Functional Programming Concepts

Caio Rodrigues Soares Silva

February 19, 2016

Contents

1 Concepts

1.1 Overview

Functional Programming

Functional Programming is all about programming with functions.

Functional Programming Features

- Pure Functions / Referential Transparency / No side effect

- Function Composition
- Lambda Functions/ Anonymous Functions
- High Order Functions
- Currying/ Partial Function Application
- Closure - Returning functions from functions
- Data Immutability
- Pattern Matching
- Lists are the fundamental data Structure

Non Essential Features:

- Static Typing
- Type Inferencing
- Algebraic Data Types

Functional Programming Design Patterns

- Curry/ Partial function application - Creating new functions by holding a parameter constant
- Closure - Return functions from functions
- Pure Functions: separate pure code from impure code.
- Function composition
- Composable functions
- High Order Functions
- MapReduce Algorithms - Split computation in multiple computers cores.
- Lazy Evaluation (aka Delayed evaluation)
- Pattern Matching
- Monads

1.2 Evaluation Strategy / Parameter Passing

1.2.1 Call-by-value

Call-by-value (aka pass-by-value, eager evaluation, strict evaluation or applicative-order evaluation) Evaluation at the point of call, before the function application application. This evaluation strategy is used by most programming languages like Python, Scheme, Ocaml and others.

```
let add2xy x y = x + x + y

add2xy (3 * 4) ( 2 + 3)
add2xy 12 5
=  12 + 12 + 5
=  29
```

Example in Python:

```
>>> def add(x, y):
...    sum = x + y
...    x = sum
...    return (x, sum)
...
>>>


>>> x = 10
>>> y = 20

# In call-by-value strategy. The function doesn't change its arguments.
#
>>> add(x, y)
(30, 30)

# They remain unchanged.
#
>>> x
10
>>> y
20
```

```
# Arguments are evaluated before the function invocation.
#
# add((10 + 20), (3 * 3))
# add(30, 9)
# (39, 39)
#
>>> add((10 + 20),(3 * 3))
(39, 39)
>>>
>>> add((10 + 20),(3 / 0))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

1.2.2 Call-by-name

Call-by-name (aka pass-by-name): It is a non-memoized lazy evaluation. Arguments are passed non evaluated, they are only evaluated when it is needed. The arguments are substituted in the function. This evaluation strategy can be inefficient when the arguments are evaluated many times.

Example:

```
// Each use of x y and replaced in the function by (3 * 4) and (2 + 3)
// respectively. The arguments are evaluated after substituted
// in the function

let add2xy x y = x + x + y
let add2xy (3 * 4) (2 + 3)
= (3 * 4) + (3 * 4) + (2 + 3)
= 12 + 12 + 5
= 29
```

Call-by-name can be simulated using thunks (functions without arguments).

Example in Python:

```
>>> def add2xy (x, y):
...     return x() + x() + y()
...
>>> add2xy (lambda : (3 * 4), lambda: (2 + 3))
```

```
29
>>>
```

Example in Scheme:

```
>  (define (add2xy x y) (+ (x) (x) (y)))
>
;; The parameter is evaluated every time it is used.
;;
> (define (displayln msg)
      (display msg)
      (newline))


> (add2xy (lambda () (displayln "Eval x") (* 3 4))
          (lambda () (displayln "Eval y") (+ 2 3))
  )
Eval x
Eval x
Eval y
29
>
```

1.2.3 Call-by-need

Call-by-need_ (aka lazy evaluation, non-strict evaluation or normal-order evaluation): Memoized lazy-evaluation. It is a memoized version of call-by-name, after its argument is evaluated, the values are stored for further computations. When the argument evaluation has no side-effects, in other words, always yields the same output when evaluated many times it produces the same result as call-by-name. This strategy is used in Haskell.

Example: In Haskell the parameters are not evaluated at the point they are passed to the function. They are only evaluated when needed.

```
> let add_xy x y z = x + y
>
> :t add_xy
add_xy :: Num a => a -> a -> t -> a
>

-- The parameter z: (3 / 0) is not evaluated
```

```
-- because it is not needed.
--
> add_xy 3 4 (3 / 0)
7
>

-- The infinite list is not fully evaluated
--
> let ones = 1:ones

> take 10 ones
[1,1,1,1,1,1,1,1,1,1]
>
> take 20 ones
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
>
```

1.2.4 Call-by-reference

- Call-by-reference (aka pass-by-reference): The function changes the value of the argument. Supported in: C++, Pascal, Python Objects.

Example in C:

```
// The function add10 changes it argument.
//
#include <stdio.h>

void add10(int * x){
  *x = *x + 10 ;
}

void main (){

  int x = 5;
  add10(&x);
  printf("x = %d\n", x);
}
```

Test:

```
$ gcc /tmp/test.c
$ ./a.out
x = 15

$ tcc -run /tmp/test.c
x = 15
```

1.2.5 References

- CSE 428: Lecture Notes 8
- Parameter-passing strategies — dProgSprog 2012 Lecture Notes
- CPS 343/543 Lecture notes: Lazy evaluation and thunks
- Evaluation strategy - Wikipedia, the free encyclopedia
- CSE 428: Lecture Notes 8 - Procedures and Functions

1.3 First-Class Function

Functions can be passed as arguments to another functions, returned from functions, stored in variables and data structures and built at run time. The majority of languages supports first-class functions like Scheme, Javascript, Python, Haskell, ML, OCaml and many others some exceptions are C, Java, Matlab (Octave open source implementation), Bash, and Forth.

Examples:

- Python:

The function f is passed as argument to the derivate function that returns a new function named _, that computes the derivate of f at x.

```
def derivate (f, dx=1e-5):
    def _(x):
        return (f(x+dx) - f(x))/dx
    return _

 #  Algebraic derivate:
 #
 #  df(x) = 2*x - 3
 #
```

```
>>> def f(x): return x**2 - 3*x + 4
...

 # Numerical derivate of f
>>> df = derivate(f)
>>>

  # Algebraic derivate of f
>>> def dfa (x): return 2*x - 3
...
>>>

 ;; Functions can be stored in variables
>>> func = f
>>> func(5)
14
>>>

>>> df = derivate(f)
>>> df(3)
3.000009999887254
>>> df(4)
5.000009999633903
>>>

>>> dfa(3)
3
>>> dfa(4)
5
>>>


>>> f(3)
4
>>> f(10)
74
>>>
```

See also:
Many examples of first class functions in several languages.

- First-class functions - Rosetta Code
- First-class Functions in Scientific Programming
- Functional programming in R

1.4 Pure Functions

Pure functions:

- Are functions without side effects, like mathematical functions.
- For the same input the functions always returns the same output.
- The result of any function call is fully determined by its arguments.
- Pure functions don't rely on global variable and don't have internal states.
- They don't do IO, i.e .:. don't print, don't write a file ...
- Pure functions are stateless
- Pure functions are deterministic

Why Pure Functions:

- Composability, one function can be connected to another.
- Can run in parallel, multi-threading, multi-core, GPU and distributed systems.
- Better debugging and testing.
- Predictability

Example of pure functions

```
def min(x, y):
    if x < y:
        return x
    else:
        return y
```

Example of impure function

- Impure functions doesn't have always the same output for the same
- Impure functions does IO or has Hidden State or Global Variables

```
exponent = 2

def powers(L):
    for i in range(len(L)):
        L[i] = L[i]**exponent
    return L
```

The function min is pure. It always produces the same result given the same inputs and it does not affect any external variable.

The function powers is impure because it not always gives the same output for the same input, it depends on the global variable exponent:

```
>>> exponent = 2
>>>
>>> def powers(L):
...     for i in range(len(L)):
...         L[i] = L[i]**exponent
...     return L
...
>>> powers([1, 2, 3])
[1, 4, 9]
>>> exponent = 4
>>> powers([1, 2, 3])  # (It is impure since it doesn't give the same result )
[1, 16, 81]
>>>
```

Another example, purifying an impure Language:

```
>>> lst = [1, 2, 3, 4]  # An pure function doesn't modify its arguments.
>>>                     # therefore lst reverse is impure
>>> x = lst.reverse()
>>> x
>>> lst
[4, 3, 2, 1]

>>> lst.reverse()
>>> lst
[1, 2, 3, 4]
```

Reverse list function purified:

```
>>> lst = [1, 2, 3, 4]
>>>
>>> def reverse(lst):
...     ls = lst.copy()
...     ls.reverse()
...     return ls
...
>>>
>>> reverse(lst)
[4, 3, 2, 1]
>>> lst
[1, 2, 3, 4]
>>> reverse(lst)
[4, 3, 2, 1]
>>> lst
[1, 2, 3, 4]
```

1.5 Closure

Closure is a function that remembers the environment at which it was created.

```
>>> x = 10

 # The function adder remembers the environment at which it was created
 # it remembers the value of x
 #
def make_adder(x):
    def adder(y):
        return x + y
    return adder

>>> add5 = make_adder(5)
>>> add10 = make_adder(10)
>>>
>>> add5(4)
9
>>> list(map(add5, [1, 2, 3, 4, 5]))
```

```
[6, 7, 8, 9, 10]

>>> x
10
>>>

>>> list(map(add10, [1, 2, 3, 4, 5]))
[11, 12, 13, 14, 15]

 #

def make_printer(msg):
    def printer():
        print(msg)
    return printer

>>> p1 = make_printer ("Hello world")
>>> p2 = make_printer ("FP programming Rocks!!")
>>>
>>> p1()
Hello world
>>> p2()
FP p

 # Mutable state with closure

idx = 100

def make_counter():
    idx = -1
    def _():
        nonlocal idx
        idx = idx + 1
        return idx
    return _

>>> idx = 100
>>> counter1 = make_counter()
>>> counter1()
0
```

```
>>> counter1()
1
>>> counter1()
2
>>> counter1()
3

>>> idx
100
>>> counter2 = make_counter ()
>>> counter2()
0
>>> counter2()
1
>>> counter2()
2

>>> counter1()
5
>>>

>>> del make_counter
>>> make_counter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'make_counter' is not defined
>>>
>>> counter1()
6
>>> counter1()
7
```

Example of closure in Clojure:

```
(defn make-adder [x]
   (fn [y] (+ x y)))

user=> (def add5 (make-adder 5))
#'user/add5
user=>
```

```
user=> (def add10 (make-adder 10))
#'user/add10
user=>
user=> (add5 10)
15
user=> (add10 20)
30
user=> (map (juxt add5 add10)  [1 2 3 4 5 6])
([6 11] [7 12] [8 13] [9 14] [10 15] [11 16])
user=>

(defn make-printer [message]

  (fn [] (println message)))

user=> (def printer-1 (make-printer "Hello world"))
#'user/printer-1
user=>
user=> (def printer-2 (make-printer "Hola Mundo"))
#'user/printer-2
user=>
user=> (printer-1)
Hello world
nil
user=> (printer-2)
Hola Mundo
nil
user=>
```

Example of closure in F# (F sharp):

```
let make_adder x =
    fun y -> x + y

val make_adder : x:int -> y:int -> int

> let add5 = make_adder 5 ;;

val add5 : (int -> int)
```

```
> let add10 = make_adder 10 ;;

val add10 : (int -> int)

> add5 20 ;;
val it : int = 25
>
- add10 30 ;;
val it : int = 40
>
- List.map add5 [1 ; 2; 3; 4; 5; 6] ;;
val it : int list = [6; 7; 8; 9; 10; 11]
>

//  As F# have currying like OCaml and Haskell
//  it could be also be done as
//

- let make_adder x y = x + y ;;

val make_adder : x:int -> y:int -> int

> let add10 = make_adder 10 ;;

val add10 : (int -> int)

> add10 20 ;;
val it : int = 30
>
```

1.6 Currying and Partial Application

1.6.1 Currying

Currying is the decomposition of a function of multiples arguments in a chained sequence of functions of a single argument. The name currying comes from the mathematician Haskell Curry who developed the concept of curried functions.

In Haskell, Standard ML, OCaml and F# all functions are curryfied by default:

```
    f (x, y) = 10*x - 3*y

    f (4, 3)  = 10* 4 - 3*3 = 40 - 9 = 31
    f (4, 3)  = 31

In the curried form becomes:

     g(x) = (x -> y -> 10 * x - 3*y)

To evaluate f(4, 3):

    h(y)  = (x -> y -> 10 * x - 3*y) 4
          = ( y -> 10 * 4 -  3*y )
          =  y -> 40 - 3*y

    h(3)  = (y -> 40 - 3*y) 3
          = 40 - 3*3
          = 31

Or:
    (x -> y -> 10 * x - 3*y) 4 3
      = (x -> (y -> 10 * x - 3*y)) 4 3
      = ((x -> (y -> 10 * x - 3*y)) 4) 3
      = (y -> 10 * 4 - 3 * y) 3
      = 10 * 4 - 3 * 3
      = 31
```

The same function h(y) can be reused: applied to another arguments, used in mapping, filtering and another higher order functions.

```
Ex1
    h(y) = (y -> 40 - 3*y)

    h(10) = 40 - 3*10 = 40 - 30 = 10

Ex2
    map(h, [2, 3, 4])
      = [h 2, h 3, h 4]
      = [(y -> 40 - 3*y) 2, (y -> 40 - 3*y) 3, (y -> 40 - 3*y) 4]
      = [34, 31, 28]
```

Example in Haskell GHCI

```
> let f x y = 10 * x - 3 * y
> :t f
f :: Num a => a -> a -> a
>
> f 4 3
31
> let h_y = f 4
> :t h_y
h_y :: Integer -> Integer
>
> h_y 3
31
> map h_y [2, 3, 4]
[34,31,28]
>

> -- It is evaluated as:

> ((f 4) 3)
31
>

{-
   The function f can be also seen in this way
-}

> let f' = \x -> \y -> 10 * x - 3 * y
>

> :t f'
f' :: Integer -> Integer -> Integer
>

> f' 4 3
31
>

> (f' 4 ) 3
```

```
31
>

> let h__x_is_4_of_y = f' 4

> h__x_is_4_of_y 3
31
>
{-
    (\x -> \y -> 10 * x - 3 * y) 4 3
    =  (\x -> (\y -> 10 * x - 3 * y) 4) 3
    =  (\y -> 10 * 4 - 3 * y) 3
    =  (10 * 4 - 3 * 3)
    =  40 - 9
    =  31
-}
> (\x -> \y -> 10 * x - 3 * y) 4 3
31
>

> ((\x -> (\y -> 10 * x - 3 * y)) 4) 3
31
>


{-
Curried functions are suitable for composition, pipelining
(F#, OCaml with the |> operator),  mapping/ filtering operations,
and to create new function from previous defined increasing code reuse.

-}

> map (f 4) [2, 3, 4]
[34,31,28]
>

> map ((\x -> \y -> 10 * x - 3 * y) 4) [2, 3, 4]
[34,31,28]
>
```

```
> -- -----------------

> let f_of_x_y_z x y z = 10 * x + 3 * y + 4 * z
>

> :t f_of_x_y_z
f_of_x_y_z :: Num a => a -> a -> a -> a

> f_of_x_y_z 2 3 5
49
>

> let g_of_y_z = f_of_x_y_z 2

> :t g_of_y_z
g_of_y_z :: Integer -> Integer -> Integer
>

> g_of_y_z 3 5
49
>

> let h_of_z = g_of_y_z 3
> :t h_of_z
h_of_z :: Integer -> Integer
>

> h_of_z 5
49
>

> -- So it is evaluated as
> (((f_of_x_y_z 2) 3) 5)
49
>
```

Example in Python 3

```
 # In Python, the functions are not curried by default as in Haskell,
```

```
 # Standard ML, OCaml and F#
 #
>>> def f(x, y): return 10 * x - 3*y

>>> f(4, 3)
    31

 # However the user can create the curried form of the function f:

>>> curried_f = lambda x: lambda y: 10*x - 3*y

>>> curried_f(4)
    <function __main__.<lambda>.<locals>.<lambda>>

>>> curried_f(4)(3)
    31

>>> h_y = curried_f(4) # x = 4 constant

>>> h_y(3)
    31

>>> h_y(5)
    25

>>> mapl = lambda f_x, xs: list(map(f_x, xs))

>>> mapl(h_y, [2, 3, 4])
    [34, 31, 28]

 # Or

>>> mapl(curried_f(4), [2, 3, 4])
    [34, 31, 28]

 # Without currying the mapping would be:

>>> mapl(lambda y: f(4, y), [2, 3, 4])
    [34, 31, 28]
```

```
   ########################################

>> f_of_x_y_z = lambda x, y, z: 10 * x + 3 * y + 4 * z

 ## Curried form:

>>> curried_f_of_x_y_z = lambda x: lambda y: lambda z: 10 * x + 3 * y + 4 * z

>>> f_of_x_y_z (2, 3, 5)
    49

>>> curried_f_of_x_y_z (2)(3)(5)
    49

>>> g_of_y_z = curried_f_of_x_y_z(2)

>>> g_of_y_z
    <function __main__.<lambda>.<locals>.<lambda>>

>>> g_of_y_z (3)(5)
    49


>>> h_of_z = g_of_y_z(3)

>>> h_of_z
    <function __main__.<lambda>.<locals>.<lambda>.<locals>.<lambda>>

>>> h_of_z(5)
    49
```

Example in Ocaml and F#

```
# let f x y = 10 * x - 3 * y ;;
val f : int -> int -> int = <fun>

# f 4 3 ;;
- : int = 31

# f 4 ;;
```

```
- : int -> int = <fun>

# (f 4) 3 ;;
- : int = 31
#

# let h_y = f 4 ;;
val h_y : int -> int = <fun>

# h_y 3 ;;
- : int = 31
#

# List.map h_y [2; 3; 4] ;;
- : int list = [34; 31; 28]
#

# List.map (f 4) [2; 3; 4] ;;
- : int list = [34; 31; 28]

# let f' = fun x -> fun y -> 10 * x - 3 * y ;;
val f' : int -> int -> int = <fun>

# (f' 4) 3 ;;
- : int = 31

# (fun x -> fun y -> 10 * x - 3 * y) 4 3 ;;
- : int = 31
#

# List.map ((fun x -> fun y -> 10 * x - 3 * y) 4) [2; 3; 4] ;;
- : int list = [34; 31; 28]
```

1.6.2 Partial Application

A function of multiple arguments is converted into a new function that takes fewer arguments, some arguments are supplied and returns function with signature consisting of remaining arguments. **Partially applied*** functions must not be confused with ***currying**.

Example in Python:

```
>>> from functools import partial

>>> def f(x, y, z): return 10 * x + 3 * y + 4 * z

>>> f(2, 3, 5)
    49

>>> f_yz = partial(f, 2) # x = 2
>>> f_yz(3, 5)
    49

>>> f_z = partial(f_yz, 3)

>>> f_z(5)
    49

>>> partial(f, 2, 3)(5)
    49

>>> list(map(partial(f, 2, 3), [2, 3, 5]))
    [37, 41, 49]

#
# Alternative implementation of partial
#
def partial(f, *xs):
    return lambda x: f( * (tuple(xs) + (x,)))

>>> list(map(partial(f, 2, 3), [2, 3, 5]))
    [37, 41, 49]
>>>
```

In languages like Haskell, Standard ML, OCaml and F# currying is similar to partial application.

Example in OCaml:

```
# let f x y z = 10 * x + 3 *y + 4 * z ;;
val f : int -> int -> int -> int = <fun>
#
```

```
# (f 2 3) ;;
- : int -> int = <fun>

# let f_z = f 2 3 ;;
val f_z : int -> int = <fun>

# f_z 5 ;;
- : int = 49
#

(** Write (f 2 3) is the same as write (f 2)(3)  *)
# List.map (f 2 3) [2; 3; 5] ;;
- : int list = [37; 41; 49]
#
```

See also:

- Java.next: Currying and partial application
- Partial application - Wikipedia
- What's Wrong with Java 8: Currying vs Closures

1.7 Lazy Evaluation

"Lazy evaluation" means that data structures are computed incrementally, as they are needed (so the trees never exist in memory all at once) parts that are never needed are never computed. Haskell uses lazy evaluation by default.

Example in Haskell:

```
> let lazylist = [2..1000000000]
>
> let f x = x^6
>
> take 5 lazylist
[2,3,4,5,6]
>
>
> {- Only the terms needed are computed. -}
> take 5 ( map f lazylist )
```

```
[64,729,4096,15625,46656]
>
```

Example in Python:

- Python uses eager evaluation by default. In order to get lazy evaluation in python the programmer must use iterators or generators. The example below uses generator.

