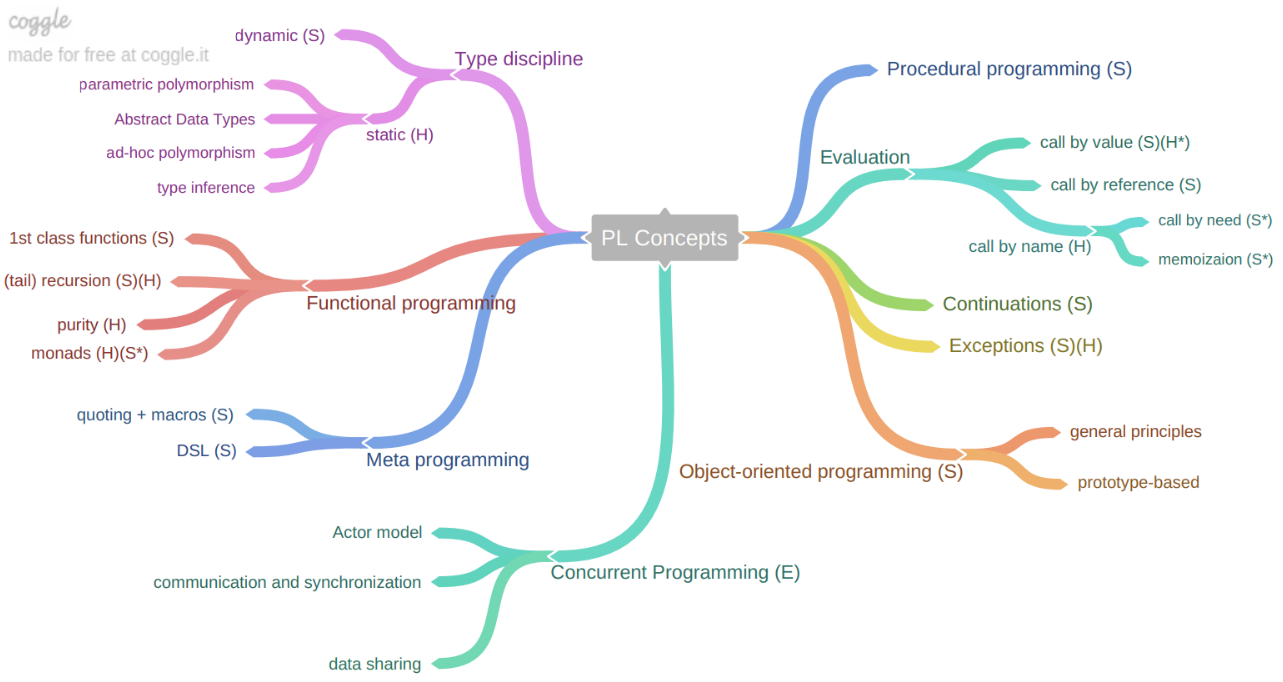
Principles of Programming Languages

github.com/martinopiaggi/polimi-notes

2022-2023

Contents

1 Languages panorama



Languages we will see:

- Scheme , for basics, memory management, introduction to functional programming and object orientation, meta-stuff
- Haskell , for static type systems and algebraic data types, functional "purity"
- Erlang, for concurrency-oriented programming

Not very useful to write real stuff, more useful to learn new concepts and be flexible. Probably we will never used this languages in our careers. A brief history of programming languages:

- 1957 Fortran (Formula Translator)
- 1958 LISP (LISt Processor)
- 1959 COBOL (Common Business Oriented Language)
- 1960 ALGOL 60 (Algorithmic Language)
- 1964 BASIC (Beginner's All-purpose Symbolic Instruction Code)
- 1967 Simula (first object-oriented lang.)
- 1970 Pascal, Forth
- 1972 **C**, Prolog (Datalog is a non-Turing complete subset of Prolog), Smalltalk
- 1975 **Scheme** (Lisp + Algol)
- 1978 ML (Meta-Language)
- 1980 Ada
- 1983 **C++**, Objective-C
- 1984 Common Lisp (Lisp + OO)
- 1986 **Erlang**
- 1987 Perl
- 1990 **Haskell**
- 1991 Python
- 1995 Java, **JavaScript**, Ruby, PHP
- 2001 **C#** (basically a variant of Java for dot environment)

- 2002 F#
- 2003 Scala (strange mix, quite interesting)
- 2007 Clojure
- 2009 Go
- 2011 Dart
- 2012 **Rust**
- 2014 Swift

2 Scheme

2.1 Main features

"Scheme . . . a lot of parentheses."

- Scheme dialect (which is in turn a language in the Lisp family) can be used in a **functional** style (but not purely functional)
- **Dynamically typed**: the type of a value is determined at runtime, rather than at compile-time. In dynamic typing, you don't have to specify the type of a variable when you declare it, and the type of the value stored in the variable can change dynamically during the execution of the program. (for example C++ is statically typed).
- **Functions are first class objects**; in this example I can pass any binary function to the function like any other object:

```
(define (fold binop n e)
  (if (<= n e)
      e
      (binop (fold binop (- n 1) e) n)))
```

- Scheme is **lexical scoped** like most programming languages (C, C++, and Java): the scope of a variable is determined by its location in the source code, more properly a variable always refers to its top-level environment. For example, if a variable is defined within a function, it will only be visible within that function and will not be accessible outside of it. This makes it easier to understand and manage the scope of variables in a program, because you can determine their visibility simply by looking at the source code. On the other hand, in dynamic scoping, the scope of a variable is determined by the order of function calls rather than its position in the source code.
- **Homoiconicity**: it means that there is no distinction between code and data and that for example statements, like expressions, are nestable.: `(+ 42 (if (< 1 0) 23 0))`. The `if` statement is nested inside the `+` expression and is treated as just another expression with a value.

2.2 Variables

- Booleans: `#t, #f`
- Numbers: `42, 1.23e+33, 23/169, 12+4i`
- Characters: `#a, #Z`
- Symbols: `a-symbol, another-symbol?, indeed!`
- Vectors: `#(1 2 3 4)`
- Strings: `"this is a string"`
- Pair: `(x . y)` where the `car` is `x` and the `cdr` is `y` .
- Lists: `'(1 2 3)` or `(quote (1 2 3))` or even `'(+ 1 2)'`

2.3 Lambda Expressions

You can use the `lambda` special form to create anonymous functions. Any lambda expression can be evaluated in any moment using `()`.

```
(lambda (<parameters>) <body> )

; here an example where I use "immediately" the lambda

> (display ((lambda (x y) (- x y))  2 3))
-1
```

2.4 Pairs and lists

A **pair** is an immutable data structure that holds two values.

- `cdr` of a list always returns a list while in a pair it returns a single value.
- `car` first element of a list
- `cdr` rest of a list
- `cons` function can be used to create a new cons cell by specifying its car and cdr values as arguments.

```
; Creating cons cells using the cons function
(cons 1 2) ; => '(1 . 2)
(cons 1 '(1 2)) ; => '(1 1 2)
(cons '(1) '(2)) ; => '((1) . (2))
(cons '(1) 'a ) ; => '((1) . a)
```

The names `cons`, `car`, and `cdr` are historical conventions from the original Lisp interpreter of 1958.

