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 \( f: \mathbb{N} \rightarrow \mathbb{N} \), \( f(5) \) 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 **absence of side effects** (purely functional computations) from the point of view of the result the **order** in which functions are applied **does not matter** (almost).

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

```scheme
(define (sum-square x y)
  (+ (* x x)
      (* y y)))
```
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?
The two evaluations differ in the **order** in which function applications are evaluated.

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) \textit{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 \((\text{sum-square} \ (+\ 1\ 2)\ (+\ 2\ 3))\) there are 3 redexes: \((\text{sum-square} \ (+\ 1\ 2)\ (+\ 2\ 3)))\), \((+\ 1\ 2)\) and \((+\ 2\ 3)\) the innermost that is also leftmost is \((+\ 1\ 2)\), which is applied, giving expression \((\text{sum-square} \ 3\ (+\ 2\ 3))\)

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

note that Scheme does not require that we take the \textit{leftmost}, but this is very common in mainstream languages
a dual evaluation strategy: redexes are evaluated in an **outermost** fashion

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

in the outermost strategy, functions are always **applied before their arguments**, this corresponds to passing arguments **by name** (like in Algol 60).
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
\textbf{Force (code)}

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

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

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:

```scheme
(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. \(\mathbb{C} \rightarrow \mathbb{C} \rightarrow \mathbb{C}\) (\(\rightarrow\) 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.
1 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
2 Standards: Haskell ’98 and ’10
3 Motto: "Avoid success at all costs"
   ex. usage: Google’s Ganeti cluster virtual server management tool
4 I mainly follow Hudak, Peterson, Fasel, A Gentle Introduction to Haskell 98, 1999
5 Beware! There are many bad tutorials on Haskell and monads, in particular, available online
A taste of Haskell’s syntax

1. more complex and "human" than Scheme: parentheses are optional!
2. function call is similar, though: \( f \ x \ y \) stands for \( f(x,y) \)
3. there are infix operators and are made of non-alphabetic characters (e.g. *, +, but also <++>)
4. elem is \( \in \). If you want to use it infix, just use ‘elem’
5. -- this is a comment
6. lambdas: \( \lambda (x \ y) (+ \ 1 \ x \ y)) \) is written \( \\backslash x \ y \rightarrow 1+x+y \)
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**
functions are declared through a sequence of *equations*

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

- this is also an example of *pattern matching*
- arguments are matched with the right parts of equations, top to bottom
- if the match succeeds, the function body is called
the previous definition of \texttt{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

\begin{quote}
\texttt{length :: [a] \to Integer}
\end{quote}

lower case letters are \textbf{type variables}, so \texttt{[a]} stands for \textit{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**
   1. it is (roughly) the *least general type that contains all the instances of the expression*
   2. 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
Lists are recursive types

1. Of course, also lists are recursive. Using Scheme jargon, they could be defined by:

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

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

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

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

```haskell
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.

```haskell
Prelude> :t (+1)
(+1) :: Num a => a -> a
Prelude> :t +
<interactive>:1:1: parse error on input ‘+’
Prelude> :t (+)

(+) :: Num a => a -> a -> a
```

2. **Prelude** is the standard library

3. we’ll see later the exact meaning of `Num a =>` with type classes. Its meaning here is that `a` must be a numerical type
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 (.)
(.) :: (b -> c) -> (a -> b) -> a -> c

2. $\$ syntax for avoiding parentheses, e.g. $(10\$) (5+3) = (10\$) 5+3$
call-by-need is very convenient for dealing with **never-ending computations** that provide data

here are some simple example functions:

```
ones = 1 : ones

numsFrom n = n : numsFrom (n+1)

squares = map (^2) (numsFrom 0)
```

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

e.g.