```
def lazy_list():
    """ Infinite list """
    x = 0
    while True:
        x += 2
        yield x


>>> gen = lazy_list()
>>> next(gen)
2
>>> next(gen)
4
>>> next(gen)
6
>>> next(gen)
8
>>> next(gen)
10
>>>

def take(n, iterable):
    return [next(iterable) for i in range(n)]

def mapi(func, iterable):
    while True:
        yield func(next(iterable))

f = lambda x: x**5

>>> take(5, lazy_list())
```

```
[2, 4, 6, 8, 10]
>>> take(10, lazy_list())
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
>>>

>>> take(5, mapi(f, lazy_list()))
[32, 1024, 7776, 32768, 100000]
>>>
>>> take(6, mapi(f, lazy_list()))
[32, 1024, 7776, 32768, 100000, 248832]
>>>
```

1.8 Tail Call Optimization and Tail Recursive Functions

1.8.1 Tail Call

A tail call is a function call which is the last action performed by a function.

Examples of tail calls and non tail calls:

Example 1: Tail call

```
def func1(x):
    return  tail_call_function (x * 2) # It is a tail call
```

Example 2: Tail recusive call or tail recursive function.

```
def tail_recursive_call (n, acc);
    if n = 0:
       return acc
       return tai_recursive_acll (n - 1, n * acc) # Tail recursive call, the
                                                  # function call is the last
                                                  # thing the function does.
```

Example 3: Non tail call

```
def non_tail_call_function(x):
    return 1 + non_tail_call_function (x + 3)
```

1.8.2 Tail Call Optimization

Tail call optimization - TCO. (aka. tail call elimination - TCE or tail recursion elimination - TRE) is a optimization that replaces calls in tail positions

with jumps which guarantees that loops implemented using recursion are executed in constant stack space. {Schinz M. and Odersky M. - 2001}

Without tail call optimization each recursive call creates a new stack frame by growing the execution stack. Eventually the stack runs out of space and the program has to stop. To support iteration by recursion functional languages need tail call optimization. {Schwaighofer A. 2009} [1]

If the language doesn't support TCO it is not possible to perform recursion safely. A big number of calls will lead to a stack overflow exception and the program will crash unexpectedly .

Sometimes non tail recursive functions can be changed to tail recursive by adding a new function with extra parameters (accumulators) to store partial results (state).

Languages with TCO support:

- Scheme
- Common Lisp
- Haskell
- Ocaml
- F# (F sharp)
- C# (C sharp)
- Scala
- Erlang

Languages without TCO support:

- Python [2] , [3]
- Ruby
- Java (Note: The JVM doesn't support TCO)
- Clojure

[1] Schwaighofer A. **Tail Call Optimization in the Java HotSpot™ VM.** Available at: `http://www.ssw.uni-linz.ac.at/Research/Papers/Schwaighofer09Master/schwaighofer09master.pdf`

[2] Neopythonic: Tail Recursion Elimination

[3] Neopythonic: Final Words on Tail Calls

- JavaScript
- R [4]
- Elisp - Emacs Lisp

1.8.3 Summary

1. To perform recursion safely a language must support TCO - Tail Call Optimization.
2. Even if there is TCO support a non tail recursive function can lead to an unexpected stack overflow.
3. Recursion allow greater expressiveness and many algorithms are better expressed with recursion.
4. Recursion must be replaced by loops constructs in languages that doesn't support TCO.

1.8.4 Examples

Example of non tail recursive function in Scheme (GNU Guile):

```
(define (factorial n)
    (if (or (= n 0) (= n 1))
        1
        (* n  (factorial (- n 1)))))

> (factorial 10)
$1 = 3628800
>

;;  For a very big number of iterations, non tail recursive functions
;;  will cause a stack overflow.
;;
> (factorial 20000000)
warnings can be silenced by the --no-warnings (-n) option
Aborted (core dumped)

;;
```

[4] Tail recursion on R Statistical Environment - Stack Overflow

```
;; This execution requires 5 stack frames
;;
;;  (factorial 5)
;;  (* 5 (factorial 4))
;;  (* 5 (* 4 (factorial 3)))
;;  (* 5 (* 4 (3 * (factorial 2))))
;;  (* 5 (* 4 (* 3 (factorial 2))))
;;  (* 5 (* 4 (* 3 (* 2 (factorial 1)))))
;;
;;  (* 5 (* 4 (* 3 (* 2 1))))
;;  (* 5 (* 4 (* 3 2)))
;;  (* 5 (* 4 6))
;;  (* 5 24)
;;  120
;;
;;
;;
;;
> ,trace (factorial 5)
trace: |  (#<procedure 99450c0> #(#<directory (guile-user) 95c3630> ...))
trace: |  #(#<directory (guile-user) 95c3630> factorial)
trace: (#<procedure 9953350 at <current input>:8:7 ()>)
trace: (factorial 5)
trace: |  (factorial 4)
trace: |  |  (factorial 3)
trace: |  |  |  (factorial 2)
trace: |  |  |  |  (factorial 1)
trace: |  |  |  |  1
trace: |  |  |  2
trace: |  |  6
trace: |  24
trace: 120
>

;;
;; It requires 10 stack frames
;;
;;
> ,trace (factorial 10)
trace: |  (#<procedure 985cbd0> #(#<directory (guile-user) 95c3630> ...))
```

```
trace: |  #(#<directory (guile-user) 95c3630> factorial)
trace: (#<procedure 9880800 at <current input>:6:7 ()>)
trace: (factorial 10)
trace: |  (factorial 9)
trace: |  |  (factorial 8)
trace: |  |  |  (factorial 7)
trace: |  |  |  |  (factorial 6)
trace: |  |  |  |  |  (factorial 5)
trace: |  |  |  |  |  |  (factorial 4)
trace: |  |  |  |  |  |  |  (factorial 3)
trace: |  |  |  |  |  |  |  |  (factorial 2)
trace: |  |  |  |  |  |  |  |  |  (factorial 1)
trace: |  |  |  |  |  |  |  |  |  1
trace: |  |  |  |  |  |  |  |  2
trace: |  |  |  |  |  |  |  6
trace: |  |  |  |  |  |  24
trace: |  |  |  |  |  120
trace: |  |  |  |  720
trace: |  |  |  5040
trace: |  |  40320
trace: |  362880
trace: 3628800
>
```

This function can be converted to a tail recursive function by using an accumulator:

```
(define (factorial-aux n acc)
    (if (or (= n 0) (= n 1))
        acc
        (factorial-aux (- n 1) (* n acc))))

> (factorial-aux 6 1)
$1 = 720

> ,trace (factorial-aux 6 1)
trace: |  (#<procedure 9becf10> #(#<directory (guile-user) 984c630> ...))
trace: |  #(#<directory (guile-user) 984c630> factorial-aux)
trace: (#<procedure 9bec320 at <current input>:26:7 ()>)
trace: (factorial-aux 6 1)
```

```
trace: (factorial-aux 5 6)
trace: (factorial-aux 4 30)
trace: (factorial-aux 3 120)
trace: (factorial-aux 2 360)
trace: (factorial-aux 1 720)
trace: 720
scheme@(guile-user)>

> (define (factorial2 n) (factorial-aux n 1))

scheme@(guile-user)> (factorial2 5)
$3 = 120

;; This function could also be implemented in this way:
;;
;;
(define (factorial3 n)
    (define (factorial-aux n acc)
        (if (or (= n 0) (= n 1))
            acc
            (factorial-aux (- n 1) (* n acc))))
    (factorial-aux n 1))

> (factorial3 6)
$4 = 720

> (factorial3 5)
$5 = 120
```

Example: Summation of a range of numbers:

```
;;  Non tail recursive function:
;;
(define (sum-ints a b)
    (if (> a b)
        0
        (+ a (sum-ints (+ a 1) b))))


;;
```

```
;; Using the trace command is possible to notice the growing amount of
;; stack frame In this case it requires 11 stack frames.

> ,trace (sum-ints 1 10)
trace: |  (#<procedure 9c42420> #(#<directory (guile-user) 984c630> ...))
trace: |  #(#<directory (guile-user) 984c630> sum-ints)
trace: (#<procedure 9c4b8c0 at <current input>:56:7 ()>)
trace: (sum-ints 1 10)
trace: |  (sum-ints 2 10)
trace: |  |  (sum-ints 3 10)
trace: |  |  |  (sum-ints 4 10)
trace: |  |  |  |  (sum-ints 5 10)
trace: |  |  |  |  |  (sum-ints 6 10)
trace: |  |  |  |  |  |  (sum-ints 7 10)
trace: |  |  |  |  |  |  |  (sum-ints 8 10)
trace: |  |  |  |  |  |  |  |  (sum-ints 9 10)
trace: |  |  |  |  |  |  |  |  |  (sum-ints 10 10)
trace: |  |  |  |  |  |  |  |  |  |  (sum-ints 11 10)
trace: |  |  |  |  |  |  |  |  |  |  0
trace: |  |  |  |  |  |  |  |  |  10
trace: |  |  |  |  |  |  |  |  19
trace: |  |  |  |  |  |  |  27
trace: |  |  |  |  |  |  34
trace: |  |  |  |  |  40
trace: |  |  |  |  45
trace: |  |  |  49
trace: |  |  52
trace: |  54
trace: 55


;;  Stack Overflow Error
;;
> (sum-ints 1 10000)
> <unnamed port>:4:13: In procedure sum-ints:
<unnamed port>:4:13: Throw to key ‘vm-error’ with args ‘(vm-run "VM: Stack overflow" ()

;;
;; Safe summation
;;
```

```
(define (sum-ints-aux a b acc)
    (if (> a b)
        acc
        (sum-ints-aux (+ a 1) b (+ a acc))))

(define (sum-ints-aux a b acc)
    (if (> a b)
        acc
        (sum-ints-aux (+ a 1) b (+ a acc))))

> (sum-ints-aux 1 10 0)
$4 = 55
>

> (sum-ints-aux 1 10000 0)
$6 = 50005000

;;
;; It uses only one stack frame each call
;;
> ,trace (sum-ints-aux 1 10 0)
trace: |  (#<procedure 985a270> #(#<directory (guile-user) 93fd630> ...))
trace: |  #(#<directory (guile-user) 93fd630> sum-ints-aux)
trace: (#<procedure 98646a0 at <current input>:31:7 ()>)
trace: (sum-ints-aux 1 10 0)
trace: (sum-ints-aux 2 10 1)
trace: (sum-ints-aux 3 10 3)
trace: (sum-ints-aux 4 10 6)
trace: (sum-ints-aux 5 10 10)
trace: (sum-ints-aux 6 10 15)
trace: (sum-ints-aux 7 10 21)
trace: (sum-ints-aux 8 10 28)
trace: (sum-ints-aux 9 10 36)
trace: (sum-ints-aux 10 10 45)
trace: (sum-ints-aux 11 10 55)
trace: 55
>

;; It can also be implemented in this way:
;;
```

```
(define (sum-ints-safe a b)
    (define (sum-ints-aux a b acc)
        (if (> a b)
            acc
            (sum-ints-aux (+ a 1) b (+ a acc))))
    (sum-ints-aux a b 0))

> scheme@(guile-user)> (sum-ints-safe 1 1000)
$2 = 500500


> (sum-ints-safe 1 10000)
$7 = 50005000

>  (sum-ints-safe 1 100000)
$8 = 5000050000
scheme@(guile-user)>
```

Example: of summation in a language without TCO: python

```
def sum_ints_aux (a, b, acc):
    if a > b:
       return acc
    else:
       return sum_ints_aux (a + 1, b, a + acc)

 # Until now works
>>>
>>> sum_ints_aux(1, 10, 0)
55
>>> sum_ints_aux(1, 100, 0)
5050

 # Now it is going to fail: Stack Overflow!

>>> sum_ints_aux(1, 1000, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in sum_ints_aux
  File "<stdin>", line 5, in sum_ints_aux
```

```
...
  File "<stdin>", line 5, in sum_ints_aux
  File "<stdin>", line 2, in sum_ints_aux
RuntimeError: maximum recursion depth exceeded in comparison
>>>

#  Solution: Turn the recursion into a loop:
#
def sum_ints (a, b):
    x  = a
    acc = 0

    while x < b:

        x = x + 1
        acc = acc + x

    return acc + 1

>> sum_ints (1, 1000)
    500500

>> sum_ints (1, 10000)
    50005000
```

Example: Implementing map with tail recursion.

```
(define (map2 f xs)
   (if (null? xs)
       '()
        (cons  (f (car xs))
               (map2 f (cdr xs)))))


(define (inc x) (+ x 1))


;; It will eventually lead to an stack overflow for a big list.
;;
> ,trace (map2 inc '(1 2 3))
```

```
trace: |  (#<procedure 9e14500> #(#<directory (guile-user) 984c630> ...))
trace: |  #(#<directory (guile-user) 984c630> map2 inc (1 2 3))
trace: (#<procedure 9e48360 at <current input>:109:7 ()>)
trace: (map2 #<procedure inc (x)> (1 2 3))
trace: |  (inc 1)
trace: |  2
trace: |  (map2 #<procedure inc (x)> (2 3))
trace: |  |  (inc 2)
trace: |  |  3
trace: |  |  (map2 #<procedure inc (x)> (3))
trace: |  |  |  (inc 3)
trace: |  |  |  4
trace: |  |  |  (map2 #<procedure inc (x)> ())
trace: |  |  |  ()
trace: |  |  (4)
trace: |  (3 4)
trace: (2 3 4)
```

```
,trace (map2 inc '(1 2 3 4 5 6 7 8 9))
trace: |  (#<procedure 9cdcb00> #(#<directory (guile-user) 984c630> ...))
trace: |  #(#<directory (guile-user) 984c630> map2 inc (1 2 3 4 5 6 ...))
trace: (#<procedure 9ceebf0 at <current input>:104:7 ()>)
trace: (map2 #<procedure inc (x)> (1 2 3 4 5 6 7 8 9))
trace: |  (inc 1)
trace: |  2
trace: |  (map2 #<procedure inc (x)> (2 3 4 5 6 7 8 9))
trace: |  |  (inc 2)
trace: |  |  3
trace: |  |  (map2 #<procedure inc (x)> (3 4 5 6 7 8 9))
trace: |  |  |  (inc 3)
trace: |  |  |  4
trace: |  |  |  (map2 #<procedure inc (x)> (4 5 6 7 8 9))
trace: |  |  |  |  (inc 4)
trace: |  |  |  |  5
trace: |  |  |  |  (map2 #<procedure inc (x)> (5 6 7 8 9))
trace: |  |  |  |  |  (inc 5)
trace: |  |  |  |  |  6
trace: |  |  |  |  |  (map2 #<procedure inc (x)> (6 7 8 9))
```

```
trace: |  |  |  |  |  |  (inc 6)
trace: |  |  |  |  |  |  7
trace: |  |  |  |  |  |  (map2 #<procedure inc (x)> (7 8 9))
trace: |  |  |  |  |  |  |  (inc 7)
trace: |  |  |  |  |  |  |  8
trace: |  |  |  |  |  |  |  (map2 #<procedure inc (x)> (8 9))
trace: |  |  |  |  |  |  |  |  (inc 8)
trace: |  |  |  |  |  |  |  |  9
trace: |  |  |  |  |  |  |  |  (map2 #<procedure inc (x)> (9))
trace: |  |  |  |  |  |  |  |  |  (inc 9)
trace: |  |  |  |  |  |  |  |  |  10
trace: |  |  |  |  |  |  |  |  |  (map2 #<procedure inc (x)> ())
trace: |  |  |  |  |  |  |  |  |  ()
trace: |  |  |  |  |  |  |  |  (10)
trace: |  |  |  |  |  |  |  (9 10)
trace: |  |  |  |  |  |  (8 9 10)
trace: |  |  |  |  |  (7 8 9 10)
trace: |  |  |  |  (6 7 8 9 10)
trace: |  |  |  (5 6 7 8 9 10)
trace: |  |  (4 5 6 7 8 9 10)
trace: |  (3 4 5 6 7 8 9 10)
trace: (2 3 4 5 6 7 8 9 10)