Lists are unbounded, possibly heterogeneous collections (Racket lists can contain items of different types) of data.

- `list` function can be used to create a new list by specifying its elements as arguments. Each argument will become an element in the resulting list (even if it is already a list itself).
- `length` length of a list
- `append` works as expected between two lists

```
; Creating lists using the list function
(list 1 2 3) ; => '(1 2 3)
(list '(1) '(2)) ; => '((1) (2))
(list 1) ;=> '(1)

> (append '(1 2) '(2 1))
'(1 2 2 1)

> (append '(1 2) '(1))
'(1 2 1)

> (append '(1 2) 1)
'(1 2 . 1)
```

2.5 Vectors

Vectors are more convenient and efficient than lists for some applications. Whereas accessing an arbitrary element in a list requires a linear traversal of the list up to the selected element, arbitrary vector elements are accessed in constant time. The *length* of a vector is the number of elements it contains. Vectors are indexed by exact nonnegative integers, and the index of the first element of any vector is 0. The highest valid index for a given vector is one less than its length.

As with lists, the elements of a vector can be of any type, and a single vector can hold more than one type of object.

A vector is written as a sequence of objects separated by whitespace, preceded by the prefix #(and followed by). For example, a vector consisting of the elements a, b, and c would be written #(a b c).

It's possible to use `(vector-length vect_name)` `(vector-ref vect i)`

> (make-vector 5 1) '#(1 1 1 1 1) (make-vector 5'(1,2,3)) '#((1 ,2 ,3) (1 ,2 ,3) (1 ,2 ,3) (1 ,2 ,3) (1 ,2 ,3))

2.6 Predicates

Predicates are procedures that returns boolean.

- `null?` often used in loops to check if we are at the end of a list: `(if (null? rest ) ;is rest = ()?`
- `eq?` tests if two objects (all objects except numbers) are the same
- `eqv?` like `eq?` but also checks numbers
- `pair?` is the predicate function.
- `equal?` is `#t` iff its arguments are equal as ordered trees:

```
- > (equal? '(1 2 3) '(1 2 3))
#t
> (equal? '(1 2 3) '(1 2 4))
#f
```

2.7 Quote and Eval

Note that in the last example `(quote )` is equivalent to `'` . In Scheme data and code are basically the same thing. The "inverse" of `(quote )` is `(eval )`. The `eval` procedure is used to evaluate a Scheme expression at runtime. It takes an expression as its argument and returns the result of evaluating that expression. For example: `scheme    (define x 5)    (eval '(+ x 3)) ; returns 8`

2.8 Apply

Both `eval` and `apply` are involved in evaluating expressions in Scheme:

- `eval` is used when you want to evaluate an individual expression dynamically at runtime.
- `apply` is used when you want to apply a function or procedure to multiple arguments provided as elements in a list.

`apply` takes *at least two* arguments, the first of them being a procedure and the last a list. It will call the procedure with the following arguments, including those inside the list.

```
(apply + 1 -2 3 '(20 20))
; => 42
```

This is the same as `(+ 1 -2 3 20 20)` `apply` has that name because it allows you to "apply" a procedure to several arguments.

```
(define arguments '(10 50 100))
(apply + arguments)
```

2.9 Let

Let is not assignment, is creating a new variable. Let is used to bind variables. We say the variables are *bound* to the values by the let.

```
(let ( (<var> <expr>) (<var> <expr>) ... )
  <let body>
)

//examples

(let ((x 3) (y 2)) ;It's parallel
  (+ x y)) ; 5
)

(let* ((x 3) (y 2))  ; let* it's sequential
  (+ x y)) ; 5
)
```

Variables created by let are local. To create top-level bindings there is define:

```
(define x 12)
(define y #(1 2 3))
```

2.10 Begin

We can perform a block of **procedural code** using `(begin ...)`:

```
(begin
    (op_1 ...)
    (op_2 ...)
    ...
    (op_n ...)
)
```

2.11 Defining Functions

If you go to the trouble of defining a function, you often want to save it for later use. You accomplish this by binding the result of a `lambda` to a variable using `define`, just as you would with any other value.

```
(define (double x)
        (* 2 x))

(define (centigrade-to-fahrenheit c)
        (+ (* 1.8 c) 32.0))
```

2.11.1 Bang Procedures

In general, procedures with side effects have a trailing bang (!) character. For example `set!` is used for assignment.

```
(begin
    (define x 23)
    (set! x 42)
    x)
```

is equals to 42.

2.11.2 Variable number of arguments

- Variable argument functions can accept zero or more additional arguments
- The additional arguments are captured as a list, which can be accessed and manipulated within the body of the procedure.
- The rest parameter acts as a placeholder for all extra arguments provided during function invocation.
- You can use various list manipulation functions like `car`, `cdr`, and others to work with the rest parameter and process its elements.

```
(define (my arg1 arg2 . args)
  (begin
    (display arg1) (newline)
    (display arg2) (newline)
    (display (car args)) (newline)
    (display (cdr args))
  )
)
```

```
(my 1 2 3 45 54 42)
```

Output:

```
1
2
3
(45 54 42)
```

2.11.2.1 Example To define a function `f` with a variable number of arguments, such that when called like `(f x1 x2 .. xn)`, it returns: `(xn (xn-1 ( .. (x1 (xn xn-1 .. x1))..))`, the function `f` can be defined using only fold operations for loops.

```
(define (f . L)
  (foldl
    (lambda (x y)
      (list x y))
    (foldl cons '() L)
    L))
```

2.12 Macros

A macro is simply a text substitution of the body of the macro over the macro name. Scheme has a very powerful Turing-complete macro system (note that Scheme is cool because you can define recursive macros).

You don't evaluate macros, you expand them.

Macros are of course expanded at compile-time and they are defined in this way:

```
(define-syntax <rule_name>
  (syntax-rules () ;variant of the rule name
  ((_ <first_parameter_here> <second_param> ...)
  <body_pattern_match>
  )))
```

In functions parameters are passed by values while in macros the parameters are passed by name.

```
(define-syntax while
(syntax-rules () ; no other needed keywords
((_ condition body ...)     ; pattern 1
(let loop () ; expansion of P
    (when condition
        (begin
        body ...
        (loop)))
))))
```

Note that _ is just a shorthand notation and ... is a way to say that you can have multiple elements as body. An other example:

```
(define-syntax block-then
  (syntax-rules (where then <-)
    ((_ ((e1 ...) ...)
       then
       ((e2 ...) ...)
       where (v <- a b) ...)
     (begin
       (let ((v a) ...)
         e1 ...))
       (let ((v b) ...)
         e2 ...)))))
```

Here's an example usage of the `block-then` construct:

```
(block-then
  ((displayln (+ x y))
   (displayln (* x y))
   (displayln (* z z)))
  then
  ((displayln (+ x y))
   (displayln (* z x)))
  where
    (x <- 12 3)
    (y <- 8 7)
    (z <- 3 2))
```

This will output:

```
20
96
9
10
6
```

2.12.1 Hygiene of macros

A macro could be expanded into a piece of code that contains a label that has the same name of a variable used by the user somewhere else. This would lead to a name clash. But in Scheme this **is not possible** since Scheme macros are **hygienic**: this means that symbols used in their definitions are actually replaced with special symbols not used anywhere else in the program.

