1 Introduction on purity and evaluation

2 Basic Haskell

3 More advanced concepts
We will consider now some basic concepts of Haskell, by implementing them in Scheme:

1. What is a pure functional language?
2. Non-strict evaluation strategies
3. Currying
What is a **functional** language?

1. In mathematics, **functions** do not have **side-effects**
2. e.g. if $f : \mathbb{N} \to \mathbb{N}$, $f(5)$ is a fixed value in $\mathbb{N}$, and do not depend on *time* (also called **referential transparency**)
3. this is clearly not true in conventional programming languages, Scheme included
4. Scheme is *mainly* functional, as programs are **expressions**, and computation is evaluation of such expressions
5. but some expressions have **side-effects**, e.g. `vector-set!`
6. Haskell is **pure**, so we will see later how to manage inherently side-effectful computations (e.g. those with I/O)
We have already seen that, in \textbf{absence of side effects} (purely functional computations) from the point of view of the result the \textbf{order} in which functions are applied \textbf{does not matter} (almost).

However, it matters in other aspects, consider e.g. this function:

\begin{verbatim}
(define (sum-square x y)
  (+ (* x x)
      (* y y)))
\end{verbatim}
A possible evaluation:

```
(sum-square (+ 1 2) (+ 2 3))
;; applying the first +
= (sum-square 3 (+ 2 3))
;; applying +
= (sum-square 3 5)
;; applying sum-square
= (+ (* 3 3)(* 5 5))
  ...
= 34
```

is it that of Scheme?
(sum-square (+ 1 2) (+ 2 3))

;; applying sum-square
= (+ (* (+ 1 2)(+ 1 2))(* (+ 2 3)(+ 2 3)))

;; evaluating the first (+ 1 2)
= (+ (* 3 (+ 1 2))(* (+ 2 3)(+ 2 3)))

...  
= (+ (* 3 3)(* 5 5))

...  
= 34

1. The two evaluations differ in the order in which function applications are evaluated.

2. A function application ready to be performed is called a reducible expression (or redex)
in the first example of evaluation of mult, redexes are evaluated according to a (leftmost) **innermost strategy**

i.e., when there is more than one redex, the leftmost one that does not contain other redexes is evaluated

e.g. in `((sum-square (+ 1 2) (+ 2 3)))` there are 3 redexes: `((sum-square (+ 1 2) (+ 2 3)))`, `(+ 1 2)` and `(+ 2 3)` the innermost that is also leftmost is `(+ 1 2)`, which is applied, giving expression `((sum-square 3 (+ 2 3)))`

in this strategy, **arguments** of functions are always evaluated **before** evaluating the function itself - this corresponds to passing arguments **by value**.

note that Scheme does not require that we take the **leftmost**, but this is very common in mainstream languages
Evaluation strategies: call-by-name

1. a dual evaluation strategy: redexes are evaluated in an **outermost** fashion.

2. we start with the redex that is **not contained in any other redex**, i.e. in the example above, with \((\text{sum-square}\ (\text{+} \ 1 \ 2)\ (\text{+} \ 2 \ 3))\), which yields \((\text{+} \ (\text{*} \ (\text{+} \ 1 \ 2)\ (\text{+} \ 1 \ 2))\ (\text{*} \ (\text{+} \ 2 \ 3)\ (\text{+} \ 2 \ 3)))\).

3. in the outermost strategy, functions are always **applied before their arguments**, this corresponds to passing arguments **by name** (like in Algol 60).
Termination and call-by-name

1. e.g. first we define the following two simple functions:

```
(define (infinity)
  (+ 1 (infinity)))

(define (fst x y) x)
```

2. consider the expression (fst 3 (infinity)):
   1. Call-by-value: (fst 3 (infinity)) = (fst 3 (+ 1 (infinity))) = (fst 3 (+ 1 (+ 1 (infinity)))) = ...
   2. Call-by-name: (fst 3 (infinity)) = 3

3. if there is an evaluation for an expression that terminates, **call-by-name terminates**, and produces the same result (Church-Rosser confluence)
Haskell is lazy: call-by-need

1. In call-by-name, if the argument is not used, it is never evaluated; if the argument is used several times, it is re-evaluated each time.
2. Call-by-need is a memoized version of call-by-name where, if the function argument is evaluated, that value is stored for subsequent uses.
3. In a “pure” (effect-free) setting, this produces the same results as call-by-name, and it is usually faster.
we saw that macros are different from function, as they do not evaluate and are expanded at **compile time**

a possible idea to overcome the nontermination of \((\text{fst} \ 3 \ (\text{infinity}))\), could be to use **thunks** to prevent evaluation, and then **force** it with an explicit call

indeed, there is already an implementation in Racket based on **delay** and **force**

we’ll see how to implement them with macros and thunks
Delay is used to return a **promise** to execute a computation (implements **call-by-name**)  

moreover, it caches the result (**memoization**) of the computation on its first evaluation and returns that value on subsequent calls (implements **call-by-need**
(struct promise
   (proc ; thunk or value
       value? ; already evaluated?
    ) #:mutable)
(define-syntax delay
  (syntax-rules ()
    ((_ (expr ...))
     (promise (lambda ()
                  (expr ...)) ; a thunk
                  #f))))) ; still to be evaluated
**force** is used to force the evaluation of a promise:

```scheme
(define (force prom)
  (cond
   ; is it already a value?
   ((not (promise? prom)) prom)
   ; is it an evaluated promise?
   ((promise-value? prom) (promise-proc prom))
   (else
    (set-promise-proc! prom ((promise-proc prom)))
    (set-promise-value?! prom #t)
    (promise-proc prom))))
```
Examples

```
(define x (delay (+ 2 5))) ; a promise
(force x) ;; => 7

(define lazy-infinity (delay (infinity)))
(force (fst 3 lazy-infinity)) ; => 3
(fst 3 lazy-infinity) ; => 3
(force (delay (fst 3 lazy-infinity))) ; => 3
```