(define (map-aux f xs acc)

    (if (null? xs)

        (reverse acc)

        (map-aux f
                 (cdr xs)
                 (cons (f (car xs))
                       acc)
        )
      )
     )

> ,trace (map-aux inc '(1 2 3 4 5) '())
trace: |  (#<procedure 9e0c420> #(#<directory (guile-user) 984c630> ...))
```

```
trace: |  #(#<directory (guile-user) 984c630> map-aux inc (1 2 3 4 5))
trace: (#<procedure 9e4c070 at <current input>:180:7 ()>)
trace: (map-aux #<procedure inc (x)> (1 2 3 4 5) ())
trace: |  (inc 1)
trace: |  2
trace: (map-aux #<procedure inc (x)> (2 3 4 5) (2))
trace: |  (inc 2)
trace: |  3
trace: (map-aux #<procedure inc (x)> (3 4 5) (3 2))
trace: |  (inc 3)
trace: |  4
trace: (map-aux #<procedure inc (x)> (4 5) (4 3 2))
trace: |  (inc 4)
trace: |  5
trace: (map-aux #<procedure inc (x)> (5) (5 4 3 2))
trace: |  (inc 5)
trace: |  6
trace: (map-aux #<procedure inc (x)> () (6 5 4 3 2))
trace: (reverse (6 5 4 3 2))
trace: (2 3 4 5 6)

;; Finally
;;

(define (map-safe f xs)
        (map-aux f xs '()))

> (map-safe inc '(1 2 3 3 4 5))
$14 = (2 3 4 4 5 6)
```

Example in F#:

```
> let inc x = x +  1 ;;

val inc : x:int -> int

let rec map_aux f xs acc =
    match xs with
    | []    ->  List.rev acc
    | hd::tl ->  map_aux f tl ((f hd)::acc)
```

```
;;

val map_aux : f:('a -> 'b) -> xs:'a list -> acc:'b list -> 'b list

> map_aux inc [1; 2; 3; 4; 5] [] ;;
val it : int list = [2; 3; 4; 5; 6]

let map2 f xs =
    map_aux f xs [] ;;

val map2 : f:('a -> 'b) -> xs:'a list -> 'b list

> map2 inc [1; 2; 3; 4; 5] ;;
val it : int list = [2; 3; 4; 5; 6]
>

//  map_aux without pattern matching
//
let rec map_aux f xs acc =
    if List.isEmpty xs
    then  List.rev acc
    else (let hd, tl = (List.head xs, List.tail xs) in
        map_aux f tl ((f hd)::acc))
;;

val map_aux : f:('a -> 'b) -> xs:'a list -> acc:'b list -> 'b list

>  map2 inc [1; 2; 3; 4; 5] ;;
val it : int list = [2; 3; 4; 5; 6]
>

//  Another way:
//
//
let map3  f xs =

    let rec map_aux f xs acc =
        match xs with
        | []    ->  List.rev acc
        | hd::tl ->  map_aux f tl ((f hd)::acc)
```

```
    in map_aux f xs []

;;

val map3 : f:('a -> 'b) -> xs:'a list -> 'b list

> map3 inc [1; 2; 3; 4; 5; 6] ;;
val it : int list = [2; 3; 4; 5; 6; 7]
>
```

Example: Tail recursive filter function.

```
let rec filter_aux f xs acc =
    match xs with
    | []      ->  List.rev acc
    | hd::tl  ->  if (f hd)
                  then  filter_aux f tl (hd::acc)
                  else  filter_aux f tl acc
;;

val filter_aux : f:('a -> bool) -> xs:'a list -> acc:'a list -> 'a list

> filter_aux (fun x -> x % 2 = 0) [1; 2; 3; 4; 5; 6; 7; 8] [] ;;
val it : int list = [2; 4; 6; 8]
>

let filter f xs =
    filter_aux f xs []
;;
val filter : f:('a -> bool) -> xs:'a list -> 'a list

> filter (fun x -> x % 2 = 0) [1; 2; 3; 4; 5; 6; 7; 8] ;;
val it : int list = [2; 4; 6; 8]
>
```

1.8.5 See also

- Tail call - Wikipedia, the free encyclopedia
- Optimizing Tail Call Recursion

- Trampolines in JavaScript
- Functional JavaScript – Tail Call Optimization and Trampolines | @taylodl's getting IT done
- java - Why does the JVM still not support tail-call optimization? - Stack Overflow
- Tail Recursion: Iteration in Haskell – Good Math, Bad Math
- Tail call explained
- Automatic Transformation of Iterative Loops into Recursive Methods
- The Road to Functional Programming in F# – From Imperative to Computation Expressions « Inviting Epiphany
- Optimizing List.map - Jane Street Tech Blogs

1.9 Fundamental Higher Order Functions

1.9.1 Overview

The functions map, filter and reduce (fold left) are ubiquitous in many programming languages and also the most used higher order functions.

They can be stricted evaluated like in Scheme and Javascript or lazy evaluated like in Python and Haskell.

1.9.2 Map

1. Overview

   The function map applies a function to each element of a sequence: list, vector, hash map or dictionary and trees.

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



         f :: a -> b
 a   ------------------------>>>  b
```

```
          map f :: [a] -> [b]
    [a] ------------------------>>> [b]
```

2. Map in Haskell

   The function map is lazy evaluated.

```
> let fun1 x = 3 * x + 1
> fun1 2
7
> map fun1 [1, 2, 3]
[4,7,10]
>

  -- The sequence 1 to 1000000 is not evaluated at all,
  --
> take 10 (map fun1 [1..1000000])
[4,7,10,13,16,19,22,25,28,31]

> take 10 (map fun1 [1..10000000000])
[4,7,10,13,16,19,22,25,28,31]
>
>
```

```
 --
 -- When applied to a function without a list, it creates
 -- another function that operates over lists because all
 -- Haskell functions are curried by default.
 --
 --         f :: (a -> b)
 --  map    :: (a -> b) -> [a] -> [b]
 --
 -- It can be seen as:
 --
 --  When map is applied to f, it will create the function fs
 --  that take list of type a and returns list of type b.
```

```
 --
 --  map    :: (a -> b) -> ([a] -> [b])
 --                |            |
 --                |            |------ fs :: [a] -> [b]
 --                |
 --                -------------------- f  :: a -> b
 --
> :t map
map :: (a -> b) -> [a] -> [b]

> let f x = 3 * x + 6
> :t f
f :: Num a => a -> a
>


> map f [1, 2, 3]
[9,12,15]
>

 -- Note: let is only needed in the REPL
 --
> let fs = map f

> :t fs
fs :: [Integer] -> [Integer]

> fs [1, 2, 3]
[9,12,15]
>
```

3. Map in Python

   In Python 3 map and filter are lazy evaluated, they return a generator.

```
>>> def fun1 (x):
    return 3*x + 6
...
>>> g = map(fun1, [1, 2, 3])
>>> g
```

```
<map object at 0xb6b4a76c>
>>> next (g)
9
>>> next (g)
12
>>> next (g)
15
>>> next (g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> g
<map object at 0xb6b4a76c>
>>>

 # Force the evaluation:
 #
 >>> list(map(fun1, [1, 2, 3]))
 [9, 12, 15]


 # Strict Version of map
 #
 # s_ stands for strict map.

def s_map (f, xs):
    return list(map(f, xs))

>>> s_map (fun1, [1, 2, 3])
[9, 12, 15]
>>>

 # Due to python doesn't have tail call optimization
 # recusion must be avoided, a higher number of iterations
 # can lead to a stack overflow.

def strict_map (f, xs):
    return [f (x) for x in xs]

>>> strict_map (fun1, [1, 2, 3])
```

```
[9, 12, 15]
>>> strict_map (fun1, range(5))
[6, 9, 12, 15, 18]
>>>

  # Lazy map implementation:
  # Note: the python native map is implemented in C, so
  # it is faster.
  #

def lazy_map (f, xs):
    for x in xs:
        yield x

>>> g = lazy_map (fun1, [1, 2, 3])
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> list(lazy_map (fun1, [1, 2, 3]))
[1, 2, 3]
>>>

 #
 # To the map function work like in Haskell and ML
 # it is need to be curried.
 #

curry2 = lambda f: lambda x: lambda y: f(x, y)

 # The function curry2 currify a function of two arguments
 #
>>> strict_map_c = curry2(strict_map)
```

```
>>> strict_map_c(fun1)
<function <lambda>.<locals>.<lambda>.<locals>.<lambda> at 0xb6afc0bc>

>>> strict_map_c(fun1)([1, 2, 3, 4])
[9, 12, 15, 18]
>>>

>>> fun1_xs = strict_map_c(fun1)
>>> fun1_xs ([1, 2, 3, 4])
[9, 12, 15, 18]
>>>
```

4. Map in Dynamic Typed Languages

   In dynamic typed languages like Python, Clojure and Scheme the function map can take multiple arguments. In typed languages the function takes only one argument.

   Map in Python:

```
>>> list(map (lambda a, b, c: 100 * a + 10 * b + c, [1, 2, 3, 4, 5], [8, 9, 10, 11
[183, 294, 407, 518, 630]
>>>
```

   Map in Scheme:

```
(map (lambda (a b c) (+ (* 100 a) (* 10 b) c))
      '(1 2 3 4 5)
      '(8 9 10 11 12)
      '(3 4 7 8 10))

$1 = (183 294 407 518 630)
```

   Map in Clojure:

```
;; f a b c = 100 * a + 10 * b + c
;;
;; 183 = f 1 8 3
;; 294 = f 2 9 4
;; ...
```

```
;; 630 = f 6 3 0
;;
user=> (map (fn [a b c] (+ (* 100 a) (* 10 b) c)) [1 2 3 4 5] [8 9 10 11 12] [3 4
(183 294 407 518 630)
user=>

;;
;; The clojure map is Polymorphic it can be applied to any collection
;; of seq abstraction like lists, vectors and hash maps.
;;

;; Map applied to a list
;;
user=> (map inc '(1 2 3 4 5 6))
(2 3 4 5 6 7)
user=>

;; Map applied to a vector
;;
user=> (map inc [1 2 3 4 5 6])
(2 3 4 5 6 7)
user=>

;; Map applied to a hash map
;;
user=> (map identity {:a 10 :b 20 :c "hello world"})
([:a 10] [:b 20] [:c "hello world"])
user=>

;; The function mapv is similar to map, but returns a vector:
;;
user=> (mapv identity {:a 10 :b 20 :c "hello world"})
[[:a 10] [:b 20] [:c "hello world"]]
user=>


;; Clojure also have destructuring
;;
user=> (map (fn [[[a b] c]] (+ (* 100 a ) (* 10 b) c))  [[[1 2] 3] [[3 4] 5] [[1 2
(123 345 124)
```

```
    user=>
```

1.9.3 Filter

Python

```
 ;;; Filter returns by default a
>>> g = filter (lambda x: x > 10, [1, 20, 3, 40, 4, 14, 8])
>>> g
<filter object at 0xb6b4a58c>
>>> [x for x in g]
[20, 40, 14]
>>> [x for x in g]
[]
>>> list(filter (lambda x: x > 10, [1, 20, 3, 40, 4, 14, 8]))
[20, 40, 14]
>>>

  # Stritct Version of filter function
  #
>>> _filter = lambda f, xs: list(filter(f, xs))
>>>
>>> _filter (lambda x: x > 10,  [1, 20, 3, 40, 4, 14, 8])
[20, 40, 14]
>>>

  # Filter implementation without recursion:
  #

def strict_filter (f, xs):
    result = []
    for x in xs:
        if f(x):
            result.append(x)
    return result

def lazy_filter (f, xs):
    for x in xs:
        if f(x):
            yield x
```

```
>>> strict_filter (lambda x: x > 10, [1, 20, 3, 40, 4, 14, 8])
[20, 40, 14]

>>> lazy_filter (lambda x: x > 10, [1, 20, 3, 40, 4, 14, 8])
<generator object lazy_filter at 0xb6b0f1bc>

>>> g = lazy_filter (lambda x: x > 10, [1, 20, 3, 40, 4, 14, 8])
>>> g
<generator object lazy_filter at 0xb6b0f194>
>>> next(g)
20
>>> next(g)
40
>>> next(g)
14
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>

>>> list(lazy_filter (lambda x: x > 10, [1, 20, 3, 40, 4, 14, 8]))
[20, 40, 14]
>>>
```

1.9.4 Reduce or Fold

1. Overview

   Fold Left

   The function fold left is tail recursive, whereas the function fold right is not. This functions is also known as reduce or inject (in Ruby). The function fold left is often called just fold like in F# or reduce (Python, Javascript, Clojure) and also Inject (Ruby).

   ```
   foldl ::  (State -> x -> State) -> State -> [x] -> State foldl
   (f ::  S -> x -> S) S [x]


   Sn = foldl f S0 [x0, x1, x2, x3 ... xn-1]
   ```

```
S1   = f S0 x0
S2   = f S1 x1     = f (f S0 x0) x1
S3   = f S2 x2     = f (f (f S0 x0) x1) x2
S4   = f S3 x3     = f (f (f (f S0 x0) x1) x2) x3
...
Sn-1 = f Sn-2 Xn-2 = ...
Sn   = f Sn-1 Xn-1 = f ...(f (f (f (f S0 x0) x1) x2) x3 ... xn

  ;;; -> Result
```

Fold Right

```
foldr ::  (x -> acc -> acc) -> acc -> [x] -> acc


S1   = f xn-1 S0
S2   = f xn-2 S1     = f xn-2 (f xn-1 S0)
S3   = f xn-3 S2     = f xn-3 (f xn-2 (f xn-1 S0))
S4   = f xn-4 S3     = f xn-4 (f xn-3 (f xn-2 (f xn-1 S0)))
....
Sn-1 = f x1   Sn-2   = ...
Sn   = f x0   Sn-1   = f x0 (f x1 ... (f xn-2 (f xn-1 S0)))
```

2. Haskell

   See also:

   - Fold (higher-order function) - Wikipedia, the free encyclopedia)
   - A tutorial on the universality and expressiveness of fold. GRAHAM HUTTON
   - Haskell unit 6: The higher-order fold functions | Antoni Diller

   Fold Left:

```
 foldl :: (acc -> x -> acc) -> acc -> [x] -> acc

                  |             |      |       |
                  |             |      |       |---> Returns the accumulated
                  |             |      |             value
                  |             |      |----- xs
                  |             |
```

```
                  |             |     Inital Value of accumulator
                  |             |---  acc0
                  |
                  |-----------------  f :: acc -> x -> acc
                                                  |
                                                  |--- Element of list
```

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

```
> :t foldl
foldl :: (a -> b -> a) -> a -> [b] -> a
>
> foldl (\acc x -> 10 * acc + x) 0 [1, 2, 3, 4, 5]
12345
>
```

It is equivalent to:

```
> let f acc x = 10 * acc + x
>
> (f 0 1)
1
> (f (f 0 1) 2)
12
> (f (f (f 0 1) 2) 3)
123
>
> (f (f (f (f 0 1) 2) 3) 4)
1234
> (f (f (f (f (f 0 1) 2) 3) 4) 5)
12345
>
```

Evaluation of Fold left:

```
> foldl (\acc x -> 10 * acc + x ) 0 [1, 2, 3, 4, 5]
12345
```

```
S0 = 0

f = \acc x -> 10 * acc + x

                  x  acc
S1 = f S0 x0 = f 0   1 = 10 * 0  + 1 = 1
S2 = f S1 x1 = f 10  2 = 10 * 1    + 2 = 12
S3 = f S2 x2 = f 12  3 = 10 * 12   + 3 = 123
S4 = f S3 x3 = f 123 4 = 10 * 123  + 4 = 1234
S5 = f S3 x3 = f 123 4 = 10 * 1234 + 5 = 12345
```

Fold right

```
 foldr :: (x -> acc -> acc) -> acc -> [x] -> acc

 foldr :: (a -> b -> b) -> b -> [a] -> b
 foldr f z []     = z
 foldr f z (x:xs) = f x (foldr f z xs)

> foldr (\x acc -> 10 * acc + x) 0 [1, 2, 3, 4, 5]
54321

> (f 0 5)
5
> (f (f 0 5) 4)
54
> (f (f (f 0 5) 4) 3)
543
> (f (f (f (f 0 5) 4) 3) 2)
5432
> (f (f (f (f (f 0 5) 4) 3) 2) 1)
54321
>

 --
 -- Derive fold_right from foldl (fold left)
 --
```

```
> let fold_right f acc xs = foldl (\x acc -> f acc x) acc (reverse xs)
>
> :t fold_right
fold_right :: (b -> a -> a) -> a -> [b] -> a
>
>
> fold_right (\x acc -> 10 * acc + x) 0 [1, 2, 3, 4, 5]
54321
>
```

Evaluation of Fold Right:

```
Example:

> foldr (\x acc -> 10 * acc + x ) 0 [1, 2, 3, 4, 5]
54321
>

f  = \x acc -> 10 * acc + x
S0 = 0
n = 5
                        x acc
S1   = f x4 S0     = f 5  0    = 10 * 0    + 5 = 5
S2   = f x3 S1     = f 4  5    = 10 * 5    + 4 = 54
S3   = f x2 S2     = f 3  54   = 10 * 54   + 3 = 543
S4   = f x1 S3     = f 2  543  = 10 * 543  + 2 = 5432
S5   = f x0 S4     = f 1  5432 = 10 * 5432 + 1 = 54321
```

3. Python

   In Python 3 the function reduce is not default anymore, however it can be found in the native library functools, that has a lot of built-in functions for functional programming. The function reduce is equivalent to Haskell function foldl (fold left) which is tail recursive.

```
reduce(function, sequence[, initial]) -> value

reduce :: (acc -> x -> acc) -> [x] ?acc0  -> acc

>>> from functools import reduce
```

```
>>>

>>> reduce (lambda acc, x: 10 *  acc + x , [1, 2, 3, 4, 5], 0)
12345
>>>

>>> f = lambda acc, x: 10 *  acc + x
>>>
>>> f(0, 1)
1
>>> f( f(0, 1), 2)
12
>>> f( f( f(0, 1), 2), 3)
123
>>> f( f( f( f(0, 1), 2), 3), 4)
1234
>>> f( f( f( f( f(0, 1), 2), 3), 4), 5)
12345
>>>

def my_reduce (f, xs, acc0=None):
    "Non recursive implementation of reduce (fold_left)
     with optional initial accumulator value.
    "

    if acc0 is None:
        acc = xs[0]
        xss = xs[1:]
    else:
        acc = acc0
        xss = xs

    for x in xss:
        acc = f (acc, x)

    return acc


>>>
>>> my_reduce(lambda acc, x: 10 * acc + x, [1, 2, 3, 4, 5], 0)
```

```
12345
>>> my_reduce(lambda acc, x: 10 * acc + x, [1, 2, 3, 4, 5])
12345
>>> my_reduce(lambda acc, x:  acc + x, [1, 2, 3, 4, 5], 0)
15
>>> my_reduce(lambda acc, x:  acc * x, [1, 2, 3, 4, 5], 1)
120
>>>

 #
 # Implementation without recursion.
 #

def fold_left (f_acc_x_to_acc, acc0, xs):
    "Haskell-like fold left function

    fold_left :: (acc -> x -> acc) -> acc -> [x]
    "
    acc = acc0

    for x in xs:
        acc = f_acc_x_to_acc (acc, x)

    return acc

>>> fold_left (lambda acc, x: 10 * acc + x, 0, [1, 2, 3, 4, 5])
12345
>>>


def fold_right (f, acc0, xs):
    return fold_left ((lambda acc, x: f(x, acc)), acc0, reversed(xs))

>>> fold_right (lambda x, acc: 10 * acc + x, 0, [1, 2, 3, 4, 5])
54321
>>>

def fold_right2 (f, acc0, xs):
    acc = acc0
```

```
    for x in reversed(xs):
        acc = f(x, acc)

    return acc

>>> fold_right2 (lambda x, acc: 10 * acc + x, 0, [1, 2, 3, 4, 5])
54321
>>>
```

Usefulness of Fold

Many functions and recursive algorithms can be implemented using the fold function, including map, filter, sum, product and others.