2.13 Loops

Generally loops can be achieve using `let`:

```
(let (( x 0))
    (let label ()
    (when (< x 10)
        (display x)
        (newline )
        (set! x (+ 1 x ))
(label)))) ; go-to label
```

`(label)` is used to jump back at the start of the iteration body. But also remember that a loops can always be expressed as recursive functions:

```
(let label (( x 0))
    (when ( < x 10)
        (display x)
        (newline)
(label (+ x 1)))) ; x++
```

2.14 Recursion

Recursion is a favored technique in functional programming. The most important concept of recursion is **tail recursion**. A tail recursive function is one that returns the result of the recursive call back without alteration.

```
(define (fold op n e)
  (fold-helper op n e n))

(define (fold-helper op n e acc)
  (if (<= n e)
      acc
      (fold-helper op (- n 1) e (op acc (- n 1)))))

>(display (fold + 10 1))
> 55
>(+ 10 9 8 7 6 5 4 3 2 1)
```

The difference between a tail recursive function and a not tail recursive one is that at each time the operation is done **before** the recursion call: in this way "there are just calls of functions on the stack and **not pending operations**".

This is **NOT** tail recursive, since after the recursion it performs an multiplication:

```
(define (std-factorial n)
  (if (zero? n)
      1
      (* n (std-factorial (- n 1)))))
```

To turn this into a tail recursion function it's necessary an **accumulator**. This is a **typical pattern**: use an **helper function** with an additional parameter which accumulates the answer (the accumulator) to convert a non-tail recursive function into a tail recursive one.

```
(define (factorial n)
  (acc-factorial n 1))

;; auxiliary function that takes an additional parameter (the accumulator,
;; i.e. the result computed so far)

(define (acc-factorial n sofar)
  (if (zero? n)
      sofar
      (acc-factorial (- n 1) (* sofar n))))
```

2.15 Mapping and Folding

Some classical higher order functions are: **map**, **foldl**, **foldr**, **filter**:

```
((map (lambda (x) (+ 1 x)) '(0 1 2)) ; =>
(1 2 3)

(foldl cons '() '(1 2 3)) ; => (3 2 1)

(foldr cons '() '(1 2 3)) ; => (1 2 3)

(filter (lambda (x) (> x 0)) '(10 -11 0)) ;
=> (10)
```

Implementation of folds:

```
(define (fold-left f i L) ;foldl is tail recursive
  (if (null? L)
      i
      (fold-left f (f (car L) i) (cdr L))))

(define (fold-right f i L) ;foldr is not tail recursive
  (if (null? L)
      i
      (f (car L) (fold-right f i (cdr L)) )))
```

2.16 Continuations

A *continuation* is a special kind of function that's like a snapshot of the current state of a running program. Continuations let you jump back to an **earlier** point in the computation, **circumventing the control flow** of the program.

- When you invoke a continuation, it sends you back to the point in the program where the bookmarked expression was evaluated.
- The continuation takes one argument, which replaces the value of the bookmarked expression. Evaluation resumes from there.
- The fun part is when you treat the continuation as an object like others and for example, **save it in a variable**: the variable contains "a fixed point" in the control flow of the program.
- Continuations are also used as "glorified GOTO" since they permit to escape from cycles or bodies of procedures.
- `call/cc` is the keyword for continuations and it always takes a `lambda` function as argument.

```
(+ 23 (call/cc (lambda (k) (+ 10 9)))) ;Risultato: 42

(+ 11 (call/cc (lambda (k) (+ 23 (k 31))))) ;Risultato: 42 poichè si esce subito da call/cc appena a k è
```

This is a function which print until it finds a negative number, in this case the continuation is used as **GOTO**:

```
(define (print_until_neg l)
  (call/cc (lambda (exit) (for-each (lambda (x)
                 (if (< x 0)
                     (exit)
                     (begin (display x) (newline))))l))))

>  (prova '(1 2 -3 4))
1
2
```

Example of an alternative implementation of `call/cc`, called `store-cc`: the continuation is called only once and it is implicit. To run the continuation, we can use the associated construct `run-cc` (which may take parameters).

```
(define *stored-cc* '())
(define-syntax store-cc
  (syntax-rules ()
    ((_ e ...)
     (call/cc (lambda (k)
                 (set! *stored-cc* (cons k *stored-cc*))
                 e ...)))))
(define (run-cc . v)
  (let ((k (car *stored-cc*)))
    (set! *stored-cc* (cdr *stored-cc*))
    `(apply k ,v))))
```

2.17 Closures

The interesting thing to do with call/cc and continuations is using it to have **closures**. A closure is basically (explained in "spaghettata mode") a function with an environment, and we can write it using the ability of continuations to capture the state of its environment. Closures are a fundamental concept in functional programming and are used for a variety of purposes, including creating function factories (functions that return functions), iterators and generators. Let's see a "generator" of Fibonacci sequence:

```
(define fibo-stack #f)

(define (fibonacci-gen)
  (let ((last-fib 0) (fib 1) (x #f))
    (call/cc (lambda (fibo)
               (set! fibo-stack fibo))) ;assignment
    (set! x  (+ last-fib fib))
    (set! last-fib fib)
    (set! fib x)
    x))
```

`call/cc` allows us to "save the stack" by capturing the current continuation and storing it in `fibo-stack`. This captured continuation can then be invoked later, allowing us to jump back to the saved state.

```
> (fibonacci-gen)
1
> (fibo-stack)
2
> (fibo-stack)
3
> (fibo-stack)
```

```
5
> (fibo-stack)
8
> (fibo-stack)
13
> (fibo-stack)
21
```

At the line of `'assignment` comment I'm "saving" the computation exactly before the assignment of `last-fib` and `fib` . Every time that I call my continuation variable `fibo-stack` I obtain the next Fibonacci number (note that when I call `fibo-stack` I jump back inside the function and I perform the `set!` operations and, at the end, I'm giving back the variable `x` which contains the Fibonacci number).

In Scheme, the "closure as classes" approach refers to a programming technique where closures are used to simulate object-oriented programming and create class-like structures.

In traditional object-oriented programming languages, classes define objects with properties (attributes) and behaviors (methods). Objects of the same class share the same structure and behavior defined by their class. In Scheme, which is a functional programming language, there are no built-in constructs for defining classes or objects.