1. here we have call-by-need only if we make every function call a promise
2. in Haskell call-by-need is the default: if we need call-by-value, we need to `force` the evaluation (we’ll see how)
Currying

1. in Haskell, functions have only **one** argument!
2. this is not a limitation, because functions with more arguments are **curried**
3. we see here in Scheme what it means. Consider the function:

   ```scheme
   (define (sum-square x y)
     (+ (* x x)
        (* y y)))
   ```

4. it has signature *sum-square* : \( \mathbb{C}^2 \rightarrow \mathbb{C} \), if we consider the most general kind of numbers in Scheme, i.e. the complex field
Currying (cont.)

1. curried version:

```
(define (sum-square x)
  (lambda (y)
    (+ (* x x)
      (* y y)))))

;; shorter version:
(define ((sum-square x) y)
  (+ (* x x)
    (* y y)))
```

2. it can be used *almost* as the usual version: `((sum-square 3) 5)`

3. the curried version has signature `sum-square : \mathbb{C} \rightarrow (\mathbb{C} \rightarrow \mathbb{C})` i.e. `\mathbb{C} \rightarrow \mathbb{C} \rightarrow \mathbb{C}` (→ is right associative)
Currying in Haskell

1. in Haskell every function is automatically curried and consequently managed
2. the name *currying*, coined by Christopher Strachey in 1967, is a reference to logician Haskell Curry
3. the alternative name *Schönfinkelisation* has been proposed as a reference to Moses Schönfinkel but didn’t catch on
Born in 1990, designed by committee to be:

1. purely functional
2. call-by-need (sometimes called lazy evaluation)
3. strong polymorphic and static typing

Standards: Haskell ’98 and ’10

Motto: "Avoid success at all costs"

ex. usage: Google’s Ganeti cluster virtual server management tool

I mainly follow Hudak, Peterson, Fasel, A Gentle Introduction to Haskell 98, 1999

Beware! There are many bad tutorials on Haskell and monads, in particular, available online
A taste of Haskell’s syntax

- more complex and "human" than Scheme: parentheses are optional!
- function call is similar, though: \( f \ x \ y \) stands for \( f(x,y) \)
- there are infix operators and are made of non-alphabetic characters (e.g. *, +, but also <++>)
- \( \text{elem} \) is \( \in \). If you want to use it infix, just use \( \text{\textasciitilde elem} \)
- \(--\) this is a comment
- lambdas: \( \lambda (x \ y) (+ \ 1 \ x \ y) \) is written \( \lambda x \ y \rightarrow 1+x+y \)
Haskell has **static** typing, i.e. the type of everything must be known at **compile time**.

There is **type inference**, so usually we do not need to explicitly declare types.

*has type* is written :: instead of : (the latter is **cons**).

E.g.,

1. 5 :: Integer
2. 'a' :: Char
3. inc :: Integer -> Integer
4. [1, 2, 3] :: [Integer] – equivalent to 1:(2:(3:[]))
5. ('b', 4) :: (Char, Integer)
6. Strings are **lists of characters**.
1. functions are declared through a sequence of *equations* 

2. e.g.

```
inc n = n + 1

length :: [Integer] -> Integer
length [] = 0
length (x:xs) = 1 + length xs
```

3. this is also an example of *pattern matching*

4. arguments are matched with the right parts of equations, top to bottom

5. if the match succeeds, the function body is called
the previous definition of `length` could work with any kind of lists, not just those made of integers

indeed, if we omit its type declaration, it is inferred by Haskell as having type

```
length :: [a] -> Integer
```

lower case letters are **type variables**, so `[a]` stands for a list of elements of type `a`, for any `a`
Main characteristics of Haskell’s type system

1. every well-typed expression is guaranteed to have a **unique principal type**
   - it is (roughly) the *least general type that contains all the instances of the expression*
   - e.g. `length :: a -> Integer` is too general, while `length :: [Integer] -> a` is too specific

2. Haskell adopts a variant of the **Hindley-Milner** type system (used also in ML variants, e.g. F#)

3. and the principal type can be **inferred automatically**

User-defined types

1. are based on **data declarations**

   ```haskell
   -- a "sum" type (union in C)
   data Bool = False | True
   ```

2. **Bool** is the (nullary) **type constructor**, while **False** and **True** are **data constructors** (nullary as well)

3. data and type constructors live in separate name-spaces, so it is possible (and common) to use the same name for both:

   ```haskell
   -- a "product" type (struct in C)
   data Pnt a = Pnt a a
   ```

4. if we apply a data constructor we obtain a **value** (e.g. Pnt 2.3 5.7), while with a type constructor we obtain a **type** (e.g. Pnt Bool)
Recursive types

1. classical recursive type example:

```haskell
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

2. e.g. data constructor Branch has type:

```haskell
Branch :: Tree a -> Tree a -> Tree a
```

3. An example tree:

```haskell
aTree = Branch (Leaf 'a')
  (Branch (Leaf 'b') (Leaf 'c'))
```

4. in this case aTree has type Tree Char
Of course, also lists are recursive. Using Scheme jargon, they could be defined by:

```haskell
data List a = Null | Cons a (List a)
```

but Haskell has special syntax for them; in "pseudo-Haskell":

```haskell
data [a] = [] | a : [a]
```

[] is a data and type constructor, while : is an infix data constructor.
An example function on Trees

fringe :: Tree a -> [a]
fringe (Leaf x) = [x]
fringe (Branch left right) = fringe left ++ fringe right

1 (++) denotes list concatenation, what is its type?
as we saw, *product types* (e.g. `data Point = Point Float Float`) are like `struct` in C or in Scheme (analogously, *sum types* are like `union`)

the access is positional, for instance we may define accessors:

```haskell
  pointx Point x _ = x
  pointy Point _ y = y
```

there is a C-like syntax to have *named fields*:

```haskell
  data Point = Point {pointx, pointy :: Float}
```

this declaration automatically defines two field names `pointx`, `pointy`

and their corresponding *selector functions*
Type synonyms

1. are defined with the keyword `type`
2. some examples

```haskell
type String = [Char]

type Assoc a b = [(a,b)]
```
3. usually for readability or shortness
newtype is used when we want to define a type with the same representation and behavior of an existing type (like type) but having a separate identity in the type system (e.g. we want to define a kind of string $\neq$ [Char])

e.g.

newtype Str = Str [Char]

note: we need to define a data constructor, to distinguish it from String

its data constructor is not lazy (difference with data)
Haskell has **map**, and it can be defined as:

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

we can partially apply also infix operators, by using parentheses: (+ 1) or (1 +) or (+)

```
map (1 +) [1,2,3]  -- =>  [2,3,4]
```
1. `:t` at the prompt is used for getting type, e.g.

```
Prelude> :t (+1)
(+1) :: Num a => a -> a
Prelude> :t +
<interactive>:1:1: parse error on input ‘+’
Prelude> :t (+)
(+) :: Num a => a -> a -> a
```
Function composition and $\cdot$

1. $(\cdot)$ is used for composing functions (i.e. $(f \cdot g)(x)$ is $f(g(x))$)

   Prelude> let dd = (*2) . (1+)
   Prelude> dd 6
   14
   Prelude> :t (\cdot)
   (\cdot) :: (b -> c) -> (a -> b) -> a -> c

2. $\cdot$ syntax for avoiding parentheses, e.g. $(10\cdot)(5+3) = (10\cdot)\cdot5+3$
Infinite computations

1. call-by-need is very convenient for dealing with never-ending computations that provide data

2. here are some simple example functions:

   ```
   ones = 1 : ones
   
   numsFrom n = n : numsFrom (n+1)
   
   squares = map (^2) (numsFrom 0)
   ```

3. clearly, we cannot evaluate them (why?), but there is take to get finite slices from them

4. e.g.

   ```
   take 5 squares = [0,1,4,9,16]
   ```
Infinite lists

1. Convenient syntax for creating infinite lists:
   e.g. `ones` before can be also written as `[1,1..]`

2. `numsFrom 6` is the same as `[6..]`

3. `zip` is a useful function having type `zip :: [a] -> [b] -> [(a, b)]`.
   ```
   zip [1,2,3] "ciao"
   -- => [(1,'c'),(2,'i'),(3,'a')]
   ```

4. List comprehensions
   ```
   [(x,y) | x <- [1,2], y <- "ciao"]
   -- => [(1,'c'),(1,'i'),(1,'a'),(1,'o'),
   (2,'c'),(2,'i'),(2,'a'),(2,'o')]
   ```
Infinite lists (cont.)

1 a list with all the Fibonacci numbers (note: tail is cdr, while head is car)

\[
\text{fib} = 1 : 1 : \\
[a + b \mid (a, b) \leftarrow \text{zip} \ \text{fib} \ \text{tail} \ \text{fib}] 
\]
1. **bottom** (aka \( \bot \)) is defined as \( \text{bot} = \text{bot} \)
2. all errors have value \( \text{bot} \), a value shared by all types
3. error :: String -> a is strange because it is polymorphic only in the output
4. the reason is that it returns **bot** (in practice, an exception is raised)
the matching process proceeds top-down, left-to-right

patterns may have **boolean guards**

\[
\text{sign } x \begin{cases} 
  x > 0 & = 1 \\
  x = 0 & = 0 \\
  x < 0 & = -1 
\end{cases}
\]

_ stands for *don’t care*

e.g. definition of **take**

\[
\begin{align*}
\text{take } 0 & \_ = [] \\
\text{take } \_ & [ ] = [] \\
\text{take } n (x:xs) & = x : \text{take } (n-1) xs
\end{align*}
\]
the order of definitions **matters**:

```haskell
Prelude> :t bot
bot :: t
Prelude> take 0 bot
[]
```

2. on the other hand, `take bot []` does not terminate

3. what does it change, if we swap the first two defining equations?
take with case:

\[
\text{take } m \text{ ys} = \text{case } (m,\text{ys}) \text{ of }
\]
\[
(0,\_\,) \rightarrow []
\]
\[
(_,[][]) \rightarrow []
\]
\[
(n,x:xs) \rightarrow x : \text{take } (n-1) \text{ xs}
\]
let and where

1. **let** is like Scheme’s letrec*:

   ```
   let x = 3
       y = 12
   in x+y  -- => 15
   ```

2. **where** can be convenient to scope binding over equations, e.g.:

   ```
   powset set = powset’ set [][] where
       powset’ [] out = out
       powset’ (e:set) out = powset’ set (out ++
               [ e:x | x <- out ])
   ```

3. Layout is like in Python, with meaningful whitespaces, but we can also use a C-like syntax:

   ```
   let {x = 3 ; y = 12} in x+y
   ```
Call-by-need and memory usage

1. **fold-left** is efficient in Scheme, because its definition is naturally tail-recursive:

   \[
   \begin{align*}
   \text{foldl } f & \quad z \quad [] \quad = \quad z \\
   \text{foldl } f & \quad z \quad (x:xs) \quad = \quad \text{foldl } f \quad (f \quad z \quad x) \quad xs
   \end{align*}
   \]

2. *note: in Racket it is defined with* \((f \times z)\)

3. this is not as efficient in Haskell, because of call-by-need:

   1. \(\text{foldl } (+) \quad 0 \quad [1,2,3]\)
   2. \(\text{foldl } (+) \quad (0 + 1) \quad [2,3]\)
   3. \(\text{foldl } (+) \quad ((0 + 1) + 2) \quad [3]\)
   4. \(\text{foldl } (+) \quad (((0 + 1) + 2) + 3) \quad []\)
   5. \(((0 + 1) + 2) + 3) = 6\)
There are various ways to enforce **strictness** in Haskell (analogously there are classical approaches to introduce laziness in strict languages)

e.g. on data with **bang patterns** (a datum marked with ! is considered **strict**)

```haskell
data Complex = Complex !Float !Float
```

there are extensions for using ! also in function parameters
Forcing evaluation

1. Canonical operator to **force evaluation** is \( \text{seq} :: \text{a} \rightarrow \text{t} \rightarrow \text{t} \)
2. \( \text{seq} \ x \ y \) returns \( y \), **only if** the evaluation of \( x \) **terminates** (i.e. it performs \( x \) then returns \( y \))
3. a strict version of **foldl** (available in *Data.List*):

   - \( \text{foldl}' \ f \ z \ [] = z \)
   - \( \text{foldl}' \ f \ z \ (x:xs) = \text{let} \ z' = f \ z \ x \)
     in \( \text{seq} \ z' \ (\text{foldl}' \ f \ z' \ xs) \)
4. strict versions of standard functions are usually primed
There is a convenient \textit{strict} variant of $\function$ (function application) called $!$

here is its definition:

\[
\begin{align*}
(\!\!\!) & :: (a \to b) \to a \to b \\
f \!\!\! x &= \text{seq} \ x \ (f \ x)
\end{align*}
\]
not much to be said: Haskell has a simple module system, with `import`, `export` and namespaces

a very simple example

```haskell
module CartProd where  --- export everything
infixr 9 -*-
-- right associative
-- precedence goes from 0 to 9, the strongest
x -*- y = [(i,j) | i <- x, j <- y]
```
import/export

```haskell
module Tree (Tree (Leaf, Branch), fringe) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)

fringe :: Tree a -> [a] ...
```

```haskell
module Main (main) where

import Tree (Tree (Leaf, Branch))

main = print (Branch (Leaf 'a') (Leaf 'b'))
```
1. modules provide the only way to build abstract data types (ADT)
2. the characteristic feature of an ADT is that the representation type is hidden: all operations on the ADT are done at an abstract level which does not depend on the representation
3. e.g. a suitable ADT for binary trees might include the following operations:

```haskell
data Tree a  -- just the type name
leaf        :: a -> Tree a
branch      :: Tree a -> Tree a -> Tree a
cell        :: Tree a -> a
left, right :: Tree a -> Tree a
isLeaf      :: Tree a -> Bool
```
module TreeADT (Tree, leaf, branch, cell, left, right, isLeaf) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)
leaf = Leaf
branch = Branch
cell (Leaf a) = a
left (Branch l r) = l
right (Branch l r) = r
isLeaf (Leaf _) = True
isLeaf _ = False

1 in the export list the type name Tree appears without its constructors
1 so the only way to build or take apart trees outside of the module is by using
the various (abstract) operations
2 the advantage of this information hiding is that at a later time we could
change the representation type without affecting users of the type
we already saw *parametric polymorphism* in Haskell (e.g. in `length`)

**Type classes** are the mechanism provided by Haskell for *ad hoc* polymorphism (aka *overloading*).

the first, natural example is that of numbers: `6` can represent an integer, a rational, a floating point number...

e.g.