It is based in the paper:

- A tutorial on the universality and expressiveness of fold. GRAHAM HUTTON

In the paper was used fold right, here was used fold left.

```
def fold_left (f_acc_x_to_acc, acc0, xs):
    "Haskell-like fold left function

    fold_left :: (acc -> x -> acc) -> acc -> [x]
    "
    acc = acc0

    for x in xs:
        acc = f_acc_x_to_acc (acc, x)

    return acc


    ;;; Function fold in curried form

curry3 = lambda f: lambda x: lambda y: lambda z: f(x, y, z)

fold = curry3(fold_left)

>>> summation = fold(lambda acc, x: acc + x)(0)
```

```
>>>
>>> summation([1, 2, 3, 4, 5, 6])
21
>>>

>>> product = fold(lambda acc, x: acc * x)(1)
>>> product([1, 2, 3, 4, 5])
120
>>>

>>> f_or = fold(lambda acc, x: acc or x)(False)
>>> f_or([False, False, False])
False
>>>
>>> f_or([False, False, True])
True
>>>

>>> f_and = fold(lambda acc, x: acc and x)(True)
>>>
>>> f_and([False, True, True])
False
>>> f_and([True, True, True])
True
>>>

>>> length = fold(lambda acc, x: acc + 1)(0)
>>> length ([1, 2, 3, 4, 5])
5

>>> _map = lambda f, xs: fold(lambda acc, x: acc + [f(x)] )([])(xs)
>>> _map (lambda x: x * 3, [1, 2, 3, 4])
[3, 6, 9, 12]
>>>

>>> _filter = lambda p, xs: fold(lambda acc, x: (acc + [x]) if p(x) else  acc )([]
>>>
>>> _filter(lambda x: x > 10, [10, 3, 8, 2, 20, 30])
[20, 30]
>>>
```

```
 #
 # Function composition
 #
 #  (f3 (f2 (f1 (f0 x))))
 #
 #  (f3 . f2 . f1 . f0) x
 #
 #  or using, forward composition:
 #
 #  (f0 >> f2 >> f1 >> f0) x
 #

>>> f1 = lambda x: 3 * x
>>> f2 = lambda x: 5 + x
>>> f3 = lambda x: 2 ** x


>>> _fcomp = lambda functions: lambda x: fold(lambda acc, f: f(acc)) (x) (functior

>>> _fcomp([f1, f2, f3])(3)
16384

>>> (f3 (f2 (f1 (3))))
16384
>>>
```

4. Clojure

   The function reduce is similar to Haskell fold left and Python reduce. This function is Polymorphic. It works on any collection of seq abstraction: lists, vectors and hash maps.

   Signature:

```
(reduce f coll)      -> reduce :: (f :: acc -> x -> acc) -> [x]

Or

(reduce f val coll)  -> reduce :: (f :: acc -> x -> acc) -> acc -> [x]
```

```
f :: acc -> x -> acc

;; Applying fold/reduce to a list
;;
;;
user=> (reduce (fn [acc x] (+ (* 10 acc) x)) 0 '(1 2 3 4 5))
12345


;; Applying fold/reduce to a vector
;;
user=> (reduce (fn [acc x] (+ (* 10 acc) x))  0 [1 2 3 4 5])
12345
user=>

user=> (reduce (fn [acc x] (+ (* 10 acc) x)) 0 [])
0

;; Applyind fold/reduce to a Hash map
;;
user=> (reduce (fn [acc x] (cons x  acc )) '()  { :a 10 :b 20 :c 30 })
([:c 30] [:b 20] [:a 10])
user=>

;; Without Initial value of accumulator it will fail on a empty list.
;;
user=> (reduce (fn [acc x] (+ (* 10 acc) x)) [1 2 3 4 5])
12345

user=> (reduce (fn [acc x] (+ (* 10 acc) x)) [])
ArityException Wrong number of args (0) passed to: user/eval44/fn--45  clojure.lan
user=>

;; Implementing fold right
;;
(defn foldr
   ([f xs]       (reduce (fn [acc x] (f x acc))     (reverse xs)))
   ([f acc xs]   (reduce (fn [acc x] (f x acc)) acc (reverse xs)))
  )
```

```
user=> (foldr (fn [x acc] (+ (* 10 acc) x)) 0 [1 2 3 4 5])
54321


;; Clojure has destructuring
;;
user=> (reduce (fn [acc [a b]] (conj acc (+ (* 10 a) b) )) '[] [[1 2] [3 4] [5 8]]
[12 34 58]
user=>

;; Implementing map with fold left (reduce)
;;
user=> (defn map2 [f xs]
          (reverse (reduce (fn [acc x] (cons (f x) acc))
                           ()
                           xs)))
#'user/map2
user=>
user=> (map2 inc '(1 2 3 3 4 5))
(2 3 4 4 5 6)
user=>

;; Implementing map with fold right
;;
;;

(defn map2 [f xs]
   (foldr (fn [x acc] (cons (f x) acc))
          ()
          xs
   ))

user=> (map2 inc '(1 2 3 4 5 6))
(2 3 4 5 6 7)
user=>
```

5. Fsharp

```
// Fold left for Lists
```

```
//
//

// List.fold (acc -> 'x -> 'acc) -> acc -> 'x list -> 'acc
//
- List.fold ;;
val it : (('a -> 'b -> 'a) -> 'a -> 'b list -> 'a)

- List.fold (fun acc x -> 10 * acc + x) 0 [1; 2; 3; 4; 5]  ;;
val it : int = 12345
>

// Array.fold
//
//
- Array.fold ;;
val it : (('a -> 'b -> 'a) -> 'a -> 'b [] -> 'a)
>

- Array.fold (fun acc x -> 10 * acc + x) 0 [| 1; 2; 3; 4; 5 |]  ;;
val it : int = 12345
>

// Fold left for Arrays
```

Example: Implementing Higher Order Functions and recursive functions with fold.

```
// Implementing fold_left for lists
//
let rec fold_left f xs acc =
    match xs with
    | []      ->   acc
    | hd::tl  ->   fold_left f tl (f acc hd)
;;

val fold_left : f:('a -> 'b -> 'a) -> xs:'b list -> acc:'a -> 'a

- fold_left (fun acc x -> 10 * acc + x) [1; 2; 3; 4 ; 5] 0 ;;
val it : int = 12345
```

```
>

let length xs = fold_left (fun acc x -> acc + 1) xs 0

> length ["a"; "b"; "c"; "d" ] ;;
val it : int = 4
> length [ ] ;;
val it : int = 0
>

- let sum xs  = fold_left (fun acc x -> acc + x) xs 0 ;;

> sum [1; 2; 3; 4; 5; 6] ;;
val it : int = 21
>

> let product xs = fold_left (fun acc x -> acc * x) xs 1 ;;

val product : xs:int list -> int

> product [1; 2; 3; 4; 5; 6] ;;
val it : int = 720
>

- let reverse xs = fold_left (fun acc x -> x :: acc) xs []
- ;;

val reverse : xs:'a list -> 'a list

> reverse [1; 2; 3; 4; 5 ] ;;
val it : int list = [5; 4; 3; 2; 1]
>

let fold_right f xs acc =
  fold_left (fun acc x -> f x acc) (reverse xs) acc
;;

val fold_right : f:('a -> 'b -> 'b) -> xs:'a list -> acc:'b -> 'b

- fold_right (fun x acc -> 10 * acc + x) [1; 2; 3; 4; 5] 0 ;;
```

```
val it : int = 54321
>

// Reverse map
//
- let rev_map f xs = fold_left (fun acc x -> (f x)::acc) xs [] ;;

val rev_map : f:('a -> 'b) -> xs:'a list -> 'b list

- rev_map (fun x -> x * 2) [1; 2; 3; 4; 5; 6] ;;
val it : int list = [12; 10; 8; 6; 4; 2]
>

- let map f xs = reverse ( fold_left (fun acc x -> (f x)::acc) xs [] ) ;;

val map : f:('a -> 'b) -> xs:'a list -> 'b list

- map (fun x -> x * 2) [1; 2; 3; 4; 5; 6] ;;
val it : int list = [2; 4; 6; 8; 10; 12]
>

// Or
//
let rev_fold_left f  xs acc = reverse (fold_left f xs acc) ;;

val rev_fold_left :
  f:('a list -> 'b -> 'a list) -> xs:'b list -> acc:'a list -> 'a list

// Map with fold left and reverse
//
//
> let map f xs = rev_fold_left (fun acc x -> (f x)::acc) xs [] ;;

val map : f:('a -> 'b) -> xs:'b list -> 'b list

- map (fun x -> x * 2) [1; 2; 3; 4; 5; 6] ;;
val it : int list = [2; 4; 6; 8; 10; 12]
>

// Map with fold right
```

```
//
> let map f xs = fold_right (fun x acc -> (f x)::acc) xs [] ;;

val map : f:('a -> 'b) -> xs:'a list -> 'b list

> map (fun x -> x * 2) [1; 2; 3; 4; 5; 6] ;;
val it : int list = [2; 4; 6; 8; 10; 12]
>

// Filter with fold left and reverse
//
let filter f xs = rev_fold_left (fun acc x -> if (f x) then (x::acc) else acc) xs

val filter : f:('a -> bool) -> xs:'a list -> 'a list

- filter (fun x -> x % 2 = 0) [1; 2; 3; 4; 5; 6; 7; 8; 9 ] ;;
val it : int list = [2; 4; 6; 8]
>

// Filter with fold right
//
let filter f xs =
  fold_right (fun x acc ->  if (f x)
                            then x::acc
                            else acc
             )
             xs
             []
             ;;

-  filter (fun x -> x % 2 = 0) [1; 2; 3; 4; 5; 6; 7; 8; 9 ] ;;
val it : int list = [2; 4; 6; 8]
>



let take n xs =
  let _, result =
    fold_left (fun acc x -> let (c, xss) = acc in
                            if c = 0
```

```
                            then  (0, xss)
                            else  (c - 1, x::xss))
              xs
              (n, [])

  in reverse result

            ;;

- take ;;
val it : (int -> 'a list -> 'a list) = <fun:clo@202-3>
>


> take 3 [1; 2; 3 ; 4; 5; 6; 7; 8] ;;
val it : int list = [1; 2; 3]
>

- take 18 [1; 2; 3 ; 4; 5; 6; 7; 8] ;;
val it : int list = [1; 2; 3; 4; 5; 6; 7; 8]
>

// drop with fold left
//
let drop n xs =
  let _, result =
    fold_left (fun acc x -> let (c, xss) = acc in
                            if c = 0
                            then  (0, x::xss)
                            else  (c - 1, xss))
              xs
              (n, [])

  in reverse result

            ;;

val drop : n:int -> xs:'a list -> 'a list
```

```
- drop 3 [1; 2; 3 ; 4; 5; 6; 7; 8] ;;
val it : int list = [4; 5; 6; 7; 8]
>
- drop 13 [1; 2; 3 ; 4; 5; 6; 7; 8] ;;
val it : int list = []
>

let take_while f xs =
  fold_right   (fun x acc ->  if (f x)
                              then  x::acc
                              else  acc )

               xs
               []
;;


let take_while f xs =
  fold_right   (fun x acc ->  if (f x)
                              then  x::acc
                              else  match acc with
                                    |  []     -> []
                                    |  _::tl -> tl
               )

               xs
               []
;;
val take_while : f:('a -> bool) -> xs:'a list -> 'a list

> take_while (fun x -> x < 10) [2; 8 ; 9 ; 26 ; 7; 10; 53] ;;
val it : int list = [2; 8; 9]
>



let find f xs =
  fold_left (fun acc x -> if (f x)
                          then Some x
                          else None
```

```
            )
            xs
            None
;;

val find : f:('a -> bool) -> xs:'a list -> 'a option

- find (fun x -> x * x > 40) [1; 2; 6; 5; 4; 8; 10; 20 ; 9 ] ;;
val it : int option = Some 9
>

- find (fun x -> x * x > 400) [1; 2; 6; 5; 4; 8; 10; 20 ; 9 ] ;;
val it : int option = None
>


// Map with side-effect
//
let for_each f xs =
  fold_left (fun acc x -> f x)
            xs
            ()
            ;;

val for_each : f:('a -> unit) -> xs:'a list -> unit

> for_each (fun x -> printfn "x = %d" x) [2; 3; 4; 5; 6] ;;
x = 2
x = 3
x = 4
x = 5
x = 6
val it : unit = ()
>

// Filter map - fusion / optimization
//
// (Eliminate intermediate data structure )
//
```

```
let filter_map f_filter f_map xs =
  fold_right (fun x acc ->  if (f_filter x)
                            then (f_map x)::acc
                            else acc
             )
             xs
             []
;;

val filter_map :
  f_filter:('a -> bool) -> f_map:('a -> 'b) -> xs:'a list -> 'b list

- filter_map (fun x -> x % 2 = 0) (fun x -> x + 3) [1; 5; 2; 6; 8; 7]
- ;;
val it : int list = [5; 9; 11]
>


 // Without optimization
- map (fun x -> x + 3) (filter (fun x -> x % 2 = 0) [1; 5; 2; 6; 8; 7]) ;;
val it : int list = [5; 9; 11]
>
-
```

1.9.5 For Each, Impure map

For each is an impure higher order function which performs side-effect on each element of a list, array or sequence. Unlike map this function neither have a standard name or return anything.

Scheme

```
> (for-each (lambda (i) (display i) (newline))  '(1 2 3 4 5 6))
1
2
3
4
5
6
>
```

```
> (for-each
   (lambda (a b c)
      (display a) (display b) (display c)
      (newline)
    )
   '(a b c d e f)
   '(1 2 3 4 5 6)
   '("x" "y" "z" "w" "h" "k"))
a1x
b2y
c3z
d4w
e5h
f6k
```

Common Lisp

```
> (mapc #'print '(1 2 3 3 4))

1
2
3
3
4
```

Scala

```
scala> var xs = List(1.0, 2.0, 3.0, 4.0, 5.0, 6.0)
xs: List[Double] = List(1.0, 2.0, 3.0, 4.0, 5.0, 6.0)

scala> xs.foreach(println)
1.0
2.0
3.0
4.0
5.0
6.0

scala> xs.foreach(x => println( "x = %.3f".format(x)))
x = 1,000
```

```
x = 2,000
x = 3,000
x = 4,000
x = 5,000
x = 6,000
```

Ocaml

```
> List.iter ;;
- : ('a -> unit) -> 'a list -> unit = <fun>

> List.iter (fun x -> print_int x ; print_string "\n") [1 ; 2; 3; 4; 5] ;;
1
2
3
4
5
- : unit = ()
```

F#

```
> List.iter ;;
val it : (('a -> unit) -> 'a list -> unit) = <fun:clo@1>

> List.iter (fun x -> printfn "x = %d" x) [1; 2; 3; 4; 5] ;;
x = 1
x = 2
x = 3
x = 4
x = 5
val it : unit = ()
>

> List.iter2 ;;
val it : (('a -> 'b -> unit) -> 'a list -> 'b list -> unit) = <fun:clo@1>
>

- List.iter2 (fun a b -> printfn "a = %d b = %d" a b) [2; 3; 4; 5] [1; 2; 3; 4] - ;;
a = 2 b = 1
a = 3 b = 2
```

```
a = 4 b = 3
a = 5 b = 4
val it : unit = ()
>

- Array.iter ;;
val it : (('a -> unit) -> 'a [] -> unit) = <fun:it@6-4>
>
-

- Array.iter (fun x -> printfn "x = %d" x) [| 1; 2; 3; 4 |] ;;
x = 1
x = 2
x = 3
x = 4
val it : unit = ()
>
```

Python

This function is not in Python standard library however, it can be defined as this.

```
def for_each(f, * xss):
    for xs in zip(* xss):
        f(*xs)

>>> for_each (print, [1, 2, 4, 5, 6])
1
2
4
5
6

>>> for_each (lambda a, b: print (a, b), [1, 2, 3, 4, 5, 6], ["a", "b", "c", "d", "e",
1 a
2 b
3 c
4 d
5 e
6 f
```

Clojure

```
user=> (defn f [a b] (println (format "a = %s , b = %s" a b)))
#'user/f


(defn for-each [f & xss]
   (doseq [args (apply map vector xss)]  (apply f args)))


user=> (for-each println [1 2 3 4])
1
2
3
4
nil


user=> (for-each f [1 2 3 4] [3 4 5 6])
a = 1 , b = 3
a = 2 , b = 4
a = 3 , b = 5
a = 4 , b = 6
nil
user=>
```

1.9.6 Apply

The function apply applies a function to a list or array of arguments. It is common in dynamic languages like Lisp, Scheme, Clojure, Javascript and Python.