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

Convenient syntax for creating infinite lists:

e.g. ones before can be also written as \([1,1..]\)

`numsFrom 6` is the same as \([6..]\)

`zip` is a useful function having type `zip :: [a] -> [b] -> [(a, b)]`

\[
\text{zip } [1,2,3] \text{ "ciao"}
-- => \[(1,'c'),(2,'i'),(3,'a')\]
\]

List comprehensions

\[
\[(x,y) \mid x <- [1,2], y <- \text{"ciao"}] \\
-- => \[(1,'c'),(1,'i'),(1,'a'),(1,'o'), \(2,'c'),(2,'i'),(2,'a'),(2,'o')\]
\]
a list with all the Fibonacci numbers (note: tail is cdr, while head is car)

\[
\text{fib} = 1 : 1 : \\
[a+b \mid (a,b) <- \text{zip fib (tail fib)}]
\]
1. **bottom** (aka $\perp$) 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)
Pattern matching

1. The matching process proceeds top-down, left-to-right
2. Patterns may have **boolean guards**
   
   \[
   \begin{align*}
   \text{sign } x & \quad | \quad x > 0 \quad = \quad 1 \\
   & \quad | \quad x == 0 \quad = \quad 0 \\
   & \quad | \quad x < 0 \quad = \quad -1
   \end{align*}
   \]
3. \_ stands for *don’t care*
4. E.g. definition of **take**
   
   \[
   \begin{align*}
   \text{take } 0 \_ & \quad = \quad [] \\
   \text{take } \_ \quad [] & \quad = \quad [] \\
   \text{take } n \quad (x:x:xs) & \quad = \quad x : \text{take} \quad (n-1) \quad xs
   \end{align*}
   \]
the order of definitions **matters**:

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

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

what does it change, if we swap the first two defining equations?
**Case**

1. **take with case:**

```haskell
take m ys = case (m,ys) of
    (0,_) -> []
    (_,[]) -> []
    (n,x:xs) -> x : take (n-1) xs
```
let and where

1. **let** is like Scheme’s letrec*:

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

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

   ```vc
   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:

   ```vc
   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:

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

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. \(\text{foldl } (+) \ 0 \ [1,2,3]\)
   2. \(\text{foldl } (+) \ (0 + 1) \ [2,3]\)
   3. \(\text{foldl } (+) \ (((0 + 1) + 2) \ [3])\)
   4. \(\text{foldl } (+) \ ((((0 + 1) + 2) + 3) \ [])\)
   5. \(((0 + 1) + 2) + 3) = 6\)
Haskell is too lazy: an interlude on strictness

1. There are various ways to enforce **strictness** in Haskell (analogously there are classical approaches to introduce laziness in strict languages)

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

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

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

1. Canonical operator to **force evaluation** is `seq :: a -> t -> t`.
2. `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`)

   ```haskell
   foldl' f z [] = z
   foldl' f z (x:xs) = let z' = f z x
                       in seq z' (foldl' f z' xs)
   ```

4. Strict versions of standard functions are usually primed.
There is a convenient *strict* variant of $ (function application) called $!.

Here is its definition:

\[
($) ! : (a \to b) \to a \to b
f \; ($) ! \; x = \text{seq} \; x \; (f \; x)
\]
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

module Tree ( Tree (Leaf, Branch), fringe ) where
data Tree a = Leaf a | Branch (Tree a) (Tree a)
fringe :: Tree a -> [a] ...

module Main (main) where
import Tree ( Tree (Leaf, Branch) )
main = print (Branch (Leaf 'a') (Leaf 'b'))
modules provide the only way to build abstract data types (ADT)

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

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
```
ADT implementation

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...

```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

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

its type should be: a -> [a] -> Bool. But this means that (==) :: a -> a -> Bool, even though equality is not defined for every type
type classes are used for overloading: a class is a "container" of overloaded operations.

