xs).

   P(x:xs) ⟸ P(xs) → inductive case

vb.

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

1. [] ++ ys = ys holds by definition

2. To prove (x:xs) ++ ys = x:(xs ++ ys), we use induction over xs.

2.1 Case xs = []

```
[] ++ []
=   {definition of ++}
[]
```

2.2 Case $xs = a:as$

$(a:as) ++ []$

$= \{\text{definition of } ++\}$

$a:(as ++ [])$

$= \{\text{induction assumption : } as ++ [] = as\}$

$a:as$

To prove that $++$ is associative, we use induction over xs, $(xs ++ (ys ++ zs)) = (xs ++ ys) ++ zs$

1. Base case $xs = []$

~~$[] ++ (ys ++ zs)$~~

$[] ++ (ys ++ zs)$

$= \{\text{definition of } ++\}$

$ys ++ zs$

$= \{\text{definition of } ++\}$

$([] ++ ys) ++ zs$

2 Inductive step, $xs = a : as$

$(a : as) ++ (ys ++ zs)$

$= \{$definition of $++\}$

$a : (as ++ (ys ++ zs))$

$= \{IH : as ++ (ys ++ zs) = (as ++ ys) ++ zs$

$a : ((as ++ ys) ++ zs)$

$= \{$definition of $++\}$

$(a : (as ++ ys)) ++ zs$

$= \{$definition of $++\}$

$((a : as) ++ ys) ++ zs$

Every recursive datatype comes with a pattern of induction.

For literals, the base case is $P(Lit\ n)$

The inductive steps a-e:

$P(x :+: y) \Leftarrow P(x) \wedge P(y)$

$P(x :*: y) \Leftarrow P_x \wedge P(y)$

Correctness of expression compile

push (evaluate e) = execute (compile e)

We can proof this by induction over the structure of e

Base case: e = Lit i

push (evaluate (Lit i))
{ definition of evaluate }
push i
{ definition of execute }
execute (Push i)
{ definition of compile }
execute (compile (Lit i))

Inductive step: $e = e1 :+: e2$

push (evaluate (e1 :+: e2))
{definition of evaluate}
push (evaluate e1 + evaluate e2)
{def property of add}
add ∘ push (evaluate e2) ∘ push (evaluate e1)
{IH : push (evaluate ei) = execute (compile ei)}
add ∘ execute (compile e2) ∘ execute (compile e1)
{definition of execute}
execute (compile e1 :^: compile e2 :^: Add)
{definition of compile}
execute (compile (e1 :+: e2))

Likewise for $e1 :*: e2$

We can also derive programs from their specifications with this prove method.

Efficient reverse:

The naive definition of reverse was:

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

→ This implementation is slow because ++ traverses over its first argument.

We need to eliminate ++, therefor we define:

reverseCat xs ys = reverse xs ++ ys

If reverseCat is efficient, the reverse:

reverse xs = reverseCat xs []

We have two cases, a base case ($xs = []$) and an inductive case ($xs = a : as$)

Case: $xs = []$

```
reverseCat [] ys
    {specification of reverseCat}
reverse [] ++ ys
    {definition of reverse}
[] ++ ys
    {definition of ++}
ys
```

So:

```
reverseCat [] ys = ys
```

Case $xs = a : as$

```
reverseCat (a:as) ys
    {specification of reverseCat}
reverse (a:as) ++ ys
    {definition of reverse}
(reverse as ++ [a]) ++ ys
    {monoid}
(reverse as) ++ ([a] ++ ys)
    {definition of ++}
reverse as ++ a:ys
    {specification of reverseCat}
reverseCat as (a:ys)
```

So:

```
reverseCat (a:as) ys = reverseCat as (a:ys)
```

Use of program schemes improves modularity.
↳ applies also to proving

Fusion: Frequent rewrite, reordering
↳ pushing a function inside

$$f(x_1 \triangleright (x_2 \triangleright (x_3 \triangleright e))) = x_1 \triangleright (x_2 \triangleright (x_3 \triangleright f\, e))$$

Captured by the fusion law:

$$f(a \triangleright b) = a \triangleright f\, b$$

Pointwise notation: $f(g\ x)$
Pointfree notation: $f \circ g$