See also: Apply Higher Oder Function - Wikipedia

Example:

Scheme

```
> (define (f1 x y z) (+ (* 2 x) (* 3 y) z))

> (apply f1 '(1 2 3))
$1 = 11
```

```
> (apply f1 1 2 '(3))
$2 = 11

> (apply f1 1 '(2 3))
$3 = 11

> (apply + '(1 2 3 4 5))
$1 = 15

;;; The function apply can be used to transform a function of
;;; multiple arguments into a function of a single argument that
;;; takes a list of arguments
;;;

(define (f-apply f)
   (lambda (xs) (apply f xs)))

> (map (f-apply f1) '((1 2 3) (3 4 5) (6 7 8)) )
$2 = (11 23 41)

;;  It can be also used to create a function that  maps a function
;;  of multiple arguments over a list of arguments.
;;
(define (map-apply f xss)
   (map (lambda (xs) (apply f xs)) xss))

> (map-apply f1  '((1 2 3) (3 4 5) (6 7 8)))
$1 = (11 23 41)
```

Clojure

```
user=> (defn f1 [x y z] (+ (* 2 x) (* 3 y) z))
#'user/f1
user=>

;; The higher order function apply is polymorphic.
;;
user=> (apply f1 '(1 2 3))
11
user=> (apply f1 [1 2 3])
```

```
11
user=> (apply f1 1 [2 3])
11
user=> (apply f1 1 2 [ 3])
11
user=> (apply f1 1 2 3 [ ])
11

user=> (apply + [1 2 3 4 5])
15

user=> (defn map-apply [f xss]
              (map (fn [xs] (apply f xs)) xss))
#'user/map-apply
user=>


user=> (map-apply f1 [[1 2 3] [3 4 5] [6 7 8]])
(11 23 41)
user=>
```

Python

```
# In Python the * asterisk notation is used to expand
# a list into function arguments.
#
>>> f1 = lambda x, y, z: 2 * x + 3 * y + z
>>>
>>> f1(1, 2, 3)
11
>>> f1(*[1, 2, 3])
11
>>>

>>> def apply (f, xs): return f(*xs)
...
>>>

>>> apply (f1, [1, 2, 3])
11
```

```
#
# The function apply can also be defined  as:
#

def apply2 (f, * args):
    return f(*(tuple(args[:-1]) + tuple(args[-1])))

>>> apply2 (f1, [1, 2, 3])
11
>>> apply2 (f1, 1, [2, 3])
11
>>> apply2 (f1, 1, 2, [3])
11
>>> apply2 (f1, 1, 2, 3, [])
11
>>>

>>> f_apply = lambda f: lambda xs: f(*xs)
>>>

>>> list(map (f_apply(f1), [[1, 2, 3], [3, 4, 5], [6, 7, 8]]))
[11, 23, 41]
>>>

def map_apply (f, xss):
    return map(lambda xs: f(*xs), xss)

>>> map_apply(f1, [[1, 2, 3], [3, 4, 5], [6, 7, 8]])
<map object at 0xb6daaaac>
>>>
>>> list(map_apply(f1, [[1, 2, 3], [3, 4, 5], [6, 7, 8]]))
[11, 23, 41]
>>>
```

1.10 Special Functions

1.10.1 Identity Function

The identity function is a polymorphic function which returns the value of its argument.

```
 -- This is a built-in Haskell function
> :t id
id :: a -> a
>

> id 10
10
> id (Just 100)
Just 100
>
> let identity x = x
> identity 200
200
>
```

1.10.2 Constant Function

The constant function returns the value of the first argument (the constant) regardless of the value of second argument.

```
> :t const
const :: a -> b -> a

> let const10 = const 10
> const10 100
10
> const10 300
10
> map const10 ["a", "b", "c", "d"]
[10,10,10,10]
>
```

Python Implementation:

```
>>> constant = lambda const: lambda x: const
>>>

>>> const10 = constant(10)
>>>
>>> const10(100)
10
>>> const10("hello world")
10
>>>
```

1.10.3 List Constructor (Cons)

The cons operator is widely used in list recursive functions and higher order functions that operates on lists.

Scheme:

```
> (cons 1 (cons 2 (cons 3 (cons 4 '()))))
(1 2 3 4)
```

Haskell:

```
 -- Cons Operator
 --
> :t (:)
(:) :: a -> [a] -> [a]

> 1:[]
[1]

> 1:2:3:4:[]
[1,2,3,4]

> let cons = (:)

> (cons 1 (cons 2 (cons 3 (cons 4 []))))
[1,2,3,4]
```

Ocaml and F#

```
> 1::[] ;;
```

```
val it : int list = [1]
>
- 1::2::3::[] ;;
val it : int list = [1; 2; 3]
>

- let cons x xs = x::xs ;;

val cons : x:'a -> xs:'a list -> 'a list

cons 1 (cons 2 (cons 3 (cons 4 []))) ;;

> cons 1 (cons 2 (cons 3 (cons 4 []))) ;;
val it : int list = [1; 2; 3; 4]
>
```

1.10.4 Zip

1. Overview

   The function zip and its variants combine two or more sequence into one sequence.

   See also: Convolution (computer science)

2. Zip in Haskell

   ```
   > :t zip
   zip :: [a] -> [b] -> [(a, b)]

   > zip [1, 2, 3, 4] ["a", "b", "c"]
   [(1,"a"),(2,"b"),(3,"c")]
   >

   > zip [1, 2, 3, 4] ["a", "b", "c"]
   [(1,"a"),(2,"b"),(3,"c")]
   >

    -- Zip a list and a infinite list

   > zip ["a", "b", "c"] [1 ..]
   [("a",1),("b",2),("c",3)]
   ```

```
>
>

> zip ["a", "b", "c", "d"] [1, 2, 3]
[("a",1),("b",2),("c",3)]
>


> :t zip3
zip3 :: [a] -> [b] -> [c] -> [(a, b, c)]
>

> zip3 ["a", "b", "c", "d"] [1, 2, 3, 4, 5, 6] [Just 10, Just 100, Nothing]
[("a",1,Just 10),("b",2,Just 100),("c",3,Nothing)]
>

 -- There is more zip functions in the module Data.List
 --
> import Data.List (zip4, zip5, zip6)
>

> :t zip4
zip4 :: [a] -> [b] -> [c] -> [d] -> [(a, b, c, d)]
>
> :t zip5
zip5 :: [a] -> [b] -> [c] -> [d] -> [e] -> [(a, b, c, d, e)]
>
```

3. Zip in Python

   The Python zip function is inspired by Haskell. The Python zip functions can take a variable number of arguments.

   Python 2: In python2 this function is evaluated eagerly.

```
>>> zip([1, 2, 3], ["a", "b", "c", "d", "e"])
[(1, 'a'), (2, 'b'), (3, 'c')]
>>>

>>> zip([1, 2, 3], ["a", "b", "c", "d", "e"], ["x", "y", "z"])
[(1, 'a', 'x'), (2, 'b', 'y'), (3, 'c', 'z')]
```

```
>>>

>>> zip([1, 2, 3], ["a", "b", "c", "d", "e"], ["x", "y", "z"], range(1, 20))
[(1, 'a', 'x', 1), (2, 'b', 'y', 2), (3, 'c', 'z', 3)]
>>>

>>> for x, y in zip([1, 2, 3, 4, 5], ["a", "b", "c", "d", "e"]):
...   print "x = ", x, "y = ", y
...
x =  1 y =  a
x =  2 y =  b
x =  3 y =  c
x =  4 y =  d
x =  5 y =  e
```

Python 3: In python3 this function returns a generator. It is evaluated lazily.

```
>>> zip([1, 2, 3, 4, 5], ["a", "b", "c", "d", "e"])
<zip object at 0xb6d01ecc>
>>>

>>> list(zip([1, 2, 3, 4, 5], ["a", "b", "c", "d", "e"]))
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
>>>

>>> g = zip([1, 2, 3, 4, 5], ["a", "b", "c", "d", "e"])
>>> g
<zip object at 0xb6747e8c>
>>> next(g)
(1, 'a')
>>> next(g)
(2, 'b')
>>> next(g)
(3, 'c')
>>> next(g)
(4, 'd')
>>> next(g)
(5, 'e')
>>> next(g)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>

>>> g = zip([1, 2, 3, 4, 5], ["a", "b", "c", "d", "e"], range(1, 1000000000))
>>> g
<zip object at 0xb6747f2c>
>>> list(g)
[(1, 'a', 1), (2, 'b', 2), (3, 'c', 3), (4, 'd', 4), (5, 'e', 5)]
>>> list(g)
[]
>>>

>>> for x, y in zip([1, 2, 3, 4, 5], ["a", "b", "c", "d", "e"]):
...    print ("x = ", x, "y = ", y)
...
x =  1 y =  a
x =  2 y =  b
x =  3 y =  c
x =  4 y =  d
x =  5 y =  e
```

4. Zip in Scheme

   It can be defined as:

```
(define (zip . lists)
   (apply map  vector lists))

> (zip '(1 2 3 4) '("a" "b" "c" "d"))
(#(1 "a") #(2 "b") #(3 "c") #(4 "d"))
>

;; Or

(define (zip . lists)
   (apply map  list lists))

> (zip '(1 2 3 4) '("a" "b" "c" "d"))
```

```
((1 "a") (2 "b") (3 "c") (4 "d"))
>

> (zip '(1 2 3 4) '("a" "b" "c" "d") '(89 199 100 43))
((1 "a" 89) (2 "b" 199) (3 "c" 100) (4 "d" 43))
>
```

5. Zip in Clojure

```
;; Zip returning list
;;
(defn zip [& seqs]
  (apply map list seqs))

user=> (zip '(1 2 3 4 5) '(x y z w k m n))
((1 x) (2 y) (3 z) (4 w) (5 k))

user=> (zip '(1 2 3 4 5) '(x y z w k m n) (range 10 1000000))
((1 x 10) (2 y 11) (3 z 12) (4 w 13) (5 k 14))

user=> (zip '[1 2 3 4 5] '(x y z w k m n) (range 10 1000000))
((1 x 10) (2 y 11) (3 z 12) (4 w 13) (5 k 14))

user=> (zip '[1 2 3 4 5] '(x y z w k m n) {:key_x "hello" :key_y "world" :key_z "c
((1 x [:key_x "hello"]) (2 y [:key_y "world"]) (3 z [:key_z "clojure"]))
user=>


;; Zip returning vector
;;
(defn zipv [& seqs]
  (apply mapv vector seqs))

user=> (zipv '(1 2 3 4 5) '(x y z w k m n))
[[1 x] [2 y] [3 z] [4 w] [5 k]]
user=>

user=>  (zipv '[1 2 3 4 5] '(x y z w k m n) (range 10 1000000))
[[1 x 10] [2 y 11] [3 z 12] [4 w 13] [5 k 14]]
user=>
```

1.11 Function Composition

1.11.1 Overview

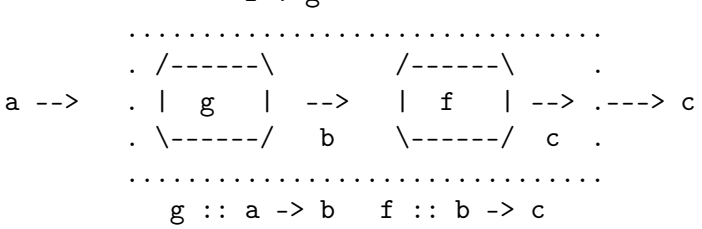
Function composition promotes shorter code, code reuse and higher modularity by creating new functions from previous defined ones. They also allow optimization of functional code when there is many maps. Only pure functions can be composed, function composition works like math functions, the output of one function is the input of another function. Haskell, ML, Ocaml and F# has features that makes easier to use function composition, like a lightweight syntax, currying, partially applied functions, static typing and composition operators that are built in to the language. In Haskell the operator (.) dot is used for composing functions.

See also: Function composition (computer science)

1.11.2 Function Composition in Haskell

```
(.) :: (b -> c) -> (a -> b) -> a -> c

Given:

    f :: b -> c
    g :: a -> b

(f . g ) x = f (g x)

    h = f . g
    h :: a -> c
```

Function Composition Block Diagram




```
    (.) :: (b -> c) -> (a -> b) -> a -> c
```

Composition Law

```
id . f = f                  Left  identity law
f . id = f                  Right identity law
(f . g) . h = f . (g . h)   Associativity


Constant Function Composition
f       . const a = const (f a)
const a . f       = const a

dentity function            -->  id x = x
const - Constant Function   --> const a b =  a
```

Simplifying Code with function composition:

```
    h( f ( g( x)))  ==>  (h . f . g ) x   OR  h . f . g  $ x
OR
    h $ f $ g x     ==>   h . f . g $ x

                                Point Free Style
composed x = h . f . g $ x ==>  composed = h . f . g
```

Function Composition with Map

```
    (map g (map f xs) == (map g . map f) xs = (map g . f) xs

OR
    map g . map f  == map (g . f)

Generalizing

    map f1 (map f2 (map f3 (map f4 xs)))
    = (map f1)
    =  map (f1 . f2 . f3 . f4)  xs
    =  f xs

Where f = map $ f1 . f2 . f3 . f4

Example:

    > map  (+3) [1, 2, 3, 4]
```

```
    [4,5,6,7]
    > map  (*2) [4, 5, 6, 7]
    [8,10,12,14]
    >
    > map  (*2) (map (+3)  [1, 2, 3, 4])
    [8,10,12,14]
    >
    > map  (*2) . map (+3) $  [1, 2, 3, 4]
    [8,10,12,14]
    >

    > map ((*2) . (+3)) [1, 2, 3, 4]
    [8,10,12,14]

    > let f = map $ (*2) . (+3)
    > f [1, 2, 3, 4]
    [8,10,12,14]
```

```
h :: c -> [a]
f :: a -> b

map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]


map     f (h c) = map    f . h $ c
filter  f (h c) = filter f . h $ c
```

Inverting Predicate Functions

```
inverted_predicate == not . predicate
```

```
> not True
False
> not False
True
>

> (>5) 10
True
> (>5) 3
```

```
False

> not . (>5) $ 10
False
> not . (>5) $ 3
True
>

> let f = not . (>5)
> f 10
False
> f 5
True

> import Data.List
>
> filter ( isPrefixOf "a" ) ["a","ab","cd","abcd","xyz"]
["a","ab","abcd"]
>
> filter ( not . isPrefixOf "a" ) ["a","ab","cd","abcd","xyz"]
["cd","xyz"]
>
```

Example:

```
> let f = (+4)
> let g = (*3)
>
>
> f (g 6) -- (+4) ((*3) 6) = (+4) 18 = 22
22
>
> (f . g) 6
22
>
> (.) f g 6
22
>
> let h = f . g
>
```

```
> h 6
22
>

> id 10
10
> id 3
3
>
> id Nothing
Nothing
> id 'a'
'a'
> id (Just 10)
Just 10
>


> (f . id) 10
14
> (id . f) 10
14
>

> const 10 20
10
> const 10 3
10
>

> (f . (const 10)) 4
14
> (f . (const 10)) 3
14
> const 10 . f $ 7
10
> const 10 . f $ 3
10
>
```

```
{- Avoiding Parenthesis with composition -}
> let g x = x * 2
> let f x = x + 10
> let h x = x - 5
>
> h (f (g 3))
11
> h $ f $ g 3
11
>
> (h . f . g ) 3
11
> h . f . g $ 3
11
>

{- Function Composition with curried functions -}

> let f1 x y = 10*x + 4*y
> let f2 a b c = 4*a -3*b + 2*c
> let f3 x = 3*x

> (f1 3 ( f3 5))
90
>
> f1 3 $ f3 5
90
>
> f1 3 . f3 $ 5
90
>
> let f = f1 3 . f3
>
> f 5
90
> f 8
126
>
```

```
> (f1 4 (f2 5 6 (f3 5)))
168
>
> f1 4 $ f2 5 6 $ f3 5
168
>
> f1 4 . f2 5 6 . f3 $ 5
168
>
> let g = f1 4 . f2 5 6 . f3 {- You can also create new functions -}
> :t g
g :: Integer -> Integer
> g 5
168
> g 10
288
>

{- Function Composition with Map and Filter -}

> import Data.Char

> :t ord
ord :: Char -> Int

> :t ordStr
ordStr :: [Char] -> [Int]
>

> ordStr "curry"
[99,117,114,114,121]
>
> let r x= x + 30
>
> map r (ordStr "curry")
[129,147,144,144,151]
>
> map r $ ordStr "curry"
[129,147,144,144,151]
>
```

```
> map r . ordStr $ "curry"
[129,147,144,144,151]
>
> sum . map r . ordStr $ "curry"
715
>

> let s =  map r . ordStr
> s "curry"
[129,147,144,144,151]
> s "haskell"
[134,127,145,137,131,138,138]
>

let sum_ord = sum . map r . ordStr

> sum_s "curry"
715
> sum_s "haskell"
950
>
> sum_ord "curry"
715
> sum_ord "haskell"
950
>


> map ord (map toUpper "haskell")
[72,65,83,75,69,76,76]
>
> map ord . map toUpper $ "haskell"
[72,65,83,75,69,76,76]
>

> map (flip (-) 10) . map ord . map toUpper $ "haskell"
[62,55,73,65,59,66,66]
>

> map chr . map (flip (-) 10) . map ord . map toUpper $ "haskell"
```

```
">7IA;BB"
>

{- The function f is in point free style -}

> let f = map chr . map (flip (-) 10) . map ord . map toUpper
>
> f "haskell"
">7IA;BB"
>
```