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

e.g. class Eq

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

now the type of (==) is

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

Eq a is a constraint on type a, it means that a must be an instance of Eq.
e.g. elem has type \((\text{Eq } a) \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}\)

we can define instances like this:

```haskell
instance (Eq a) \Rightarrow \text{Eq } (\text{Tree } a) \text{ where }
  -- type \(a\) must support equality as well
  \text{Leaf } a == \text{Leaf } b = a == b
  (\text{Branch } l1 r1) == (\text{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 offers also a standard definition of $\neq$, derived from $(==)$:

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

2. we can also extend Eq with comparison operations:

```
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:
   
   ```haskell
   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:
   
   ```haskell
   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'>>
```
Deriving

1. Usually it is not necessary to explicitly define instances of some classes, e.g. `Eq` and `Show`.
2. Haskell can be quite smart and do it automatically, by using `deriving`.
3. 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)
   ```

4. 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)
   ```
An example with class Ord

1. Rock-paper-scissors in Haskell

```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
```

2. note that we only needed to define (<=) to have the instance
a simple re-implementation of rational numbers

data Rat = Rat !Integer !Integer deriving Eq

simplify (Rat x y) = let g = gcd x y
             in Rat (x ‘div’ g) (y ‘div’ g)
makeRat x y = simplify (Rat x y)

instance Num Rat where
  (Rat x y) + (Rat x’ y’) = makeRat (x*y’+x’*y) (y*y’)
  (Rat x y) - (Rat x’ y’) = makeRat (x*y’-x’*y) (y*y’)
  (Rat x y) * (Rat x’ y’) = makeRat (x*x’) (y*y’)
  abs (Rat x y) = makeRat (abs x) (abs y)
  signum (Rat x y) = makeRat (signum x * signum y) 1
  fromInteger x = makeRat x 1
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’)
   ```
what is the type of the standard function \texttt{getChar}, that gets a character from the user? \texttt{getChar :: theUser \rightarrow Char}?

first of all, it is not \textit{referentially transparent}: two different calls of \texttt{getChar} could return different characters

In general, IO computation is based on \textit{state change} (e.g. of a file), hence if we perform a \textit{sequence of operations}, they must be performed in \textit{order} (and this is not easy with \textit{call-by-need})
1. getChar can be seen as a function :: Time -> Char.
2. indeed, it is an **IO action** (in this case for Input): getChar :: IO Char
3. quite naturally, to print a character we use **putChar**, that has type: putChar :: Char -> IO ()
4. IO is an instance of the **Monad** class, and in Haskell it is considered as an **indelible stain of impurity**
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 ++ "\".";
}
```

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

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

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

types:

- main :: IO ()
- putStrLn :: String -> IO ()
- getline :: IO String
编译时可以使用 `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` 读取 "stuff.txt" 并显示在屏幕上

3. `hGetContents` 以懒惰方式读取文件内容
Of course, purely functional Haskell code can raise exceptions: \textit{head} [], 3 \textit{\texttt{div}} 0, \ldots

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

\texttt{handle :: Exception e => (e -> IO a) \to IO a \to IO a;} the 1st argument is the \textit{handler}

Example: we catch the errors of \texttt{readfile}

\begin{verbatim}
import Control.Exception
import System.IO.Error
...
main = handle handler readfile
  where handler e
        | isDoesNotExistError e = putStrLn "This file does not exist."
        | otherwise = putStrLn "Something is wrong."
\end{verbatim}
Other classical data structures

1. What about usual, practical data structures (e.g. arrays, hash-tables)?
2. Traditional versions are imperative! If really needed, there are libraries with imperative implementations living in the IO monad
3. Idiomatic approach: use immutable arrays (Data.Array), and maps (Data.Map, implemented with balanced binary trees)
4. **find** are respectively $O(1)$ and $O(\log n)$; **update** $O(n)$ for arrays, $O(\log n)$ for maps
5. of course, the update operations *copy* the structure, do not change it
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)
   
   exarr evaluates to ("alpha","Beta","Alpha")
   ```
We saw that IO is a type constructor, instance of Monad.

But we still do not know what a Monad is.