However, using closures in Scheme allows you to emulate some aspects of object-oriented programming. Closures are functions that capture variables from their surrounding environment. By combining closures with data structures like lists or records, you can create objects with associated state and behavior.

```
(define (test-closures)
  (define (greet)
    (display "Hello"))

  (define (farewell)
    (display "Goodbye"))

  (lambda (message)
    ((case message
       ((greet) greet)
       ((farewell) farewell)))))

(load "ex.rkt")
((test-closures) 'greet )
((test-closures) 'farewell )

(define t (test-closures))
(t 'greet)
(t 'farewell)
```

Explanation:

- The `test-closures` function defines two inner functions, `greet` and `farewell`, which are used to display different messages.
- It returns a lambda function that takes a message as an argument.
- Inside the lambda function, there is a case statement that matches the given message with either `'greet` or `'farewell`.
- If it matches `'greet`, it calls the `greet` function. If it matches `'farewell`, it calls the `farewell` function.
- The code then loads this file using `(load "ex.rkt")`.

- It demonstrates how to call these functions by directly invoking them with `(test-closures)` followed by either `'greet` or `'farewell`.
- Alternatively, you can assign `(test-closures)` to a variable `t`, and then call `t` with either `'greet` or `'farewell`.

2.18 Structs

New types can be defined with `struct` similar to C, but with some differences. Related procedures such as the constructor and a predicate to check the type of the new object are automatically created:

```
(struct node
  (left right))

> (define a (node 1 2))
```

- typeof: `(<struct>? <instance>)` to understand if a particular instance `<instance>` is of type `<struct_name>`
- getter: `(<struct>-<field> <instance>)`

Structs can inherit:

```
(struct node node-base ;inheritance
  (left right))
```

- setter: `(set-<struct>-<field>! <instance> <value>)` to set a value for a **mutable** field.

```
(struct leaf
  ((value #:mutable)) ) ;making a mutable field

> (define a (leaf 23))

> (display a)
#<leaf>

> (leaf-value a)
23

> (set-leaf-value! a 42) ;SETTER
> (leaf-value a)
42
```

2.19 Setup to code

1) Install Linux package Racket
2) Omit the #lang declaration in the source file
3) Launch from CLI `Racket`
4) Use `(load <file>)`

```
> racket
Welcome to Racket 7.6.
> (load "hello.rkt")
'hello-world
> (hello)
'hello-world
```

3 Haskell

3.1 Introduction

Haskell is a **functional** programming language that emphasizes **purity** and **lazy** evaluation. In order to understand how Haskell works, it's important to first understand the concepts of **function evaluation** and **termination**.

Born in 1990, Haskell is a functional programming language designed by a committee of experts to embody the following characteristics:

1. **Purely Functional**: Haskell is a purely functional language, meaning that all computations are performed using functions and function application.
2. **Call-by-Need** (Lazy Evaluation): values are not computed until they are actually needed. This permits us to make infinite computations and infinite lists.
3. **Strong Polymorphic and Static Typing**: every expression has a well-defined type that can be checked at compile time. It also has polymorphic typing, which allows you to write functions that can operate on values of any type as long as they meet certain requirements (type inference … indeed usually we don´t need to explicitly declare types).

3.1.1 What is a functional language?

In mathematics, functions do not have side-effects. Haskell is purely funtional, so we will see later how to manage inherently side-effectful computations (e.g. those with I/O).

3.2 Evaluation of functions

A function application ready to be performed is called a reducible expression (or **redex**). We basically can have two strategies regarding **evaluations of functions**:

- **call-by-value**: in this strategy, arguments of functions are always evaluated before evaluating the function itself - this corresponds to passing arguments by value.
- **call-by-name**:We start with the redex that is not contained in any other redex. Functions are always applied before their arguments, this corresponds to passing arguments by name.

3.2.1 Haskell is lazy: call-by-need

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. In Haskell it's used **Call-by-need** which is a *memoized version of call-by-name* where, if the function argument is evaluated, that value is stored for subsequent uses. In a "pure" (effect-free) setting, this produces the same results as call-by-name, and it is usually faster. Call-by-need is very convenient for dealing with **never-ending computations** that provide data

```
makeStream :: [l] -> [l]
makeStream l = l ++ makeStream l

ghci> i = makeStream 1
ghci> take 5 i
[1,1,1,1,1]
```

3.2.2 Currying

Currying is an important concept and refers to the fact that functions have only one argument.

```
answerToEverything = 42
```

```
complexcalc x y z = x * y *  z * answerToEverything

:t complexcalc
complexcalc : : Integer -> Integer -> Integer -> Integer
complexcalc : : Integer-> (Integer -> (Integer -> Integer))
```

The term is a reference to logician Haskell **Curry**. The alternative name Schönfinkelisation has been proposed as a reference to Moses Schönfinkel but didn't catch on.

3.2.3 Function definition

Functions are declared through a sequence of equations using **pattern matching**. Haskell has a powerful mechanism for defining custom data types called "algebraic data types". You should become familiar with how to define and use these types, as well as pattern matching.

```
first (x,_,_) = x
second (_,x,_) = x
third (_,_,z) = z
first (1,2,3) -> 1
second (1,2,3) -> 2
third (1,2,3) -> 3
```

The _ means the same thing as it does in list comprehensions. It means that we really don't care what that part is, so we just write a _.

Other useful stuff regarding functions:

- `.` is used for composing functions `(f . g)(x)` is `f(g(x))`
- `$` symbol in Haskell is called the "function application operator". It has a very low precedence, which means that it allows you to avoid using parentheses when applying functions: `f (g x)` is like: `f $ g x`. Another way to think about it is that `$` acts as a "parentheses eraser" in function chaining. It is not necessary but makes code concise and readable.

3.3 Data and type

`data` and `type` keywords are both used to define new data structures, but they have different purposes.

- A type is a set of values and it's made using the keyword `data` (and not type lol)
- A type-class is a set of types. `Show` , `Eq`, `Ord` are typeclasses.
- A type-class is a way to define operations to other types . . . very similar to **interfaces** in for example Java. They are the mechanism provided by Haskell for ad hoc polymorphism.

Haskell	Java (or similar OOP)
Type Class	Interface
Type	Class
Value	Object
Method	Method

- `data` is used when you want to define a new type you use `data`.

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

- `type` is used to define type synonyms, which are alternative names for existing types. For example, you could use `type` to define a synonym for a list of integers:

```
type IntList = [Int]`
```

- `newtype` is 'weird' since we want to make an alias 'basically identical' to another already define `type` but I want to say to ghc to consider them two different types.

Recursive Types:

```
-- Trees
data Tree a = Leaf a | Branch (Tree a) (Tree a)

Branch :: Tree a -> Tree a -> Tree a
aTree = Branch (Leaf 'a') (Branch (Leaf 'b') (Leaf 'c'))

-- Lists are recursive types
data List a = Null | Cons a (List a)
data [a] = [] | a : [a]
```

3.4 Lists

Lists in Haskell are represented by square brackets `[]` and the elements (of the same type) are separated by commas.

```
numbers = [1,3,5,7]
numbers !! 2  -> 5
null numbers -> False --tell us if the list is empty
head numbers -> 1
tail numbers -> [3,5,7]
init numbers -> [1,3,5]
last numbers -> 7
drop 2 numbers -> [5,7]
take 2 numbers -> [1,3]
elem 1 [1,2,3] -> True
elem 21 [1,2,3] -> False
```

3.4.1 Take

`take` is used to take `n` items from a list: `take n (x:xs)`.