```haskell
Prelude> 6 :: Float
6.0
Prelude> 6 :: Integer -- unlimited
6
Prelude> 6 :: Int -- fixed precision
6
Prelude> 6 :: Rational
6 % 1
```
also numeric operators and equality work with different kinds of numbers

let’s start with equality: it is natural to define equality for many types (but not every one, e.g. functions - it’s undecidable)

we consider here only **value equality**, not **pointer equality** (like Java’s `==` or Scheme’s `eq?`), because pointer equality is clearly **not referentially transparent**

let us consider **elem**

\[
\begin{align*}
x \ `\text{elem}` \ [ & ] & = \text{False} \\
x \ `\text{elem}` \ (y:ys) & = x==y \ || \ (x \ `\text{elem}` \ ys)
\end{align*}
\]

its type should be: \(a \rightarrow [a] \rightarrow \text{Bool}\). But this means that \(==\) :: \(a \rightarrow a \rightarrow \text{Bool}\), even though equality is not defined for every type
**class Eq**

1. **type classes** are used for overloading: a class is a "container" of overloaded operations

2. we can declare a type to be an **instance** of a type class, meaning that it implements its operations

3. e.g. class Eq

   ```haskell
   class Eq a where
   (==) :: a -> a -> Bool
   ```

4. now the type of (==) is

   ```haskell
   (==) :: (Eq a) => a -> a -> Bool
   ```

5. Eq a is a **constraint** on type a, it means that a must be an instance of Eq
Defining instances

e.g. elem has type (Eq a) => a -> [a] -> Bool

we can define instances like this:

instance (Eq a) => Eq (Tree a) where
-- type a must support equality as well
  Leaf a == Leaf b = a == b
  (Branch l1 r1) == (Branch l2 r2) = (l1==l2) && (r1==r2)
_ == _ = False

an implementation of (==) is called a method

CAVEAT do not confuse all these concepts with the homonymous concepts in OO programming: there are similarities but also big differences
## Haskell vs Java concepts

<table>
<thead>
<tr>
<th>Haskell</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Interface</td>
</tr>
<tr>
<td>Type</td>
<td>Class</td>
</tr>
<tr>
<td>Value</td>
<td>Object</td>
</tr>
<tr>
<td>Method</td>
<td>Method</td>
</tr>
</tbody>
</table>

1. in Java, an Object is an *instance* of a Class
2. in Haskell, a Type is an *instance* of a Class
Eq and Ord in the Prelude

1. Eq offers also a standard definition of $\neq$, derived from $(==)$:
   
   ```haskell
   class Eq a where
     (==), (/=) :: a -> a -> Bool
     x /= y = not (x == y)
   ```

2. We can also extend Eq with comparison operations:
   
   ```haskell
   class (Eq a) => Ord a where
     (<), (<=), (>=), (>) :: a -> a -> Bool
     max, min :: a -> a -> a
   ```

3. Ord is also called a subclass of Eq.

4. It is possible to have multiple inheritance: class $(X a, Y a) \Rightarrow Z a$
Another important class: Show

1. It is used for **showing**: to have an instance we must implement `show`.

2. E.g., functions do not have a standard representation:

```
Prelude> (+)
<interactive>:2:1:
    No instance for (Show (a0 -> a0 -> a0))
    arising from a use of ‘print’
Possible fix:
    add an instance declaration for (Show (a0 -> a0 -> a0))
```

3. Well, we can just use a trivial one:

   ```
   instance Show (a -> b) where
   show f = "<< a function >>"
   ```
we can also represent binary trees:

```haskell
instance Show a => Show (Tree a) where
  show (Leaf a) = show a
  show (Branch x y) = "<" ++ show x ++ " | " ++ show y ++ ">
```

e.g.

Branch
  (Branch
    (Leaf 'a') (Branch (Leaf 'b') (Leaf 'c')))
  (Branch
    (Leaf 'd') (Leaf 'e'))

is represented as

```
<<'a' | <'b' | 'c'>> | <'d' | 'e'>
```
usually it is not necessary to explicitly define instances of some classes, e.g. Eq and Show

Haskell can be quite smart and do it automatically, by using `deriving`

for example we may define binary trees using an infix syntax and automatic Eq, Show like this:

```haskell
infixr 5 :^:
data Tr a = Lf a | Tr a :^: Tr a

deriving (Show, Eq)
```

e.g.

```haskell
*Main> let x = Lf 3 :^: Lf 5 :^: Lf 2
*Main> let y = (Lf 3 :^: Lf 5) :^: Lf 2
*Main> x == y
False
*Main> x
Lf 3 :^: (Lf 5 :^: Lf 2)
```
Rock-paper-scissors in Haskell

data RPS = Rock | Paper | Scissors deriving (Show, Eq)

instance Ord RPS where
  x <= y | x == y       = True
  Rock <= Paper      = True
  Paper <= Scissors  = True
  Scissors <= Rock   = True
  _        <= _      = False

note that we only needed to define \((\leq)\) to have the instance
An example with class \texttt{Num}

a simple re-implementation of rational numbers

data \texttt{Rat} = \texttt{Rat} \ !\text{Integer} \ !\text{Integer} \ \text{deriving Eq}

\begin{align*}
\text{simplify} \ (\text{Rat} \ x \ y) &= \ \text{let} \ g = \text{gcd} \ x \ y \\
&\quad \text{in} \ \text{Rat} \ (x \ \text{‘div’} \ g) \ (y \ \text{‘div’} \ g) \\
\text{makeRat} \ x \ y &= \text{simplify} \ (\text{Rat} \ x \ y)
\end{align*}

\begin{align*}
\text{instance Num Rat where} \\
(\text{Rat} \ x \ y) + (\text{Rat} \ x' \ y') &= \text{makeRat} \ (x*y' + x'*y) \ (y*y') \\
(\text{Rat} \ x \ y) - (\text{Rat} \ x' \ y') &= \text{makeRat} \ (x*y' - x'*y) \ (y*y') \\
(\text{Rat} \ x \ y) \times (\text{Rat} \ x' \ y') &= \text{makeRat} \ (x*x') \ (y*y') \\
\text{abs} \ (\text{Rat} \ x \ y) &= \text{makeRat} \ (\text{abs} \ x) \ (\text{abs} \ y) \\
\text{signum} \ (\text{Rat} \ x \ y) &= \text{makeRat} \ (\text{signum} \ x \ast \text{signum} \ y) \ 1 \\
\text{fromInteger} \ x &= \text{makeRat} \ x \ 1
\end{align*}
An example with class Num (cont.)

1. **Ord:**
   
   ```haskell
   instance Ord Rat where
   (Rat x y) <= (Rat x' y') = x*y' <= x'*y
   ```

2. **a better show:**
   
   ```haskell
   instance Show Rat where
   show (Rat x y) = show x ++ "/" ++ show y
   ```

3. **note:** Rationals are in the Prelude!

4. **moreover,** there is class Fractional for `/` (not covered here)

5. **but we could define our version of division as follows:**
   
   ```haskell
   x // (Rat x' y') = x * (Rat y' x')
   ```
1. what is the type of the standard function `getChar`, that gets a character from the user? `getChar :: theUser -> Char`?

2. first of all, it is not **referentially transparent**: two different calls of `getChar` could return different characters.

3. In general, IO computation is based on **state change** (e.g. of a file), hence if we perform a **sequence of operations**, they must be performed in **order** (and this is not easy with **call-by-need**).
getChar can be seen as a function :: Time -> Char.

indeed, it is an **IO action** (in this case for Input): getChar :: IO Char

quite naturally, to print a character we use **putChar**, that has type:

putChar :: Char -> IO ()

**IO** is an instance of the **Monad** class, and in Haskell it is considered as an **indelible stain of impurity**
A very simple example of an IO program

1. **main** is the default entry point of the program (like in C)

   ```haskell
   main = do {
     putStr "Please, tell me something>";
     thing <- getLine;
     putStrLn $ "You told me " ++ thing ++ "\"."
   }
   ``