Recent versions of GHC make the trip a bit longer, because we need first to introduce the following classes:

1. Foldable (not required, but useful)
2. Functor
3. Applicative (Functor)
Foldable is a class used for folding, of course

The main idea is the one we know from foldl and foldr for lists:

we have a container, and a binary operation $f$, and we want to apply $f$ to all the elements in the container

a minimal implementation of Foldable requires foldr
Example: foldable binary trees

1. Let’s go back to our binary trees
   
   ```haskell
   data Tree a = Empty | Leaf a | Node (Tree a) (Tree a)
   ```

2. we can easily define a `foldr` for them

   ```haskell
   tfoldr f z Empty = z
   tfoldr f z (Leaf x) = f x z
   tfoldr f z (Node l r) = tfoldr f (tfoldr f z r) l
   ```

   instance Foldable Tree where
   
   ```haskell
   foldr = tfoldr
   ```

   ```haskell
   > 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 *Just* $v$, if we are lucky, or *Nothing*.

2. It is basically a simple "conditional container"

   data Maybe a = Nothing | 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 _ _ Nothing = z
  foldr f _ (Just x) = f x z
**Functor** is the class of all the types that offer a `map` operation

(so there is an analogy with Foldable vs folds)

the map operation of functors is called **`fmap`** and has type:

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

it is quite natural to define `map` for a container, e.g.:

```haskell
instance Functor Maybe where
    fmap _ Nothing = Nothing
    fmap f (Just a) = Just (f a)
```
Functor laws

Well-defined functors should obey the following laws:

1. $fmap \, id = id$ (where $id$ is the identity function)
2. $fmap \, (f \circ g) = fmap \, f \circ fmap \, g$
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:

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

That’s all we need:

```haskell
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:

```haskell
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:

1. `pure` takes a value and returns an `f` containing it
2. `<*>` 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:

```haskell
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 \texttt{concat}

\texttt{concat :: Foldable t => t [a] -> [a]}

So we start from a container of lists, and get a list with the \textit{concatenation} of them:

\texttt{concat \texttt{[[1,2],[3],[4,5]]} is \texttt{[1,2,3,4,5]}}

it can be defined as: \texttt{concat \texttt{l} = foldr (++) \texttt{[]} \texttt{l}}

its composition with \textit{map} is called \texttt{concatMap}

\texttt{concatMap \texttt{f \texttt{l} = concat \$ map \texttt{f \texttt{l}}}\n> \texttt{concatMap (\texttt{x \rightarrow [x, x+1]})) \texttt{[1,2,3]}}
> \texttt{[1,2,2,3,3,4]}}
Lists are instances of Applicative

1 With concatMap, we get the standard implementation of \(<*\) for lists:

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

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

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

3 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:

- \( \text{tconc} \) Empty \( t = t \)
- \( \text{tconc} \) \( t \) Empty = \( t \)
- \( \text{tconc} \) \( t_1 \) \( t_2 = \text{Node} \) \( t_1 \) \( t_2 \)

Now, \( \text{concat} \) and \( \text{concatMap} \) (here \( \text{tconcmap} \) for short) are like those of lists:

- \( \text{tconcat} \) \( t = \text{tfoldr} \) \( \text{tconc} \) Empty \( t \)
- \( \text{tconcmap} \) \( f \) \( t = \text{tconcat} \$ \text{tmap} \) \( f \) \( t \)
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))
```
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*.

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.

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

in practice, through them it is possible to define an \textbf{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
Note that only >>= is required, all the other methods have standard definitions.

> >>= and >> are called **bind**

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

* **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.

**bind** operators are used to compose actions

1. x >>= y performs the computation x, takes the resulting value and passes it to y; then performs y.
2. x >> y is analogous, but "throws away" the value obtained by x
Maybe is a Monad

Its definition is straightforward