If `n` is equal to zero, an empty list is returned. If `n` is greater than the length of the list, the entire list is returned.

3.4.2 Zip

The `zip` function is a common operation in programming languages that takes two or more lists as input and combines them into a single list of tuples, where each tuple contains corresponding elements from the input lists.

- Resulting list will have the same length as the shortest input list.
- Especially useful for when you want to combine two lists in a way or traverse two lists simultaneously.

```
zip [1,2,3,4,5] [5,5,5,5,5]  -- [(1,5),(2,5),(3,5),(4,5),(5,5)]
zip [1 .. 5] ["one", "two", "three", "four", "five"]  -- [(1,"one"),(2,"two"),(3,"three"),(4,"four"),(5,
```

Implementation:

```
zippa :: [a] -> [b] -> [(a,b)]
zippa [] _ = []
zippa _ [] = []
zippa (x:xs) (y:ys) = (x,y) : zippa (xs) (ys)
```

3.4.3 Cons

The : operator is called the `cons` operator. It is used to prepend an element to a list.

```
0 : [1, 2, 3] -> [0, 1, 2 ,3]
```

Example of a list comprehension:

```
length' xs = sum [1 | _ <- xs]
fi xs = [x | x <- xs, odd x == False]
```

3.5 Tuples

`(1,2,3)` in Haskell is a tuple (while `[1,2,3]` is a list). A tuple is a fixed-length, ordered collection of elements. While lists are homogeneous, tuples are heterogeneous. For example, `(1, "Hello", 3.14)` is a valid tuple in Haskell.

Tuples are useful for when you want to group together a fixed number of elements that may have different types, while lists are useful for when you want to store a sequence of elements that are all of the same type. `fst` and `snd` are built-in functions that operate on pairs:

```
pair = (3, 5)
firstElement = fst pair -- returns 3

pair = (3, 5)
secondElement = snd pair -- returns 5
```

3.6 Let

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

--or
let {x = 3 ; y = 12} in x+y


euler :: Point -> Point -> Float
euler (Point x1 y1) (Point x2 y2) =
    let dx = x1 - x2
        dy = y1 - y2
    in sqrt ( dx*dx + dy*dy )
```

Example with `where`:

```
bmi :: Float -> Float -> String
bmi w h
    | calc <= 18.5 = "Underweight"
    | calc <= 25.0 ="Normal"
    | calc <= 30.0 = "Overweight"
    | otherwise = "Obese"
    where calc = w/h^2
```

3.7 If

```haskell
checkSign :: Int -> String
checkSign n =
  if n >= 0
    then "Positive"
    else "Negative"
```

3.8 Equality

To make instances of the `Eq` typeclass, we need to implement the `(==)` function. Example on how to create an instance of the `Eq` typeclass:

```haskell
-- Define a custom data type called Person
data Person = Person String Int

-- Implementing Eq instance for Person
instance Eq Person where
    (Person name1 age1) == (Person name2 age2) =
        name1 == name2 && age1 == age2
```

3.9 Show

```haskell
data Queue a = Queue [a] [a]

instance (Show a) => Show (Queue a) where
  show (Queue x y) = show x ++ "|" ++ show y


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

instance Show x => Show (Tree x) where
    show Empty = "()"
    show (Leaf a) = "(" ++ show a ++ ")"
    show (Node a b) = "[" ++ show a ++ show b ++ "]"
```

Usually it is not necessary to explicitly define instances of some classes. Haskell can be quite smart and do it automatically, by using deriving.

```haskell
data Point = Point Float Float deriving (Show,Eq)
```

3.10 Ord

`Ord` typeclass is used for types that can be ordered or sorted.

```haskell
data Student = Student { rollNumber :: Int, grade :: Char }

instance Ord Student where
    compare student1 student2 = compare (rollNumber student1) (rollNumber student2)
```

3.11 The Enumeration Class

```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
```

3.12 Road to Monads

Here is a high-level summary of the steps to reach Monads. Monads are the most powerful of these type classes, and they are used for dealing with side effects like error handling or state management.

- **Foldable**: A type class that enables folding over data structures, such as lists or trees.
- **Functor**: A type class that defines a way to apply a function over a structure, preserving the structure's type.
- **Applicative Functors**: A type class that extends the Functor class to allow for sequential application of functions over multiple structures, with the ability to lift functions of multiple arguments into the context of the structures.
- **Monad**: A type class that extends the Applicative Functor class to enable sequencing of actions or computations, with the ability to chain together actions that may produce values, handle errors, or make decisions based on the results of previous actions.

Actually it's not necessary to implement all these steps to have a Monad.

3.13 Foldable

Foldable is a class used for folding. The main idea is the same as `foldl` and `foldr` for lists: given a container and a binary operation `f`, we want to apply `f` to all elements in the container. A minimal implementation of Foldable requires `foldr`.

3.13.1 Foldr

`foldr` starts from right.

```
ghci> :t foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

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

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
foldr = tfoldr

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

3.13.2 Foldl

The `foldl` function in Haskell is short for "fold left". This means that it starts at the leftmost element of a list and applies a given binary operator to each element and an accumulator value, accumulating the result as it goes. Note that in Racket it is defined with `(f x z)`.

```
ghci> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b

foldl (\acc x -> x + acc) 0 [1,2,3]
```

In `foldl (+) 0 [1,2,3]` the steps are:

- Start with an accumulator value of `0`.
- Apply `(+) 0 1`, which gives us a new accumulator value of `1`.
- Apply `(+) 1 2`, which gives us a new accumulator value of `3`.
- Apply `(+) 3 3`, which gives us a final result of `6`.

Example of multiple arguments in the lambda expression of `foldl` :

```
foldl (\acc (x, y) -> x + y + acc) 0 [(1, 2), (3, 4), (5, 6)]
```

Implementation of `elem` with `foldl`:

```
isThere y ys = foldl (\acc x -> if x == y then True else acc) False ys
```

`Foldl` can be expressed in term of foldr (id is the identity function):

```
foldl f a bs = foldr (\b g x -> g (f x b))
```

Actually `foldr` may work on infinite lists, unlike `foldl`: `foldl` starts from left to right but actually it starts to compute the recurrence from the last element (but if it doesn't exists .. that's not possible).

3.14 Functor

Functor is the class of all the types that offer a map operation Generally it's natural to make every data structure an instance of functor.