1.11.3 Function Composition in Python

```
def compose(funclist):

    def _(x):
        y = x

        for f in reversed(funclist):
            y = f(y)
        return y

    return _

>>> add10 = lambda x: x + 10

>>> mul3 = lambda x: x * 3

>>> x = 3
>>> a = add10(x)
>>> a
    13
>>> b = mul3(a)
>>> b
    39


>>> def f_without_composition (x):
 ...    a = add10(x)
 ...    b = mul3(a)
```

```
 ...    return b
 ...

>>> f_without_composition(3)
    39

>>> f_without_composition(4)
    42

 # It will create the function f = (mul3 ° add10)(x)
 # The flow is from right to left
 #
 #
 #     (mul3 . add10) 3
 #   =  mul3 (add10 3)
 #   =  mul3 13
 #   =  39
 #
>>> f = compose ([mul3, add10])

>>> f(3)
    39

>>> f(4)
    42

>>> f
    <function __main__.compose.<locals>._>

>>> compose ([add10, mul3])(3)
    39

>>> compose ([add10, mul3])(4)
    42

 #
 # Composition is more intuitive when the flow is from
 # left to right, the functions in the left side are
 # executed first.
 #
```

```
 #

 # Compose Forward
def composef (funclist):

    def _(x):
        y = x
        for f in funclist:
            y = f(y)
        return y

    return _

 #
 #   The symbol (>>) from F# will be used to mean forward composition
 #   here
 #
 #      (add10 >> mul3) 3
 #    = mul3 (add10 3)
 #    = mul3 13
 #    = 39
 #                          add10 >> mul3
 #    Input  ..............................................  Output
 #           .    |----------|           |---------|         .   39
 #   3  ---> .--> |  add10   | --------> |   mul3  | ------->.  ------->
 #           .  3 |----------| 13 =(10+3)|---------|  39     .
 #           .                                  39 = 3 * 13    .
 #           ..............................................
 #
 #  The execution flow is from left to right, in the same order as the
 #  functions are written in the code.
 #

>>> g = composef ([add10, mul3])

>>> g(3)
    39

>>> g(4)
    42
```

```
>>> ### A more useful example: parse the following table:

text = """
 12.23,30.23,892.2323
 23.23,90.23,1000.23
 3563.23,100.23,45.23

"""



 # Unfortunately Python, don't have a favorable syntax to function
 # composition like: composition operator, lightweight lambda and function
 # application without parenthesis.
 #

>>> mapl = lambda f: lambda xs: list(map(f, xs))
>>> filterl = lambda f: lambda xs: list(filter(f, xs))


>>> splitlines = lambda s: s.splitlines()
>>> reject_empty = lambda xs: list(filter(lambda x: x, xs))
>>> strip = lambda s: s.strip()
>>> split = lambda sep: lambda s: s.split(sep)


>>> composef([splitlines])(text)
    ['',
 ' 12.23,30.23,892.2323',
 ' 23.23,90.23,1000.23',
 ' 3563.23,100.23,45.23',
 '']


>>> composef([splitlines, reject_empty])(text)
    [' 12.23,30.23,892.2323',
    ' 23.23,90.23,1000.23',
    ' 3563.23,100.23,45.23']
```

```
>>> composef([splitlines, reject_empty, mapl(strip)])(text)
    ['12.23,30.23,892.2323', '23.23,90.23,1000.23',
    '3563.23,100.23,45.23']


>>> composef([splitlines, reject_empty, mapl(strip), mapl(split(","))])(text)
    [['12.23', '30.23', '892.2323'],
 ['23.23', '90.23', '1000.23'],
 ['3563.23', '100.23', '45.23']]

>>> composef([splitlines, reject_empty, mapl(strip), mapl(split(",")), mapl(mapl(float)
    [[12.23000, 30.23000, 892.23230],
 [23.23000, 90.23000, 1000.23000],
 [3563.23000, 100.23000, 45.23000]]

parse_csvtable =  composef(
    [splitlines,
    reject_empty,
    mapl(strip),
    mapl(split(",")),
    mapl(mapl(float))]
    )


>>> parse_csvtable(text)
    [[12.23000, 30.23000, 892.23230],
 [23.23000, 90.23000, 1000.23000],
 [3563.23000, 100.23000, 45.23000]]

    #  Notice there is three maps together, so that it can be optimized
    #  each map is like a for loop, by composing the functions in map1,
    #  map2 and map3 the code can be more faster.
    #
    # parse_csvtable =  composef(
    # [splitlines,
    # reject_empty,
    # mapl(strip),          ---> map1
    # mapl(split(",")),     ---> map2
```

```
    # mapl(mapl(float))]    ---> map3
    # )


parse_csvtable_optmized =  composef(
    [splitlines,
    reject_empty,
    mapl(composef([strip, split(","), mapl(float)]))
    ])

>>> parse_csvtable_optmized(text)
    [[12.23000, 30.23000, 892.23230],
 [23.23000, 90.23000, 1000.23000],
 [3563.23000, 100.23000, 45.23000]]
```

1.11.4 Function Composition in F#

F# uses the operator («) for composition which is similar to Haskell composition operator (.) dot. It also uses the operator (») for forward composition that performs the operation in the inverse order of operator («).

Composition Operator («):

```
- (<<) ;;
val it : (('a -> 'b) -> ('c -> 'a) -> 'c -> 'b)
>

> let h x = x + 3 ;;

val h : x:int -> int

> let g x = x * 5 ;;

val g : x:int -> int

- let m x = x - 4 ;;

val m : x:int -> int

// The composition is performed in the same way as math composition
// from right to left.
```

```
//
- h (g 4) ;;
val it : int = 23
>
-
- (h << g) 4 ;;
val it : int = 23
>



> m (h (g 4)) ;;
val it : int = 19
>
- (m << h << g) 4 ;;
val it : int = 19
>

// It is the same as math composition: f(x) = m ° h ° g
//
- let f = m << h << g ;;

val f : (int -> int)

> f 4 ;;
val it : int = 19
>
```

Forward Composition Operator (»):

```
> (>>) ;;
val it : (('a -> 'b) -> ('b -> 'c) -> 'a -> 'c)

> let h x = x + 3 ;;

val h : x:int -> int

> let g x = x * 5 ;;

val g : x:int -> int
```

```
- let m x = x - 4 ;;

val m : x:int -> int


- h (g 4) ;;
val it : int = 23
>

- (g >> h) 4 ;;
val it : int = 23
>
- let f = g >> h ;;

val f : (int -> int)

> f 4 ;;
val it : int = 23
>


- m (h (g 4)) ;;
val it : int = 19
>
-

- (g >> h >> m ) 4 ;;
val it : int = 19
>
-

// The argument is seen flowing from left to right, in the inverse
// order of math composition and Haskell composition operator (.) dot,
// which is more easier to read.
//
//  (g >> h >> m) 4 => It is seen as passing through each function
//
//   Evaluating:  g >> h >> m
//
```

```
//  Input              f =  g >> h >> m                          Output
//       ..........................................................
//       .                                                       .
//       .   |-----------|      |-----------|     |-----------|  .
//     ----> |g = x * 5  ||---->| h = x + 3 |---->| m = x - 4 |----->
//   4   .   |-----------|  20  |-----------| 23  |-----------|  .  19
//       .       = 4 * 5 = 20    = 20 + 3 = 23    = 23 - 4 = 19  .
//       ..........................................................
//

- let f = g >> h >> m ;;

val f : (int -> int)

> f 4 ;;
val it : int = 19
>
```

F# Argument Piping operator (|>)

```
- let h x = x + 3 ;;

val h : x:int -> int

> let g x = x * 5 ;;

val g : x:int -> int

>  let m x = x - 4 ;;

val m : x:int -> int

>

// The piping operator feeds the function input forward.
//
- (|>) ;;
val it : ('a -> ('a -> 'b) -> 'b)
>
```

```
- (g 4) ;;
val it : int = 20
>
- 4 |> g ;;
val it : int = 20
>

// It is the same as:  4 |> g |> h
//
- h (g 4) ;;
val it : int = 23
>

- 4 |> g |> h ;;
val it : int = 23
>

// It is the same as:  4 |> g |> h |> m ;;
//
- m (h (g 4)) ;;
val it : int = 19
>

- 4 |> g |> h |> m ;;
val it : int = 19
>
```

Example of a non numeric function composition:

```
let text = "
 12.23,30.23,892.2323
 23.23,90.23,1000.23
 3563.23,100.23,45.23

"

// Negate predicate. Invert a predicate function.
//
> let neg pred = fun x -> not (pred x) ;;
```

```
val neg : pred:('a -> bool) -> x:'a -> bool



let split_string sep str =
    List.ofArray  (System.Text.RegularExpressions.Regex.Split (str, sep))

> let split_lines = split_string "\n" ;;

val split_lines : (string -> string list)

> let trim_string (s: string) = s.Trim() ;;

val trim_string : s:string -> string

- let is_string_empty (s: string) = s.Length = 0
- ;;

val is_string_emtpy : s:string -> bool

- text |> split_lines ;;
val it : string list =
  [""; " 12.23,30.23,892.2323"; " 23.23,90.23,1000.23";
   " 3563.23,100.23,45.23"; ""; ""]
>

- text |> split_lines |> List.filter (neg is_string_empty) ;;
val it : string list =
  [" 12.23,30.23,892.2323"; " 23.23,90.23,1000.23"; " 3563.23,100.23,45.23"]
>

- text |> split_lines |> List.filter (neg is_string_empty) |> List.map trim_stri- ng ;
val it : string list =
  ["12.23,30.23,892.2323"; "23.23,90.23,1000.23"; "3563.23,100.23,45.23"]
>
-

- text |> split_lines |> List.filter (neg is_string_empty) |> List.map trim_stri- ng |>
val it : string list list =
  [["12.23"; "30.23"; "892.2323"]; ["23.23"; "90.23"; "1000.23"];
```

```
   ["3563.23"; "100.23"; "45.23"]]
>
-

- text |> split_lines |> List.filter (neg is_string_empty) |> List.map trim_stri- ng |>
val it : float list list =
  [[12.23; 30.23; 892.2323]; [23.23; 90.23; 1000.23]; [3563.23; 100.23; 45.23]]
>

// Or in multiple lines:

text
|> split_lines
|> List.filter (neg is_string_empty)
|> List.map trim_string
|> List.map (split_string ",")
|> List.map (List.map float)
;;

val it : float list list =
  [[12.23; 30.23; 892.2323]; [23.23; 90.23; 1000.23]; [3563.23; 100.23; 45.23]]
>

// Then transformed into a function:
//
let parse_csv text =
   text
   |> split_lines
   |> List.filter (neg is_string_empty)
   |> List.map trim_string
   |> List.map (split_string ",")
   |> List.map (List.map float)
;;

val parse_csv : text:string -> float list list

> parse_csv text ;;
val it : float list list =
  [[12.23; 30.23; 892.2323]; [23.23; 90.23; 1000.23]; [3563.23; 100.23; 45.23]]
>
```

```
-

// This operation can be optimized with function (forward) composition.
//
let parse_csv2 text =
   text
   |> split_lines
   |> List.filter (neg is_string_empty)
   |> List.map (trim_string >> (split_string ",") >> (List.map float))
;;

val parse_csv2 : text:string -> float list list

> parse_csv2 text ;;
val it : float list list =
  [[12.23; 30.23; 892.2323]; [23.23; 90.23; 1000.23]; [3563.23; 100.23; 45.23]]
>

// It could be implemented using the math compostion (<<) operator.
// in the same as in Haskell.
//
let parse_csv3 text =
   text
   |> split_lines
   |> List.filter (neg is_string_empty)
   |> List.map ((List.map float) << (split_string ",") << trim_string)
;;

val parse_csv3 : text:string -> float list list

- parse_csv3 text ;;
val it : float list list =
  [[12.23; 30.23; 892.2323]; [23.23; 90.23; 1000.23]; [3563.23; 100.23; 45.23]]
>

//  934.6923 = 12.23 + 30.23 + 892.2323
//
- parse_csv3 text |> List.map List.sum ;;
val it : float list = [934.6923; 1113.69; 3708.69]
>
```

```
-

- parse_csv3 text |> List.map List.sum |> List.sum ;;
val it : float = 5757.0723
>
```

1.12 Functors

1.12.1 Overview

Functors are type constructors that can be mapped over like lists are mapped with map function. Its concept comes from category theory and like many mathematical concepts it is defined by the laws that it satisfies: Functor's laws.

In Haskell functors are instances of the type class Functor that must implement the function fmap for each functor implementation.

```
class  Functor f  where
  fmap :: (a -> b) -> f a -> f b
```

```
Map Diagram                                    Functor Diagram


         f :: a -> b              =====>         f :: a -> b
  a   ----------------------> b             a ------------------------> b

        map f :: [a] -> [b]                      fmap f :: F a -> F b
  [a] ----------------------> [b]          F a ----------------------> F b
```

```
Where F is a type constructor.
```

A functor must satisfy the laws:

- Identity Law.

  fmap id == id
  Where id is the identity function.

- Composition Law

  fmap (f . g) == fmap f . fmap g
  The composition law can be generalized to:
  fmap (f1 . f2 . f3 . . . fn) = fmap f1 . fmap f2 . fmap f3 . . . fmap fn

1.12.2 List all Functor instances

```
> :info Functor
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
   -- Defined in ‘GHC.Base’
instance Functor (Either a) -- Defined in ‘Data.Either’
instance Functor Maybe -- Defined in ‘Data.Maybe’
instance Functor ZipList -- Defined in ‘Control.Applicative’
instance Monad m => Functor (WrappedMonad m)
  -- Defined in ‘Control.Applicative’
instance Functor (Const m) -- Defined in ‘Control.Applicative’
instance Functor [] -- Defined in ‘GHC.Base’
instance Functor IO -- Defined in ‘GHC.Base’
instance Functor ((->) r) -- Defined in ‘GHC.Base’
instance Functor ((,) a) -- Defined in ‘GHC.Base’
>

> :info IO
newtype IO a
  = GHC.Types.IO (GHC.Prim.State# GHC.Prim.RealWorld
                  -> (# GHC.Prim.State# GHC.Prim.RealWorld, a #))
   -- Defined in ‘GHC.Types’
instance Monad IO -- Defined in ‘GHC.Base’
instance Functor IO -- Defined in ‘GHC.Base’
instance Applicative IO -- Defined in ‘Control.Applicative’
>
> :info Maybe
data Maybe a = Nothing | Just a  -- Defined in ‘Data.Maybe’
instance Eq a => Eq (Maybe a) -- Defined in ‘Data.Maybe’
instance Monad Maybe -- Defined in ‘Data.Maybe’
instance Functor Maybe -- Defined in ‘Data.Maybe’
instance Ord a => Ord (Maybe a) -- Defined in ‘Data.Maybe’
instance Read a => Read (Maybe a) -- Defined in ‘GHC.Read’
instance Show a => Show (Maybe a) -- Defined in ‘GHC.Show’
instance MonadPlus Maybe -- Defined in ‘Control.Monad’
instance Applicative Maybe -- Defined in ‘Control.Applicative’
instance Alternative Maybe -- Defined in ‘Control.Applicative’
>
```

1.12.3 Functors Implementations

1. Identity

```
-- File: identity.hs
--
data Id a = Id a  deriving (Show, Eq)

instance Functor Id where
  fmap f (Id a) = Id (f a)

 -- Specialized version of fmap
 --
fmap_id :: (a -> b) -> (Id a -> Id b)
fmap_id f = fmap f

-- -- End of file identity.hs ------
-------------------------------------

> :load /tmp/identity.hs

> Id 10
Id 10

> fmap (\x -> x + 3) (Id 30)
Id 33
>

> let plus5 =  \x -> x + 5
> :t plus5
plus5 :: Integer -> Integer
>

> plus5 10
15
>

> fmap plus5 (Id 30)
Id 35
>
```

```
> let fmap_plus5 = fmap plus5

<interactive>:68:18:
    No instance for (Functor f0) arising from a use of ‘fmap’
    The type variable ‘f0’ is ambiguous

 -- Solution:


> let fmap_plus5 = fmap plus5 :: Id Integer -> Id Integer
> :t fmap_plus5
fmap_plus5 :: Id Integer -> Id Integer
>

> fmap_plus5 (Id 30)
Id 35
>



> :t fmap_id
fmap_id :: (a -> b) -> Id a -> Id b
>

> fmap_id sqrt (Id 100.0)
Id 10.0
>

> let sqrt_id = fmap_id sqrt
> :t sqrt_id
sqrt_id :: Id Double -> Id Double
>

> sqrt_id (Id 100.0)
Id 10.0
>
```

2. List

   For lists the function fmap is equal to map.

```
instance Functor [] where
  fmap = map

> map (\x -> x + 3) [1, 2, 3]
[4,5,6]
> fmap (\x -> x + 3) [1, 2, 3]
[4,5,6]
>

> let f = fmap (\x -> x + 3) :: [Int] -> [Int]
> :t f
f :: [Int] -> [Int]

> f [1, 2, 3, 4]
[4,5,6,7]
>
```

Alternative Implementation of list functor:

```
-- File: list_functor.hs
--

data List a  = Cons a (List a)  | Nil
  deriving (Show, Eq)


instance Functor List where

  fmap f xss = case xss of
               Nil          -> Nil
               Cons x xs    -> Cons (f x) (fmap f xs)


-- End of file -------------
---------------------------

> :load /tmp/list_functor.hs

> Cons 10 (Cons 20 (Cons 30 Nil))
Cons 10 (Cons 20 (Cons 30 Nil))
```

```
>

> let xs = Cons 10 (Cons 20 (Cons 30 Nil))
> xs
Cons 10 (Cons 20 (Cons 30 Nil))
> :t xs
xs :: List Integer
>

> fmap (\x -> x + 5) xs
Cons 15 (Cons 25 (Cons 35 Nil))
>

> let fm = fmap (\x -> x + 5) :: List Integer -> List Integer

> fm  xs
Cons 15 (Cons 25 (Cons 35 Nil))
>
```

3. Maybe / Option

```
data Maybe a = Just a | Nothing deriving (Eq, Show)

instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

Example:

```
> fmap (\x -> x + 10) (Just 5)
Just 15
>
> fmap (\x -> x + 10) Nothing
Nothing
>

> let f = \x -> x + 10
> :t f
f :: Integer -> Integer
```

```
>

> let fmap_f = fmap f :: Maybe Integer -> Maybe Integer
> fmap_f (Just 20)
Just 30
>

> fmap_f Nothing
Nothing
>

> import Text.Read (readMaybe)


> readMaybe "100" :: Maybe Integer
Just 100
>
> readMaybe "asd100" :: Maybe Integer
Nothing
>

> let parseInteger str = readMaybe str :: Maybe Integer
>
> :t parseInteger
parseInteger :: String -> Maybe Integer
>

> parseInteger "100"
Just 100
>
> parseInteger "Not a number"
Nothing
>

> fmap (\x -> x + 10) (parseInteger "200")
Just 210
>
> fmap (\x -> x + 10) (parseInteger "2sadas00")
Nothing
>
```

```
 -- Specialized version of fmap
 --
fmap_maybe :: (a -> b) -> (Maybe a -> Maybe b)
fmap_maybe func = fmap func

> fmap_maybe (\x -> x + 10) (Just 10)
Just 20
>

> fmap_maybe (\x -> x + 10) Nothing
Nothing
>
```

4. Either

   The type constructor Either is similar to Maybe and it can short circuit a computation in the similar way to Maybe, however it allows to attach an error message.