2. special syntax for working with IO: **do**, `<-

3. we will see its real semantics later, used to define an IO action as an ordered sequence of IO actions

4. "<-" (note: not =) is used to obtain a value from an IO action

5. types:

   ```haskell
   main       :: IO ()
   putStrLn   :: String -> IO ()
   getline    :: IO String
   ```
 compile with e.g. **ghc readFile.hs**

```haskell
import System.IO
import System.Environment

readfile = do
  args <- getArgs; -- command line arguments
  handle <- openFile (head args) ReadMode;
  contents <- hGetContents handle; -- note: lazy
  putStrLn contents;
  hClose handle;

main = readFile
```

2. **readFile stuff.txt** reads "stuff.txt" and shows it on the screen
3. **hGetContents** reads lazily the contents of the file
Of course, purely functional Haskell code can raise exceptions: `head []`, `3 ‘div’ 0`, …

but if we want to catch them, we need an IO action:

```haskell
handle :: Exception e => (e -> IO a) -> IO a -> IO a; the 1st argument is the handler
```

Example: we catch the errors of `readfile`

```haskell
import Control.Exception
import System.IO.Error
...
main = handle handler readfile
    where handler e |
           | isDoesNotExistError e = putStrLn "This file does not exist."
           | otherwise = putStrLn "Got a problem."
```
What about usual, practical data structures (e.g. arrays, hash-tables)?

Traditional versions are imperative! If really needed, there are libraries with imperative implementations living in the IO monad.

Idiomatic approach: use immutable arrays (Data.Array), and maps (Data.Map, implemented with balanced binary trees).

`find` are respectively $O(1)$ and $O(\log n)$; `update` $O(n)$ for arrays, $O(\log n)$ for maps.

Of course, the update operations `copy` the structure, do not change it.
Example code: Maps

import Data.Map

exmap = let m = fromList [("nose", 11), ("emerald", 27)]
    n = insert "rug" 98 m
    o = insert "nose" 9 n
    in (m ! "emerald", n ! "rug", o ! "nose")

exmap evaluates to (27,98,9)
Example code: Arrays

1. (//) is used for update/insert

2. `listArray`’s first argument is the **range** of indexing (in the following case, indexes are from 1 to 3)

   ```haskell
   import Data.Array

   exarr = let m = listArray (1,3) ["alpha","beta","gamma"]
           n = m // [(2,"Beta")]
           o = n // [(1,"Alpha"),(3,"Gamma")]
           in (m ! 1, n ! 2, o ! 1)
   ``

3. `exarr` evaluates to ("alpha","Beta","Alpha")
Class **Foldable**

1. **Foldable** is a class used for *folding*, of course
2. The main idea is the one we know from *foldl* and *foldr* for lists:
3. we have a container, and a binary operation $f$, and we want to apply $f$ to all the elements in the container
4. a minimal implementation of Foldable requires *foldr*
Let’s go back to our binary trees

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

we can easily define a foldr for them

tfoldr f z Empty = z
\[ tfoldr f z (\text{Leaf } x) = f \ x \ z \]
\[ tfoldr f z (\text{Node } l \ r) = tfoldr f (tfoldr f z r) l \]

instance Foldable Tree where
    foldr = tfoldr

> foldr (+) 0 (Node (Node (Leaf 1) (Leaf 3)) (Leaf 5))
9
Maybe

1. **Maybe** is used to represent computations that may fail: we either have \textit{Just }v\textit{, if we are lucky, or }Nothing.\textit{.}

2. It is basically a simple "conditional container"

   \texttt{data Maybe a = Nothing } \mid \texttt{Just a}

3. It is adopted in many recent languages, to avoid NULL and limit exceptions usage.

4. Examples are Scala (basically the ML family approach): Option[T], with values None or Some(v); Swift, with Optional<T>.

5. It is quite simple, so we will use it in our examples with Functors & C.
Of course, Maybe is foldable

instance Foldable Maybe where
  foldr _ z Nothing = z
  foldr f z (Just x) = f x z
1. The **Functor** class is the class of all the types that offer a *map* operation (so there is an analogy with Foldable vs folds).

2. The map operation of functors is called **fmap** and has type:

   \[ \text{fmap} : (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b \]

3. It is quite natural to define map for a container, e.g.:

   ```hs
   instance Functor Maybe where
   fmap _ Nothing = Nothing
   fmap f (Just a) = Just (f a)
   ```
Well-defined functors should obey the following laws:

1. \( \text{fmap id} = \text{id} \) (where \( \text{id} \) is the identity function)
2. \( \text{fmap (f \ . \ g)} = \text{fmap f \ . \ fmap g} \)
3. You can try, as an exercise, to check if the functors we are defining obey the laws
First, let us define a suitable map for trees:

\[
\begin{align*}
tmap f \text{ Empty} &= \text{ Empty} \\
tmap f (\text{Leaf } x) &= \text{ Leaf } f \ x \\
tmap f (\text{Node } l \ r) &= \text{ Node } (tmap f \ l) (tmap f \ r)
\end{align*}
\]

That's all we need:

```
instance Functor Tree where
  fmap = tmap
```

-- example
> fmap (+1) (Node (Node (Leaf 1) (Leaf 2)) (Leaf 3))
Node (Node (Leaf 2) (Leaf 3)) (Leaf 4)
In our voyage toward monads, we must consider also an extended version of functors, i.e. *Applicative functors*

The definition looks indeed exotic:

```
class (Functor f) => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

- note that \( f \) is a type constructor, and \( f \ a \) is a Functor type
- moreover, \( f \) must be parametric with one parameter
- if \( f \) is a container, the idea is not too complex:
  - pure takes a value and returns an \( f \) containing it
  - \(<*>\) is like `fmap`, but instead of taking a function, takes an \( f \) containing a function, to apply it to a suitable container of the same kind
Maybe is an Applicative Functor

Here is its definition:

```
instance Applicative Maybe where
    pure = Just
    Just f <> m = fmap f m
    Nothing <> _ = Nothing
```
Of course, lists are instances of Foldable and Functor. What about Applicative?

For that, it is first useful to introduce **concat**

```haskell
concat :: Foldable t => t [a] -> [a]
```

So we start from a container of lists, and get a list containing the **concatenation** of them:

```haskell
concat [[1,2],[3],[4,5]] is [1,2,3,4,5]
```

It can be defined as:

```haskell
concat l = foldr (++) [] l
```

its composition with **map** is called **concatMap**

```haskell
concatMap f l = concat $ map f l
```

> concatMap (\x -> [x, x+1]) [1,2,3] = [1,2,2,3,3,4]
With `concatMap`, we get the standard implementation of `<*>` for lists:

```haskell
instance Applicative [] where
  pure x = [x]
  fs <*> xs = concatMap (\f -> map f xs) fs
```

What can we do with it? For instance we can apply list of operations to lists:

```haskell
> [(+1),(*2)] <*> [1,2,3]
[2,3,4,2,4,6]
```

Note that we `map` the operations in sequence, then we `concatenate` the resulting lists.
Following the list approach, we can make our binary trees an instance of Applicative Functors.

First, we need to define what we mean by tree concatenation:

\[
\begin{align*}
t\text{conc } \text{Empty } t &= t \\
t\text{conc } t \text{ Empty } &= t \\
t\text{conc } t_1 t_2 &= \text{Node } t_1 t_2
\end{align*}
\]

now, concat and concatMap (here tconcmap for short) are like those of lists:

\[
\begin{align*}
t\text{concat } t &= \text{tfoldr } t\text{conc } \text{Empty } t \\
t\text{concmap } f t &= t\text{concat } \$ t\text{map } f t
\end{align*}
\]
Here is the natural definition (practically the same of lists):

```haskell
instance Applicative Tree where
    pure = Leaf
    fs <*> xs = tconcmap (\f -> tmap f xs) fs
```

Let’s try it:

```haskell
> (Node (Leaf (+1))(Leaf (*2))) <*>
    Node (Node (Leaf 1) (Leaf 2)) (Leaf 3)

Node (Node (Node (Leaf 2) (Leaf 3))
     (Leaf 4))
  (Node (Node (Leaf 2) (Leaf 4))
       (Leaf 6))
```
A peculiar type class: Monad

1 introduced by Eugenio Moggi in 1991, a monad is a kind of algebraic data type used to represent computations (instead of data in the domain model) - we will often call these computations actions

2 monads allow the programmer to chain actions together to build an ordered sequence, in which each action is decorated with additional processing rules provided by the monad and performed automatically

3 monads are flexible and abstract. This makes some of their applications a bit hard to understand.
monads also can be used to make imperative programming easier in a pure functional language.

In practice, through them it is possible to define an imperative sub-language on top of a purely functional one.

There are many examples of monads and tutorials (many of them quite bad) available in the Internet.
class Applicative m => Monad m where
  -- Sequentially compose two actions, passing any value produced
  -- by the first as an argument to the second.
  (>>=) :: m a -> (a -> m b) -> m b
  -- Sequentially compose two actions, discarding any value produced
  -- by the first, like sequencing operators (such as the semicolon)
  -- in imperative languages.
  (>>) :: m a -> m b -> m b
  m >> k = m >>= \_ -> k
  -- Inject a value into the monadic type.
  return :: a -> m a
  return = pure
  -- Fail with a message.
  fail :: String -> m a
  fail s = error s
1. Note that only `>>=` is required, all the other methods have standard definitions.