- `fmap id = id` (where id is the identity function)
- `fmap (f.g) = (fmap (f)).(fmap (g))` (homomorphism)

```
ftmap :: (a->b) -> (Tree a) -> (Tree b)
ftmap _ Empty = Empty
ftmap f (Leaf x) = (Leaf (f x))
ftmap f (Node z1 z2) = (Node (ftmap f z1) (ftmap f z2))

instance Functor Tree where
    fmap = ftmap
```

3.15 Applicative Functors

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

The `pure` method is a useful helper function for taking an ordinary value or function and putting it into a context.

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

ghci> concatMap (\x -> [x, x+1]) [1,2,3]
[1,2,2,3,3,4]
```

```
ghci> concatMap (++"?") ["a","b","c"]
"a?b?c?"

ghci> concatMap (++"") ["a","b","c"]
"abc"

ghci> concatMap (return "a") ["a","b","c"]
"aaa"


-- with concatMap, we get the standard implementation of <*> (the main op of applicative)

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

ghci> z = Node (Leaf (*3)) (Leaf (*2))
ghci> y = Node (Leaf 2) (Leaf 1)
ghci> z <*> y
[[(6)(3)][(4)(2)]]
```

Applicative instance example on a binary tree where in each node is stored data, together with the number of nodes contained in the subtree of which the current node is root.

```
data Ctree a = Cnil | Ctree a Int (Ctree a) (Ctree a) deriving (Show, Eq)

cvalue :: Ctree a -> Int
cvalue Cnil = 0
cvalue (Ctree _ x _ _) = x

cnode :: a -> Ctree a -> Ctree a -> Ctree a
cnode x t1 t2 = Ctree x ((cvalue t1) + (cvalue t2) + 1) t1 t2

cleaf :: a -> Ctree a
cleaf x = cnode x Cnil Cnil

instance Functor Ctree where
 fmap f Cnil = Cnil
 fmap f (Ctree v c t1 t2) = Ctree (f v) c (fmap f t1)(fmap f t2)

instance Foldable Ctree where
foldr f i Cnil = i
foldr f i (Ctree x _ t1 t2) = f x $ foldr f (foldr f i t2) t1

x +++ Cnil = x
Cnil +++ x = x
(Ctree x v t1 t2) +++ t = cnode x t1 (t2 +++ t)

ttconcat = foldr (+++) Cnil

ttconcmap f t = ttconcat $ fmap f t

instance Applicative Ctree where
pure = cleaf
```

```
x <*> y = ttconcmap (\f -> fmap f y) x
```

Example with a `Slist` data structure for lists that store their length.

```
data Slist a = Slist Int [a] deriving (Show, Eq)

makeSlist v = Slist (length v) v

instance Foldable Slist where
    foldr f i (Slist n xs) = foldr f i xs

instance Functor Slist where
    fmap f (Slist n xs) = Slist n (fmap f xs)

instance Applicative Slist where
 pure v = Slist 1 (pure v)
 (Slist x fs) <*> (Slist y xs) = Slist (x*y) (fs <*> xs)

instance Monad Slist where
 fail _ = Slist 0 []
 (Slist n xs) >>= f =
     makeSlist (xs >>= (\x -> let Slist n xs = f x in xs) )
```

Other example

```
instance Applicative Tree where
    pure x = (Leaf x)
    _ <*> Empty = Empty
    Leaf f1 <*> Leaf x = Leaf (f1 x)
    Node fl fr <*> Node l r =
        Node (Node (fl <*> l) (fl <*> r))
        (Node (fr <*> l) (fr <*> r))
```

3.16 Monad

Introduced by Eugenio Moggi in 1991, a monad is a kind of algebraic data type used to represent computations.

```
:t (>>=)
Monad m => m a -> (a -> m b) -> m b
```

The implementation with monad permits me to perform specific operations with `>>=`. `>>=` sequentially compose two actions, discarding any value produced by the first, like sequencing operators in imperative languages. Note that `do` is a "syntax sugar" (alternative to `>>=`) since it provides a more readable and convenient way of writing Monadic code (this is like the `begin` structure in Scheme).

```
-- Define a Monad instance for Tree.
instance Monad Tree where
  -- Define 'return' (also known as 'pure') to create a new leaf node with the given value.
  return = Leaf
  -- For an empty tree, simply return another empty tree.
  Empty >>= f = Empty
  -- For a leaf node containing value x, apply function f to x and return the result.
  Leaf x >>= f = f x
  -- For a non-leaf node with left subtree l and right subtree r,
  -- recursively apply function f to both subtrees and combine them into one larger tree using Node cons
  Node l r >>= f = Node (l >>= f) (r >>= f)
```

This implementation allows us to use standard monadic operations like bind (>>=) on trees constructed using our custom datatype.

```
type Log = [String] --remember that is just an alias
data Logger a = Logger a Log

instance (Eq a) => Eq (Logger a) where
    (Logger x _) == (Logger y _) = x == y

instance (Show a) => Show (Logger a) where
   show (Logger x s) = show x ++ "{" ++ show s ++ "}"

instance Functor Logger where
    fmap f (Logger x s) = Logger (f x) s

instance Applicative Logger where
    pure x = Logger x ["Init"]
    Logger f f_name <*> (Logger x log) =  (Logger (f x) (log ++ f_name))

instance Monad Logger where
    return = pure
    (Logger x l) >>= f =
        let Logger x' l' = f x
        in Logger x' (l ++ l')

data CashReg a = CashReg { getReceipt :: (a,Float) } deriving (Show, Eq)

getCurrentItem = fst . getReceipt
getPrice = snd . getReceipt

instance Functor CashReg where
    fmap f cr = CashReg (f $ getCurrentItem cr, getPrice cr)

instance Applicative CashReg where
    pure a = CashReg (a,0.0)
    fcr <*> cr = CashReg (getCurrentItem fcr $ getCurrentItem cr, getPrice fcr + getPrice cr)

instance Monad CashReg where
    return = pure
    cr >>= f =
        let newCr = f $ getCurrentItem cr
        in CashReg (getCurrentItem newCr, getPrice cr + getPrice newCr)
```

3.16.1 Maybe

Maybe is used to represent computations that may fail: `data Maybe a = Nothing | Just a`. It is adopted in many recent languages, to avoid NULL and limit exceptions usage: mainly because exceptions are sometimes complex to manage.

3.17 Misc

3.17.1 Modules

Haskell has a simple module system, with import, export and namespaces. Modules provide the only way to build abstract data types (ADT)

3.17.2 IO

IO is not 'compatible' with the functional philosophy. 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). `IO` is an instance of the monad class.