```
data Either a b = Left a | Right b

instance Functor (Either a) where
    fmap f (Right x) = Right (f x)
    fmap f (Left x) = Left x
```

   Example:

```
--
-- File: either_functor.hs
-------------------------


 --  Specialized version of fmap to Either type
 --  constructor like map is specialized for lists.
 --
fmap_either :: (a -> b) -> (Either s a -> Either s b)
fmap_either f = fmap f

import Text.Read (readMaybe)
```

```
data ErrorCode =
        ParserError
      | InvalidInput
      deriving (Eq, Show)

describeErrorCode ParserError  = "The parser failed."
describeErrorCode InvalidInput = "Input out function domain."

describeError (Right x)    = Right x
describeError (Left  code) = Left (describeErrorCode code)

parseDouble :: String -> Either String Double
parseDouble  str =
    case (readMaybe str :: Maybe Double) of
    Just x  ->  Right x
    Nothing ->  Left  "Error: Not a Double"

sqrt_safe :: Double -> Either String Double
sqrt_safe x =
    if x >= 0
    then (Right x)
    else (Left "Error: Square root of negative number")


parseDouble2 :: String -> Either ErrorCode Double
parseDouble2  str =
    case (readMaybe str :: Maybe Double) of
    Just x  ->  Right x
    Nothing ->  Left  ParserError

sqrt_safe2 :: Double -> Either ErrorCode Double
sqrt_safe2 x =
    if x >= 0
    then (Right x)
    else (Left InvalidInput)

---------------------------------------
--             End of file           -
---------------------------------------
```

```
> :load /tmp/either_functor.hs

> sqrt_safe 100
Right 100.0
>
> sqrt_safe (-100)
Left "Error: Square root of negative number"
>

> sqrt_safe2 100
Right 100.0
>
> sqrt_safe2 (-100)
Left InvalidInput
>


> parseDouble "100.25e3"
Right 100250.0
>

> parseDouble "not a double"
Left "Error: Not a Double"
>

> parseDouble2 "-200.3"
Right (-200.3)
>

> parseDouble2 "Not a double"
Left ParserError
>

> fmap sqrt (Right 100.0)
Right 10.0
>
> fmap sqrt (Left "Error not found")
Left "Error not found"
>
```

```
> let fmap_sqrt = fmap sqrt :: Either String Double -> Either String Double
> fmap_sqrt (parseDouble "400.0")
Right 20.0
>
> fmap_sqrt (parseDouble "4adsfdas00.0")
Left "Error: Not a Double"
>


> describeError (sqrt_safe2 100)
Right 100.0
>
> describeError (sqrt_safe2 (-100))
Left "Input out function domain."
>

> describeError (parseDouble2 "200.23")
Right 200.23
>

> describeError (parseDouble2 "2dsfsd00.23")
Left "The parser failed."
>

> fmap_either (\x -> x + 10) (Right 200)
Right 210
>

> let sqrt_either = fmap_either sqrt
>
> :t sqrt_either
sqrt_either :: Either s Double -> Either s Double
>

> sqrt_either (Right 100.0)
Right 10.0
>
> sqrt_either (Left "Failed to fetch data")
Left "Failed to fetch data"
```

```
>
```

5. IO

   Source: Book learnyouahaskell

```
instance Functor IO where
    fmap f action = do
        result <- action
        return (f result)

-- File: fmap_io.hs
--

fmap_io :: (a -> b) -> (IO a -> IO b)
fmap_io f = fmap f

--
------------------

> :load /tmp/fmap_io.hs

> import System.Directory (getDirectoryContents)
>

> :t getDirectoryContents "/etc/R"
getDirectoryContents "/etc/R" :: IO [FilePath]
>

> getDirectoryContents "/etc/R"
["Renviron",".","ldpaths","..","repositories","Makeconf","Renviron.site","Rprofile
>

 -- It will fail because the data is inside an IO container
 -- and can only be extracted inside another IO container
 -- or IO Monad.
 --
> length (getDirectoryContents "/etc/R")

<interactive>:165:9:
```

```
    Couldn't match expected type ‘[a0]’
                with actual type ‘IO [FilePath]’
    In the return type of a call of ‘getDirectoryContents’
    In the first argument of ‘length’, namely
      ‘(getDirectoryContents "/etc/R")’
    In the expression: length (getDirectoryContents "/etc/R")


> fmap length (getDirectoryContents "/etc/R")
8
> :t fmap length (getDirectoryContents "/etc/R")
fmap length (getDirectoryContents "/etc/R") :: IO Int
>


> fmap_io length (getDirectoryContents "/etc/R")
8
>

> :t length
length :: [a] -> Int
>

> let length_io = fmap_io length
>

> :t length_io
length_io :: IO [a] -> IO Int
>

> fmap_io length (getDirectoryContents "/etc/R")
8


-- In the REPL is possible to extract the data wrapped inside an IO
-- type constructor.
--

> dirlist <- getDirectoryContents "/etc/R"
> :t dirlist
```

```
dirlist :: [FilePath]
>
> dirlist
["Renviron",".","ldpaths","..","repositories","Makeconf","Renviron.site","Rprofile
>
> length dirlist
8
>

> :t length dirlist
length dirlist :: Int
>
```

1.13 Monads

1.13.1 Overview

A monad is a concept from Category Theory, which is defined by three things:

- a type constructor m that wraps a, parameter a;
- a return (unit) function: takes a value from a plain type and puts it into a monadic container using the constructor, creating a monadic value. The return operator must not be confused with the "return" from a function in a imperative language. This operator is also known as unit, lift, pure and point. It is a polymorphic constructor.
- a bind operator (»=). It takes as its arguments a monadic value and a function from a plain type to a monadic value, and returns a new monadic value.

In Haskell the type class Monad specifies the type signature of all its instances. Each Monad implementation must have the type signature that matches the Monad type class.

```
class Monad m where

    -- Constructor (aka unit or lift)
    --
    return :: a -> m a
```

```
-- Bind operator
--
(>>=)  :: m a -> (a -> m b) -> m b

(>>)   :: m a -> m b -> m b

fail   :: String -> m a
```

1.13.2 List Monad

1. List Monad in Haskell

   The list Monad is used for non-deterministic computations where there is a unknown number of results. It is useful for constraint solving: solve a problem by trying all possibilities by brute force.

   In Haskell it is defined as an instance of Monad type class:

```
instance  Monad []  where
    m >>= k          = concat (map k m)
    return x         = [x]
    fail s           = []
```

   Examples:

   - return wraps a value into a list

```
> :t return
return :: Monad m => a -> m a
>
 -- Wraps a value into a list
 --
> return 10 :: [Int]
[10]
>
```

   Bind operator for list monad:

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

```
>  [10,20,30] >>= \x -> [2*x, x+5]
[20,15,40,25,60,35]
>

-- It is equivalent to:
--
--

> map ( \x -> [2*x, x+5] ) [10, 20, 30]
[[20,15],[40,25],[60,35]]
>
> concat (map ( \x -> [2*x, x+5] ) [10, 20, 30])
[20,15,40,25,60,35]
>
```

Do notation:

```
  The do notation is a syntax sugar to ->
```

```
Do Notation:                    Do Notation Dessugarized:

cartesianProduct  = do          cartesianProduct =
   x <- [1, 2, 3, 4]              [1, 2, 3, 4] >>= \x ->
   y <- ["a", "b"]                ["a", "b"]   >>= \y ->
   return (x, y)                  return (x, y)

                                Or

                                cartesianProduct =
                                  bind [1 2, 3, 4]  (\x ->
                                  bind ["a", "b"]   (\y ->
                                  unit (x, y)))
```

Do notation examples for List monad:

```
-- file cartesian.hs
```

```
--
-- Run in the repl:
--                  :load cartesian.hs
--

cartesianProduct = do
    x <- [1, 2, 3, 4]
    y <- ["a", "b"]
    return (x, y)

--  End of file: cartesian.hs
-- -----------------

> cartesianProduct
[(1,"a"),(1,"b"),(2,"a"),(2,"b"),(3,"a"),(3,"b"),(4,"a"),(4,"b")]
>

 -- Or it can be typed in the repl directly:
 --


> :set +m  -- Enable multiline paste
>

--  Or copy the following code in the repl
--  by typing :set +m to enable multiline paste
--

let cartesianProduct = do
    x <- [1, 2, 3, 4]
    y <- ["a", "b"]
    return (x, y)

> :set +m  -- paste
>
> let cartesianProduct = do
*Main Control.Exception E Text.Read|     x <- [1, 2, 3, 4]
*Main Control.Exception E Text.Read|     y <- ["a", "b"]
*Main Control.Exception E Text.Read|     return (x, y)
*Main Control.Exception E Text.Read|
```

```
>


 --
 -- Or: Dessugarizing

> [1, 2, 3, 4] >>= \x -> ["a", "b"] >>= \y -> return (x, y)
[(1,"a"),(1,"b"),(2,"a"),(2,"b"),(3,"a"),(3,"b"),(4,"a"),(4,"b")]
>

cartesian :: [a] -> [b] -> [(a, b)]
cartesian xs ys = do
    x <- xs
    y <- ys
    return (x, y)

> cartesian [1, 2, 3, 4] ["a", "b"]
[(1,"a"),(1,"b"),(2,"a"),(2,"b"),(3,"a"),(3,"b"),(4,"a"),(4,"b")]
>


-- Returns all possible combinations between a, b and c
--
--
triples :: [a] -> [b] -> [c] -> [(a, b, c)]
triples  xs ys zs = do
  x <- xs
  y <- ys
  z <- zs
  return (x, y, z)

--   The triples have 24 results
--
--   x -> 2 possibilities
--   y -> 3 possibilities
--   z -> 4 possibilities
--
--  Total of possibilities:  2 * 3 * 4 = 24
--  the computation will return 24 results
--
--
```

```
> triples [1, 2] ["a", "b", "c"] ["x", "y", "z", "w"]
[(1,"a","x"),(1,"a","y"),(1,"a","z"),(1,"a","w"),(1,"b","x"),
(1,"b","y"),(1,"b","z"),(1,"b","w"),(1,"c","x"),(1,"c","y"),
(1,"c","z"),(1,"c","w"),(2,"a","x"),(2,"a","y"),(2,"a","z"),
(2,"a","w"),(2,"b","x"),(2,"b","y"),(2,"b","z"),(2,"b","w"),
(2,"c","x"),(2,"c","y"),(2,"c","z"),(2,"c","w")]

> length ( triples [1, 2] ["a", "b", "c"] ["x", "y", "z", "w"] )
24
>


--
--  Find all numbers for which a ^ 2 + b ^ 2 = c ^ 2
--  up to 100:
--
--  There is 100 x 100 x 100 = 1,000,000 of combinations
--  to be tested:
--
pthytriples = do
    a <- [1 .. 100]
    b <- [1 .. 100]
    c <- [1 .. 100]

    if (a ^ 2 + b ^ 2 == c ^ 2)
      then (return (Just (a, b, c)))
      else (return Nothing)

> import Data.Maybe (catMaybes)

> take 10 (catMaybes pthytriples)
[(3,4,5),(4,3,5),(5,12,13),(6,8,10),(7,24,25),(8,6,10),(8,15,17),(9,12,15),(9,40,4
>

-- Example: Find all possible values of a functions applied
--          to all combinations possible of 3 lists:
--
applyComb3 = do
   x <- [1, 2, 3]
   y <- [9, 8]
   z <- [3, 8, 7, 4]
```

```
   return ([x, y, z], 100 * x + 10 *  y + z)

> applyComb3
[([1,9,3],193),([1,9,8],198),([1,9,7],197),([1,9,4],194) ...]


-- Example: Crack a 4 letter password using brute force
--
--


import Data.List (find)
import Data.Maybe (isJust)

alphabet = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"

make_password :: String -> String -> Bool
make_password password input = password == input

-- It will try 62 combinations for each letter
-- wich means it will try up to 62 x 62 x 62 x 62 = 14,776,336
-- (=~ 15 millions) combinations at the worst case.
--
--
crack_4pass  pass_function = do
    ch0 <- alphabet
    ch1 <- alphabet
    ch2 <- alphabet
    ch3 <- alphabet
    let trial = [ch0, ch1, ch2, ch3] in
      if   pass_function trial
      then return (Just trial)
      else return Nothing

crackpass pass_function =
   find isJust (crack_4pass pass_function)

passwd1 = make_password "fX87"
```

```
> :set +s

> crackpass passwd1
Just (Just "fX87")
(2.00 secs, 434045068 bytes)
>

> crackpass (make_password "2f8z")
Just (Just "2f8z")
(18.19 secs, 4038359812 bytes)
>
```

2. List Monad in OCaml

```
module ListM =
  struct
  let bind ma f = List.concat (List.map f ma)

  let (>>=) = bind

  (* return  *)
  let unit a  = [a]

end
;;


module ListM :
  sig
    val bind : 'a list -> ('a -> 'b list) -> 'b list
    val ( >>= ) : 'a list -> ('a -> 'b list) -> 'b list
    val unit : 'a -> 'a list
  end

(*
 Haskel Code:

 cartesian :: [a] -> [b] -> [(a, b)]              cartesian :: [a] -> [b] -> [(a,
 cartesian xs ys = do                             carteasian xs ys =
    x <- xs                                ==>>        xs >>= \x ->
```

```
    y <- ys                                          ys >>= \y ->
    return (x, y)                                    return (x, y)


 cartesian :: [a] -> [b] -> [(a, b)]
 carteasian xs ys =
   bind xs (\x ->
   bind ys (\y ->
   return (x, y)))

*)


let cartesian xs ys =
    let open ListM in
    xs >>= fun x ->
    ys >>= fun y ->
    unit (x, y)
;;

val cartesian : 'a list -> 'b list -> ('a * 'b) list = <fun>


>   cartesian [1; 2; 3; 4; ] ["a"; "b"; "c"]  ;;
- : (int * string) list =
[(1, "a"); (1, "b"); (1, "c"); (2, "a"); (2, "b"); (2, "c"); (3, "a");
 (3, "b"); (3, "c"); (4, "a"); (4, "b"); (4, "c")]


(*

Haskel Code                                  Do-natation Dessugarized

triples :: [a] [b] [c] -> [(a, b, c)]        triples :: [a] [b] [c] -> [(a, b, c)]
triples xs ys zs = do                        tripes xs ys zs =
    x <- xs                                      xs >>= \x ->
    y <- ys                      ==>             ys >>= \y ->
    z <- zs                                      zs >>= \z ->
    return (x, y, z)                             return (x, y, z)
```

```
*)

let triples (xs: 'a list) (ys: 'b list) (zs: 'c list) : ('a * 'b * 'c) list =
    let open ListM in
    xs >>= fun x ->
    ys >>= fun y ->
    zs >>= fun z ->
    unit (x, y, z)
;;

val triples : 'a list -> 'b list -> 'c list -> ('a * 'b * 'c) list = <fun>


>  triples ["x"; "z"; "w"]  [Some 10; None] [1; 2; 3; 4; 5] ;;
- : (string * int option * int) list =
[("x", Some 10, 1); ("x", Some 10, 2); ("x", Some 10, 3); ("x", Some 10, 4);
 ("x", Some 10, 5); ("x", None, 1); ("x", None, 2); ("x", None, 3);
 ("x", None, 4); ("x", None, 5); ("z", Some 10, 1); ("z", Some 10, 2);
 ("z", Some 10, 3); ("z", Some 10, 4); ("z", Some 10, 5); ("z", None, 1);
 ("z", None, 2); ("z", None, 3); ("z", None, 4); ("z", None, 5);
 ("w", Some 10, 1); ("w", Some 10, 2); ("w", Some 10, 3); ("w", Some 10, 4);
 ("w", Some 10, 5); ("w", None, 1); ("w", None, 2); ("w", None, 3);
 ("w", None, 4); ("w", None, 5)]
```

3. List Monad in F#

   Example without syntax sugar:

```
module ListM =

  let bind ma f = List.concat (List.map f ma)

  let (>>=) = bind

  (* return  *)
  let unit a  = [a]
;;


module ListM = begin
```

```
  val bind : ma:'a list -> f:('a -> 'b list) -> 'b list
  val ( >>= ) : ('a list -> ('a -> 'b list) -> 'b list)
  val unit : a:'a -> 'a list
end


(*  Example:

cartesian :: [a] -> [b] -> [(a, b)]
cartesian xs ys = do
    x <- xs
    y <- ys
    return (x, y)

> cartesian [1, 2, 3, 4] ["a", "b", "c"]
*)


let cartesian xs ys =
    let (>>=) = ListM.(>>=) in
    let unit  = ListM.unit in

    xs >>= fun x ->
    ys >>= fun y ->
    unit (x, y)
;;

val cartesian : 'a list -> 'b list -> ('a * 'b) list = <fun>


>
> cartesian [1; 2; 3; 4; ] ["a"; "b"; "c"]  ;;
val it : (int * string) list =
  [(1, "a"); (1, "b"); (1, "c");
   (2, "a"); (2, "b"); (2, "c");
   (3, "a"); (3, "b"); (3, "c");
   (4, "a"); (4, "b"); (4, "c")]
>
```

```
(*

F# List type is eager evaluated so the it will
really loop over 100 * 100 * 100 = 100,000,000
of combinations:

pthytriples = do
    a <- [1 .. 100]
    b <- [1 .. 100]
    c <- [1 .. 100]

    if (a ^ 2 + b ^ 2 == c ^ 2)
      then (return (Just (a, b, c)))
      else (return Nothing)