2. `>>=` and `>>` are called **bind**.

3. `m a` is a *computation* (or action) resulting in a value of type `a`.

4. **return** is by default **pure**, so it is used to create a single monadic action. E.g. `return 5` is an action containing the value 5.

5. While **bind** operators are used to compose actions:
   - `x >>= y` performs the computation `x`, takes the resulting value and passes it to `y`; then performs `y`.
   - `x >> y` is analogous, but "throws away" the value obtained by `x"
Its definition is straightforward

```haskell
instance Monad Maybe where
    (Just x) >>= k       = k x
    Nothing >>= _       = Nothing
    fail _               = Nothing
```
The information managed automatically by the monad is the “bit” which encodes the **success** (i.e. *Just*) or failure (i.e. *Nothing*) of the action sequence.

1. e.g. `Just 4 >>= Just >> Nothing >> Just 6` evaluates to *Nothing*.
2. a variant: `Just 4 >>= Just >> Nothing >> Just 6` (what is the result in this case?)
for a monad to behave correctly, method definitions must obey the following laws:

1) *return* is the **identity element**:

\[
(return \ x) >>= f \iff f \ x
\]
\[
m >>= return \iff m
\]

2) **associativity** for binds:

\[
(m >>= f) >>= g \iff m >>= (\lambda x \to (f \ x >>= g))
\]

3) (monads are analogous to **monoids**, with *return* \(= 1\) and \(>>= = \cdot\))
Example: monadic laws application with Maybe

1. \( \text{return } 4 :: \text{Maybe Integer} \gg= \ \lambda x -> \text{Just } (x+1) \)
   \>
   Just 5
>
   Just 5 \gg= \text{return}
   Just 5

2. \( \text{return } 4 \gg= \ \lambda x -> \text{Just } (x+1) \)
    \>
    \gg= \ \lambda x -> \text{Just } (x*2) \)
    
    Just 10
>
    return 4 \gg= (\ \lambda y ->
    ((\ \lambda x -> \text{Just } (x+1)) y)
    \gg= \ \lambda x -> \text{Just } (x*2))
    
    Just 10
1. The **do** syntax is used to avoid the explicit use of `>>=` and `>>`
2. The essential translation of **do** is captured by the following two rules:
   
   \[
   \begin{align*}
   & \text{do } e_1 ; e_2 \quad \iff \quad e_1 >> e_2 \\
   & \text{do } p <- e_1 ; e_2 \quad \iff \quad e_1 >>= \lambda p \rightarrow e_2 \\
   \end{align*}
   \]

3. note that they can also be written as:
   
   \[
   \begin{align*}
   & \text{do } e_1 \quad \text{do } e_2 \\
   & \text{do } p <- e_1 \quad \text{do } e_2 \\
   \end{align*}
   \]

4. or:
   
   \[
   \begin{align*}
   & \text{do } \{ \quad e_1 \; ; \\
   & \quad \text{do } \{ \quad p <- e_1 \; ; \\
   & \quad \text{do } \} \\
   & \quad \text{do } \} \\
   \end{align*}
   \]
Caveat: **return** does not return

1. IO is a build-in monad in Haskell: indeed, we used the *do* notation for performing IO
2. there are some catches, though – it looks like an imperative sub-language, but its semantics is based on binds and pure
3. For example:

   ```haskell
   esp :: IO Integer
   esp = do x <- return 4
            return (x+1)
   ```

   ```
   > esp
   5
   ```
The List Monad

1. **List**: monadic binding involves joining together a set of calculations for each value in the list.

2. In practice, `bind` is `concatMap`

   ```haskell
   instance Monad [] where
      xs >>= f = concatMap f xs
   fail _ = []
   ```
Lists: do vs comprehensions

1. list comprehensions can be expressed in do notation
2. e.g. this comprehension

\[ (x, y) \mid x \leftarrow [1,2,3], y \leftarrow [1,2,3] \]
3. is equivalent to:

\[
\begin{align*}
\text{do} & \quad x \leftarrow [1,2,3] \\
& \quad y \leftarrow [1,2,3] \\
& \quad \text{return} \ (x,y)
\end{align*}
\]
to understand our example of comprehension, i.e.

```haskell
testcomp = do x <- [1,2,3]
  y <- [1,2,3]
  return (x,y)
```

we can rewrite it following the monad definition:

```haskell
testcomp' =
  [1,2,3] >>= (\x -> [1,2,3] >>=
                (\y ->
                  return (x,y)))
```
that is:

testcomp'' =
  concatMap f0 [1,2,3]
  where f0 x = concatMap f1 [1,2,3]
    where f1 y = [(x,y)]
We can now try to define our own monad with out binary trees

Knowing about lists, is not too hard:

```haskell
instance Monad Tree where
    xs >>= f = tconcmap f xs
    fail _ = Empty
```
Now some examples

A simple monadic comprehension:

```haskell
exmon :: (Monad m, Num r) => m r -> m r -> m r
exmon m1 m2 = do x <- m1
               y <- m2
               return $ x - y
```
Let’s apply it to lists and trees

1 First, we try with lists:
   > exmon [10, 11] [1, 7]
   [9,3,10,4]

2 on trees is not much different
   > exmon (Node (Leaf 10) (Leaf 11)) (Node (Leaf 1) (Leaf 7))
   Node (Node (Leaf 9) (Leaf 3))
   (Node (Leaf 10) (Leaf 4))