```
main = do { --sequence of IO actions
putStr "Please, tell me something>";
thing <- getLine; --a way to get the input
putStrLn $ "You told me \"" ++ thing ++ "\".";
}
```

```
import System.IO
import System.Environment

readfile = do {
    args <- getArgs; -- command line arguments
    handle <- openFile (head args) ReadMode;
    contents <- hGetContents handle;
    putStr contents;
    hClose handle;
}

main = readfile
```

4 Erlang

4.1 Introduction

Initially developed for telecommunication applications such as switches, Erlang is a concurrent-oriented functional language that has almost transparent distribution. It runs on an ad hoc **VM** called **BEAM**. BEAM offers many features for parallel and distributed systems, including slow performance degradation and fault-tolerance, making it very robust. Erlang has been used in several relevant industrial applications such as WhatsApp, Amazon's SimpleDB, and Facebook's chat function. One of Erlang's core principles is the "**let it crash**" principle. This principle involves designing the application with a supervision structure in place to handle errors and crashes. When an error occurs, the process responsible for the error is terminated, and supervision processes restart it and other necessary processes. Erlang also supports **hot-swapping** of code, allowing for updating application code at runtime. Processes running the previous version of the code continue to execute, while new invocations run the updated code. Erlang syntax is heavily influenced by Prolog.

4.2 Integers

Integers in Erlang are used to denote whole numbers (positive or negative). In Erlang there is no maximum size for integers and arbitrarily large whole numbers can be used. It's also easy to express integers in different bases other than 10 usng the `Base#Value` notation. If omitted, base 10 is assumed. Operations on integers are standard except that the `/` operator is used only for floating point division while the `div` operator is used for integer division.

4.3 Atoms

Atoms are like symbols in Scheme: constant literals that stands for themselves. They are similar to enumeration types in other programming languages. If the atom is not enclosed in quotes, valid characters are letters, digits, the AT symbol (`@`) and the underscore (`_`). If the atom is enclosed in single quotes, any character is allowed.

4.3.1 Booleans

The atoms `true` and `false` are just atoms already defined. The following Boolean operators are available:

- `and`
- `andalso`
- `or`
- `orelse`
- `xor`
- `not`

4.4 Variables

Variables in Erlang have some rules like:

- There is the "Single Assignment": once you've bound a variable, you cannot change its value . . . so yeah .. it's not a variable.
- They start with an **uppercase letter**, followed by uppercase and lowercase letters, integers and underscores.

4.5 Equality

In Erlang, expressions can be compared using the following operators:

```
== Equal to
/= Not equal to
=:= Exactly equal to
=/= Exactly not equal to
=< Less than or equal to
< Less than
<= Less than or equal to
```

4.6 Tuples

Tuples are used to store a fixed number of items. They can contain any type of data and they are delimited by `{ }` :

```
> element(1, {martino,piaggi}).
martino
> element(2, {martino,piaggi}).
piaggi
```

4.7 Lists

Erlang offers also **lists**, which are delimited by square brackets (like in Haskell) and their elements are comma-separated. Are very useful for list comprehension. Main lists operations:

- Using `lists:map` to square each element in a list:

```
Squares = lists:map(fun(X) -> X*X end, [1, 2, 3]).
```

Result: Squares will be `[1, 4, 9]`

- Using `lists:nth` to access elements at specific indices in a list:

```
Element1 = lists:nth(1, [3, 23, 42]).
```

Result: Element1 will be `3`

```
Element3 = lists:nth(3,[3 ,23 ,42]).
```

Result: Element3 will be `42`

- Using `lists:member` to check if an element is present in a list:

```
IsMember = lists:member(3,[3 ,23 ,42]).
```

Result: IsMember will be `true`

- Using `lists:foldr` to apply a function over all elements of a list from right-to-left:

```
Sum = lists:foldr(fun(X,Y) -> X+Y end, 0 ,[1 ,2 ,3]).
```

Result : Sum will be `6`

- Using `lists_seq:interval/2`, which is equivalent to creating a sequence of numbers from start to stop (inclusive):

```
Sequence = lists_seq:interval(1 ,5).
```

Result : Sequence will be `[1,2,3,4,5]`

4.8 Pattern Matching

Pattern matching is more expressive than Haskell. `Pattern = Expression` can be used to:

- Assign values to variables, for example: `{a, b, C} = {a, b, foo}.` is a way to bound `C` variable to atom `foo`.
- Control the execution flow of programs
- Extract values from compound data types

4.9 Functions

Erlang function definitions can't be typed directly in the Erlang shell but they are grouped together in **modules**. Examples of built-in functions are:

```
date ()
time ()
size ( {a, b, c})
atom_to_list (an_atom)
integer_to_list (2234)
tuple_to_list ({Y})
```

4.9.1 Function syntax

- The arrow `->` separates the head of the function from its body.
- The function can consists of one or more clauses, separated by semicolons `;` except for the last one which is terminated by a dot `.`
- Each clause defines the action of the function on data which matches the pattern in the head of the clause.
- Clauses are scanned in order until a match is found.
- When a match is found, variables in the head of the clause are bound.
- Variables in each clause are local and are allocated and deallocated automatically.
- The body of the clause is evaluated sequentially using commas to separate expressions.

4.9.2 Guarded Function Clauses

```
factorial (0) -> 1;
factorial (N) when N > 0 ->
N * factorial (N - 1) .
```

Erlang uses guarded function clauses to define functions. A guard is introduced using the keyword `when`. It is similar to Haskell. The guard sub-language is restricted because it must be evaluated in constant time. This means users cannot use their own predicates while using guards.

4.10 Funs

Funs are used to define **anonymous functions** like `lambda` in Scheme.

```
F = fun (Arg1, Arg2, ... ArgN) -> ... End
```

```
4> lists:foldl (fun (X,Y) -> X+Y end, 0, [1,2,3]).
6
```

Funs can be passed as usual to higher order functions: `lists:map(Square, [1,2,3] ).` returns `[1,4,9]`. To pass standard functions, we need to prefix their name with fun and state their arity: `lists:foldr(fun my_function/2, 0, [1,2,3]).`

4.11 Apply

`apply(Module, Func, Args)` applies function `Func` in module `Mod` to the arguments in the list `Args`. Any Erlang expression can be used in the arguments to apply.

```
-module(apply_example).
-export([add/2]).

add(X, Y) ->
    X + Y.

main() ->
    Args = [5, 3],
    Result = apply(apply_example, add, Args),
    io:format("Result: ~p~n", [Result]).
```

4.12 If else

Erlang's syntax does not include a specific keyword for an else clause.