```haskell
instance Monad Maybe where
    (Just x) >>= k     = k x
    Nothing >>= _     = Nothing
    fail _             = Nothing
```
Examples with **Maybe**

1. 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

2. e.g. `Just 4 >> Just 5 >> Nothing >> Just 6` evaluates to `Nothing`

3. a variant: `Just 4 >>= Just >> Nothing >> Just 6`

4. another: `Just 4 >> Just 1 >>= Just` (what is the result in this case?)
The monadic laws

1. for a monad to behave correctly, method definitions must obey the following laws:

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

   \[(\text{return } x) >>= f \Leftrightarrow f x\]
   \[m >>= \text{return} \Leftrightarrow m\]

3. 2) **associativity** for binds:

   \[(m >>= f) >>= g \Leftrightarrow m >>= (\lambda x -\rightarrow (f x >>= g))\]

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

1. \( (\text{return } 4 :: \text{Maybe Integer}) \gg= \ \lambda x \rightarrow \text{Just } (x+1) \)
   
   \begin{align*}
   & \text{Just } 5 \\
   \text{> } & \text{Just } 5 \gg= \text{return} \\
   & \text{Just } 5
   \end{align*}

2. \( (\text{return } 4 \gg= \ \lambda x \rightarrow \text{Just } (x+1)) \)
   \begin{align*}
   & \gg= \ \lambda x \rightarrow \text{Just } (x \times 2) \\
   & \text{Just } 10 \\
   \text{> } & \text{return } 4 \gg= (\lambda y \rightarrow \text{((\lambda x \rightarrow \text{Just } (x+1)) y)\gg= \ \lambda x \rightarrow \text{Just } (x \times 2)}) \\
   & \text{Just } 10
The **do** syntax is used to avoid the explicit use of `>>=` and `>>`

The essential translation of **do** is captured by the following two rules:

\[
\begin{align*}
\text{do } e_1 & \ ; \ e_2 & \iff & \ e_1 \ >> \ e_2 \\
\text{do } p \ <- \ e_1 & \ ; \ e_2 & \iff & \ e_1 \ >>= \ \lambda p \ -> \ e_2
\end{align*}
\]

Note that they can also be written as:

\[
\begin{align*}
\text{do } e_1 \quad \quad \quad \quad \text{do } p \ <- \ e_1 \\
\quad e_2 \quad \quad \quad \quad \quad e_2
\end{align*}
\]

Or:

\[
\begin{align*}
\text{do } \{ \quad e_1 \ ; \\
\quad e_2 \quad \} \\
\text{do } \{ \quad p \ <- \ e_1 \ ; \\
\quad e_2 \quad \} 
\end{align*}
\]
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 bind and pure
3. For example:

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

   > esp
   5
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:
   ```
   do x <- [1,2,3]
     y <- [1,2,3]
     return (x,y)
   ```
1. to understand our example of comprehension, i.e.
   
   \[
   \text{testcomp} = \text{do } x \leftarrow [1,2,3] \\
   y \leftarrow [1,2,3] \\
   \text{return } (x,y)
   \]

2. we can rewrite it following the monad definition:

   \[
   \text{testcomp'} = \\
   [1,2,3] >>= (\ x \rightarrow [1,2,3] >>= \\
   \ (\ y \rightarrow \\
   \ \ \ \ \ \text{return } (x,y)))
   \]
that is:

\[
\text{testcomp'' =}
\]

\[
\text{concatMap f0 [1,2,3]}
\]

where \( f0 \ x = \text{concatMap f1 [1,2,3]} \)

where \( f1 \ y = [(x,y)] \)
We can now to define our own monad with binary trees

Knowing about lists, it is not too hard:

```haskell
instance Monad Tree where
  xs >>= f = tconcmap f xs
  fail _ = Empty
```
Monads are abstract, so monadic code is very flexible, because it can work with any instance of Monad

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))
   ```
Monads can be used to implement parsers, continuations, ... and, of course, IO.

Let’s try exmon with IO Int:

```haskell
-- read is like in Scheme, here is used to parse the number
exmon (do putStrLn "?> "
         x <- getLine;
         return (read x :: Int))
(return 10)
```

What is the result, if we enter 12?
©2012-16 by Matteo Pradella
Licensed under Creative Commons License, Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0)