```
if
    integer(X) -> ... ;
    tuple(X) -> ... ;
    true -> ... % works as an "else"
end,
```

By convention, it is common practice to end an `if` expression with the condition `true`. This ensures that there is always a fallback option in case none of the other conditions match.

4.13 Maps

In Erlang, maps are a data structure similar to hash tables which allow you to store key-value pairs. Maps support pattern matching, which makes it easy to extract specific values from them. An example:

```
% Creating a map using the #{...} syntax
Io = #{name => "Martino", age => 23}.
```

Result: `#{age => 23, name => "Martino"}`

```
% Accessing the entire map
Io.
```

Result: `#{age => 23, name => "Martino"}`

```
% Accessing a specific value from the map using its key
MyName = Io:name.
```

Result: `"Martino"`

Other example using maps:

```
proxyFun(Table) ->
    receive
        {remember, Pid, Name} ->
            proxyFun(Table#{Name => Pid});
        {question, Name, Data} ->
            #{Name := Id} = Table,
            Id ! {question,Data},
            proxyFun(Table);
        {answer, Name, Data} ->
            #{Name := Id} = Table,
            Id ! {answer, Data},
            proxyFun(Table)
    end.
```

4.14 Message passing

Erlang relies on the "Actor Model", in which actors (independent unit of computation) can only communicate through messages. Processes are represented using different actors communicating only through messages. Each actor is a lightweight process, handled by the VM: it is not mapped directly to a thread or a system process, and the VM schedules its execution. The syntax `!` is used to send anything to any other actor/process:

```

B ! {self(),{mymessage,[1,2,3,42]}}

Then in b:

receive
    {A,{mymessage,D}} -> work_on_data(D);
end
```

With Erlang library you can make very easily and quickly a client-server system. There is construct to make:

```
sleep (T) ->
    receive
        foo -> Actions1
    after
    ... Time -> Actions2
    end.
```

When creating a new process in Erlang, it is assigned a process identifier (Pid), which is known only to the parent process. A process can send a message to another process using its identifier.

- `register(Alias,Pid)` is a way to make an alias in complex applications.
- There is also `Flush()` to clear message queue.

Messages can carry data and provided that the data is applicable, variables can become bound when receiving the message.

Often **Message Passing** is used with **list comprehensions** to manage the processes:

```
% standard map
map(_, []) -> [];
map(F,[X|Xs]) -> [F(X) | map(F,Xs)].

% parallel map
pmap(F, L) ->
    Ps = [ spawn(?MODULE, execute, [F, X, self()]) || X <- L],
    [receive
         {Pid, X} -> X
     end || Pid <- Ps].

execute(F, X, Pid) ->
    Pid ! {self(), F(X)}.
```

The processes are spawn and organized using lists comprehension.

```
filterr(F, [H|T]) ->
    case F(H) of
        true  -> [H|filterr(F, T)];
        false -> filterr(F, T)
    end;
filterr(F, []) -> [].

%same as pmap but we have to discard
pfilter(F, L) ->
    Ps = [ spawn(?MODULE, execute2, [F, X, self()]) || X <- L],
    lists:foldl(fun (P,Vo) ->
                receive
                   {P,true, X} -> Vo ++ [X];
                   {P,false,_} -> Vo
                 end end, [], Ps).

execute2(F, X, Pid) ->
    case F(X) of
        true  -> Pid ! {self(), true,X};
        false -> Pid ! {self(), false,X}
    end.
```

Implementatioon of a parallel **foldl** where the binary operator `F` is associative, and `N` is the number of parallel processes in which to split the evaluation of the fold.

```
partition(L, N) ->
    M = length(L),
    Chunk = M div N,
    End = M - Chunk*(N-1),
    parthelp(L, N, 1, Chunk, End, []).

parthelp(L, 1, P, _, E, Res) ->
    Res ++ [lists:sublist(L, P, E)];
parthelp(L,N,P,C,E ,Res) ->
    R = lists:sublist(L,P,C),
    parthelp( L,N-1,P+C,C,E ,Res ++ [R]).

parfold(F,L,N)->
   Ls= partition( L,N ),
   W=[spawn(?MODULE,dofold,[self(),F,X]) || X <-Ls],
   [R|Rs]=[receive {P,V}->V end||P<-W],
   lists:foldl(F,R,Rs).

dofold(Proc,F,[X|Xs])->
     Proc ! {self(),lists:foldl(F,X,Xs)}.
```

A function which takes two list of PIDs `[x1, x2, ...]`, `[y1, y2, ...]`, having the same length, and a function `f`, and creates a different "broker" process for managing the interaction between each pair of processes `xi` and `yi`. At start, the broker process i must send its PID to xi and yi with a message `{broker, PID}`. Then, the broker i will receive messages `{from, PID, data, D}` from xi or yi, and it must send to the other one an analogous message but with the broker PID and data D modified by applying f to it.

A special stop message can be sent to a broker i that will end its activity sending the same message to xi and yi.

```
broker(X,Y,F) ->
    X ! {broker,self()},
    Y ! {broker,self()},
    receive
        {from,X,data,D} ->
            Y ! {from,self(),data,F(D)},
            broker(X,Y,F);
        {from,Y,data,D} ->
            X ! {from,self(),data,F(D)},
            broker(X,Y,F);
        stop ->
            X ! stop,
            Y ! stop,
            ok
    end.

twins([],_,_) -> ok;
twins([X|Xs],[Y|Ys],F) -> spawn(?MODULE,broker,[X,Y,F]), twins(Xs,Ys,F).
```

4.15 Timeouts

In this way we can build **sleep** function:

```
sleep (T) ->
    receive
    after
        T -> true
    end.
```

4.15.1 Alarm example

```
setAlarm(T,What)->
    spawn(?MODULE,set,[self(),T,What]),
    receive
        {Alarm} -> io:format("~p~n", [Alarm])
    end.
```

This will print out the contents of `Alarm` using Erlang's format string syntax (`~p`) and then add a newline character (`~n`). This way, when you call `setAlarm(100,ciao).`, it will print out "ciao" instead of "ok".

```
set(Pid,T,Alarm) ->
    receive
    after
        T -> Pid ! {Alarm}
    end.
```

4.16 Managing errors

In the given Erlang code, errors are managed using pattern matching and conditional statements.

```
master_loop(Count) ->
    receive
        {value, Child, V} ->
            io:format("child ~p has value ~p ~n", [Child, V]),
            Child ! {add, rand:uniform(10)},
            master_loop(Count);
        {'EXIT', Child, normal} ->
            io:format("child ~p has ended ~n", [Child]),
            if
                Count =:= 1 -> ok; % this was the last
                true -> master_loop(Count-1) % works as an "else"
            end;
        {'EXIT', Child, _} -> % "unnormal" termination
            NewChild = spawn_link(?MODULE, child, [0]),
            NewChild ! {add, rand:uniform(10)},
            master_loop(Count)
    end.

child(Data) ->
    receive
        {add,V} ->
             NewData = Data+V,
             BadChance = rand:uniform(10) < 2,
             if
                 % random error in child:
                 BadChance -> error("I'm dying");
                 % child ends naturally:
                 NewData > 30 -> ok;
                 % there is still work to do:
                 true -> the_master ! {value,self(),NewData},
                         child(NewData)
             end
     end.
```

In the `master_loop/1` function, there are two cases where errors can occur:

- when a child process terminates normally (`{EXIT, Child, normal}`)
- when a child process terminates abnormally (`{EXIT, Child, _}`).

Overall, errors in this code are handled by throwing an error message when a random condition is met in the child process, and by replacing terminated child processes with new ones in the master process.

4.17 Shell commands

Please note that in Erlang shell every instruction must be terminated by a dot `.` . `f()` is used to "forget" all possible value bounds to the assignments. `f(Variable)` if you want to unbound a specific variable.

4.18 Links and resources

Good to start: https://www.tryerlang.org/ Learn you some Erlang