*)


(* Tail recursive function

*)
let range start stop step  =

    let rec range_aux start acc =
        if start >= stop
        then  List.rev acc
        else  range_aux (start + step)  (start::acc)

    in range_aux start []
  ;;

val range : start:int -> stop:int -> step:int -> int list

>  range 1 11 1 ;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]


let pthytriples () =
    let (>>=) = ListM.(>>=) in
```

```
    let unit  = ListM.unit in

    range 1 101 1 >>= fun a ->
    range 1 101 1 >>= fun b ->
    range 1 101 1 >>= fun c ->
    if (a * a + b * b = c * c)
    then unit (Some (a, b, c))
    else unit None
;;

val pthytriples : unit -> (int * int * int) option list


let option_to_list opt_list =
    List.foldBack                     (* Fold right *)
      (fun x acc -> match x with
                    | Some a -> a::acc
                    | None   -> acc
      )
      opt_list
      []
;;


- option_to_list (pthytriples ()) ;;

val it : (int * int * int) list =
  [(3, 4, 5); (4, 3, 5); (5, 12, 13); (6, 8, 10); (7, 24, 25); (8, 6, 10);
   (8, 15, 17); (9, 12, 15); (9, 40, 41); (10, 24, 26); (11, 60, 61);
   (12, 5, 13); (12, 9, 15); (12, 16, 20); (12, 35, 37); (13, 84, 85);
   (14, 48, 50); (15, 8, 17); (15, 20, 25); (15, 36, 39); (16, 12, 20);
   (16, 30, 34); (16, 63, 65); (18, 24, 30); (18, 80, 82); (20, 15, 25);
   (20, 21, 29); (20, 48, 52); (21, 20, 29); (21, 28, 35); (21, 72, 75);
   (24, 7, 25); (24, 10, 26); (24, 18, 30); (24, 32, 40); (24, 45, 51);
   (24, 70, 74); (25, 60, 65); (27, 36, 45); (28, 21, 35); (28, 45, 53);
   (28, 96, 100); (30, 16, 34); (30, 40, 50); (30, 72, 78); (32, 24, 40);
   (32, 60, 68); (33, 44, 55); (33, 56, 65); (35, 12, 37); (35, 84, 91);
   (36, 15, 39); (36, 27, 45); (36, 48, 60); (36, 77, 85); (39, 52, 65);
   (39, 80, 89); (40, 9, 41); (40, 30, 50); (40, 42, 58); (40, 75, 85);
   (42, 40, 58); (42, 56, 70); (44, 33, 55); (45, 24, 51); (45, 28, 53);
```

```
   (45, 60, 75); (48, 14, 50); (48, 20, 52); (48, 36, 60); (48, 55, 73);
   (48, 64, 80); (51, 68, 85); (52, 39, 65); (54, 72, 90); (55, 48, 73);
   (56, 33, 65); (56, 42, 70); (57, 76, 95); (60, 11, 61); (60, 25, 65);
   (60, 32, 68); (60, 45, 75); (60, 63, 87); (60, 80, 100); (63, 16, 65);
   (63, 60, 87); (64, 48, 80); (65, 72, 97); (68, 51, 85); (70, 24, 74);
   (72, 21, 75); (72, 30, 78); (72, 54, 90); (72, 65, 97); (75, 40, 85);
   (76, 57, 95); (77, 36, 85); (80, 18, 82); (80, 39, 89); ...]
>
```

Example with F# "workflow" or "computation expression" syntax:

```
- List.concat [[1]; []; [2; 3; 4; 5]; [10; 20]] ;;
val it : int list = [1; 2; 3; 4; 5; 10; 20]
>

(* The F# workflow works like Haskell do-noation

*)
type ListBuilder () =
    member this.Bind(xs, f) = List.concat (List.map f xs)

    member this.Return(x) = [x]
;;


type ListBuilder =
  class
    new : unit -> ListBuilder
    member Bind : xs:'b list * f:('b -> 'c list) -> 'c list
    member Return : x:'a -> 'a list
  end

let listDo = ListBuilder () ;;

val listDo : ListBuilder

(*
cartesian :: [a] -> [b] -> [(a, b)]
cartesian xs ys = do
    x <- xs
```

```
    y <- ys
    return (x, y)
*)

let cartesian xs ys =
    listDo {
      let! x = xs
      let! y = ys
      return (x, y)
   }
;;

val cartesian : xs:'a list -> ys:'b list -> ('a * 'b) list

>  cartesian [1; 2; 3; 4; ] ["a"; "b"; "c"] ;;
val it : (int * string) list =
  [(1, "a"); (1, "b"); (1, "c");
   (2, "a"); (2, "b"); (2, "c");
   (3, "a"); (3, "b"); (3, "c");
   (4, "a"); (4, "b"); (4, "c")]
>
```

4. List Monad in Python

```
from functools import reduce

def concat(xss):
    "concat :: [[a]] -> [a]"
    return  reduce(lambda acc, x: acc + x, xss, [])


def listUnit (x):
    "listUnit :: x -> [x]"
    return [x]

def listBind (xss, f):
    "listBind :: [a] -> (a -> [b]) -> [b] "
    return concat(map(f, xss))

# Haskel Code:
```

```
#
# cartesian :: [a] -> [b] -> [(a, b)]              cartesian :: [a] -> [b] -> [(a,
# cartesian xs ys = do                             carteasian xs ys =
#    x <- xs                              ==>>        xs >>= \x ->
#    y <- ys                                          ys >>= \y ->
#    return (x, y)                                    return (x, y)
#
#
# cartesian :: [a] -> [b] -> [(a, b)]
# carteasian xs ys =
#   bind xs (\x ->
#   bind ys (\y ->
#   return (x, y)))
#
def cartesian(xs, ys):

    return listBind(xs,
                    lambda x: listBind(ys,
                    lambda y: listUnit ((x, y))))


def triples(xs, ys, zs):

    return listBind(xs,
                    lambda x: listBind(ys,
                    lambda y: listBind(zs,
                    lambda z: listUnit((x, y, z)))))

>>> cartesian([1, 2, 3, 4], ["a", "b", "c"])
[(1, 'a'), (1, 'b'), (1, 'c'),
(2, 'a'), (2, 'b'), (2, 'c'),
(3, 'a'), (3, 'b'), (3, 'c'),
(4, 'a'), (4, 'b'), (4, 'c')]
>>>

>>> triples([1, 2], ["a", "b", "c"], ["x", "y", "z"])
[(1, 'a', 'x'),
(1, 'a', 'y'),
(1, 'a', 'z'),
(1, 'b', 'x'),
```

```
(1, 'b', 'y'),
(1, 'b', 'z'),
(1, 'c', 'x'),
...
>>>


 # Emulate ML module
 #
class ListM ():

    @classmethod
    def bind(cls, xss, f):
        return concat(map(f, xss))

    @classmethod
    def unit(cls, x):
        return [x]

def cartesian (xs, ys):
    return ListM.bind( xs, lambda x:
           ListM.bind( ys, lambda y:
           ListM.unit ((x, y))))

>>> cartesian([1, 2, 3, 4], ["a", "b", "c"])

[(1, 'a'), (1, 'b'), (1, 'c'),
(2, 'a'), (2, 'b'), (2, 'c'),
(3, 'a'), (3, 'b'), (3, 'c'),
(4, 'a'), (4, 'b'), (4, 'c')]
>>>
```

1.13.3 Maybe / Option Monad

1. Overview

   The Maybe type (Haskell) or Option type (ML, F# and OCaml) is used to indicate that a function might return nothing, the value might not exists or that a computation maight fail. This is helpful to remove nested null checks, avoid null pointer or null object exceptions.

   Some functions are a natural candidates to return a maybe or option

type like parser function, user input validation, lookup functions that search an input in data structure or database and functions with invalid input.

2. Maybe Monad in Haskell

   The Maybe monad ends the computation if any step fails. The module Data.Maybe has useful function to deal with Maybe type.

```
data Maybe a = Just a | Nothing

instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing  >>= k = Nothing

  return = Just
```

   Example: The function below parses two numbers and adds them.

```
--- File: test.hs
--
import Data.List (lookup)
import Text.Read (readMaybe)


  -- Parser functions

parseInt :: String -> Maybe Int
parseInt x = readMaybe x :: Maybe Int

parseDouble :: String -> Maybe Double
parseDouble x = readMaybe x :: Maybe Double

 -- Function with invalid input

sqrtSafe :: Double -> Maybe Double
sqrtSafe x = if x > 0
             then Just (sqrt x)
             else Nothing
```

```
addSafe :: Maybe Double -> Maybe Double -> Maybe Double
addSafe some_x some_y = do
  x <- some_x
  y <- some_y
  return (x + y)


addOneSafe :: Maybe Double -> Double -> Maybe Double
addOneSafe a b = do
  sa <- a
  let c =  3.0 * (b + sa)
  return (sa + c)


 -- s - stands for Some
 --
addSqrtSafe :: Double -> Double -> Maybe Double
addSqrtSafe x y = do
  sx <- sqrtSafe x
  sy <- sqrtSafe y
  return (sx + sy)

-- addSqrtSafe desugarized
--
addSqrtSafe2 x y =
   sqrtSafe x >>= \sx ->
   sqrtSafe y >>= \sy ->
   return (sx +  sy)


 -- End of test file
 --------------------------------

> :load /tmp/test.hs

> :t readMaybe
readMaybe :: Read a => String -> Maybe a
>

> parseInt "100"
```

```
Just 100


 -- Returns nothing if computation fails
 -- instead of perform an exception
 --
> parseInt "not an int."
Nothing
>

> parseDouble "1200"
Just 1200.0
> parseDouble "1200.232"
Just 1200.232
> parseDouble "12e3"
Just 12000.0
> parseDouble "not a valid number"
Nothing
>


-- This haskell function is safe, however in another language
-- it would yield a exception.
--
> sqrt (-100.0)
NaN
>

> sqrtSafe (-1000.0)
Nothing
>

> sqrtSafe 100.0
Just 10.0
>

-- Thea function fmap is a generalization of map and applies a function
-- to the value wrapped in the monad.
--
```

```
> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f b
>

> fmap (\x -> x + 1.0) (Just 10.0)
Just 11.0
> fmap (\x -> x + 1.0) Nothing
Nothing
>
> fmap (\x -> x + 1.0) (parseDouble "10.233")
Just 11.233
> fmap (\x -> x + 1.0) (parseDouble "ase10.2x3")
Nothing
>

--   return function
--

> :t return
return :: Monad m => a -> m a
>

 return 10 :: Maybe Int
Just 10
>

> return "hello world" :: Maybe String
Just "hello world"
>


-- Bind function
--
-- The bind operator or funciton short circuits the computation if
-- it fails at any point
--
--

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

```
>

> Just "100.0" >>= parseDouble >>= sqrtSafe
Just 10.0
>
> Nothing  >>= parseDouble >>= sqrtSafe
Nothing
>

> return "100.0" >>= parseDouble >>= sqrtSafe
Just 10.0
>
> return "-100.0" >>= parseDouble >>= sqrtSafe
Nothing
>


-- Do noation
--
--
-- addSafe some_x some_y = do          addSafe = do
--   x <- some_x                        some_x >>= \x ->
--   y <- some_y                 ==>    some_y >>= \y ->
--   return (x + y)                           return (x + y)
--
--
--
--
--
> :t addSafe
addSafe :: Maybe Double -> Maybe Double -> Maybe Double
>



> addSafe (Just 100.0) (Just 20.0)
Just 120.0
>

> addSafe (Just 100.0) Nothing
```

```
Nothing
>

> addSafe Nothing (Just 100.0)
Nothing
>

> addSafe Nothing Nothing
Nothing
>

> addSafe (parseDouble "100.0") (sqrtSafe 400.0)
Just 120.0
>

> addSafe (Just 100.0) (sqrtSafe (-20.0))
Nothing
>

> addSafe (parseDouble "asd100.0") (sqrtSafe 400.0)
Nothing
>


> :t addSqrtSafe
addSqrtSafe :: Double -> Double -> Maybe Double
>

> addSqrtSafe 100.0 400.0
Just 30.0
>

> addSqrtSafe (-100.0) 400.0
Nothing
>

> addSqrtSafe2 (-100.0) 400.0
Nothing
> addSqrtSafe2 100.0 400.0
Just 30.0
```

```
>

> addOneSafe (Just 10.0) 20.0
Just 100.0
>
> addOneSafe Nothing 20.0
Nothing
>

> addOneSafe (parseDouble "100.0") 20.0
Just 460.0
>

> addOneSafe (parseDouble "1dfd00.0") 20.0
Nothing
>
```

3. Maybe / Option Monad in Ocaml

   The maybe type is called Option in Ocaml and F#.

```
-> Some 100 ;;
- : int option = Some 100


-> None ;;
- : 'a option = None

module OptM =
  struct

    let unit x = Some x

    let bind ma f =
      match ma with
      | Some x  ->  f x
      | None    ->  None

    let (>>=) = bind
```

```
    (* Haskell fmap *)
    let map f ma =
      match ma with
      | Some x -> Some (f x)
      | None   -> None

  end


module OptM :
  sig
    val unit : 'a -> 'a option
    val bind : 'a option -> ('a -> 'b option) -> 'b option
    val ( >>= ) : 'a option -> ('a -> 'b option) -> 'b option
    val map : ('a -> 'b) -> 'a option -> 'b option
  end


#  OptM.unit ;;
- : 'a -> 'a option = <fun>

# OptM.bind ;;
- : 'a option -> ('a -> 'b option) -> 'b option = <fun>
#

# OptM.map ;;
- : ('a -> 'b) -> 'a option -> 'b option = <fun>
#

# OptM.map (fun x -> x + 3) (Some 10) ;;
- : int option = Some 13

#  OptM.map (fun x -> x + 3) None ;;
- : int option = None

#  float_of_string "100.23" ;;
- : float = 100.23
# float_of_string "a100.23" ;;
Exception: Failure "float_of_string".
```

```
let parseFloat x =
  try Some (float_of_string x)
  with _ -> None
;;

#  parseFloat "100.00" ;;
- : float option = Some 100.
#
#  parseFloat "asds100.00" ;;
- : float option = None
#

(*

addSafe :: Maybe Double -> Maybe Double -> Maybe Double
addSafe some_x some_y = do
  x <- some_x
  y <- some_y
  return (x + y)

addSafe some_x some_y =
  some_x >>= \x ->
  some_y >>= \y ->
  return (x + y)

*)

let addSafe sx sy =
    let open OptM in
    sx >>= fun x ->
    sy >>= fun y ->
    unit (x +. y)
;;

val addSafe : float option -> float option -> float option = <fun>

# addSafe (Some 10.0) (Some 20.0) ;;
- : float option = Some 30.
# addSafe None (Some 20.0) ;;
- : float option = None
```

```
# addSafe None None ;;
- : float option = None
#


(*

If Haskell functions were impure it would work:

addInputsSafe () = do
  x <- parseDouble (readLine ())
  y <- parseDouble (readLine ())
  z <- parseDouble (readLine ())
  return (x + y + z)

addInputsSafe some_x some_y =
   readLine () >>= \x ->
   readLine () >>= \y ->
   readLine () >>= \x ->
   return (x + y + z)
*)


let prompt  message =
    print_string message ;
    parseFloat (read_line())
;;

val prompt : string -> float option = <fun>

let addInputsSafe () =
    let open OptM in
    prompt "Enter x: "  >>= fun x ->
    prompt "Enter y: "  >>= fun y ->
    prompt "Enter z: "  >>= fun z ->
    unit (print_float  (x +. y +. z))
;;

val addInputsSafe : unit -> unit option = <fun>
```

```
(* It will stop the computation if any input is invalid *)

# addInputsSafe () ;;
Enter x: 10.0
Enter y: 20.0
Enter z: 30.0
60.- : unit option = Some ()
#

# addInputsSafe () ;;
Enter x: 20.0
Enter y: dsfd
- : unit option = None
#

- :  addInputsSafe () ;;
Enter x: 20.0
Enter y: 30.0
Enter z: a20afdf
- : unit option = None

# addInputsSafe () ;;
Enter x: erew34
- : unit option = None
#
```

1.13.4 See also

Monads

- Yet Another Monad Tutorial in 15 Minutes - Carpe diem (Felix's blog)
- Haskell Monad Tutorial - The Greenhorn's Guide to becoming a Monad Cowboy
- Monads for functional programming - Philip Wadler
- Haskell/Understanding monads - Wikibooks, open books for an open world
- Chapter14.Monads - Real World Haskell

- Monads Demystified | tech guy in midtown
- In search of a Monad for system call abstractions - Taesoo Kim - MIT CSAIL
- All about Monads - Haskell Wiki
- CS 596 Functional Programming and Design Fall Semester, 2014 Doc 22 Monads & Design Patterns

List Monad

- learning Scalaz — List Monad
- Haskell/Understanding monads/List - Wikibooks, open books for an open world
- Haskell/Understanding monads/Maybe - Wikibooks, open books for an open world

Monads in Ocaml

- Lecture 21: Monads - CS 3110 Spring 2015 - Data Structures and Functional Programming

Monads in F#

- The F# Computation Expression Zoo
- Computation Expressions (F#)
- F Sharp Programming/Computation Expressions
- The F# Computation Expression Zoo (PADL'14)

Option/ Maybe Monad

- One Div Zero: Why Scala's "Option" and Haskell's "Maybe" types will save you from null
- The Option Pattern/ Code Commit
- MayBe Monad: Usage Examples - CodeProject
- Sean Voisen » A Gentle Intro to Monads . . . Maybe?
- Niwi.Nz : A little overview of error handling.
- A Monad in Practicality: First-Class Failures - Quils in Space
- Option Monad in Scala | Patrick Oscar Boykin's Personal Weblog

2 Functional Languages

Note: There is no consensus about what really is a functional language. In this table were selected programming languages which can be programmed in functional-style or favors this style.

Some Functional programming languages:

Language	Evaluation	Typing	Type Inference	Pattern Matching	Syntax Sug
Haskell	Lazy	Static	Yes	Yes	Yes
Ocaml	Strict	Static	Yes	Yes	Yes
F# (F sharp)	Strict	Static	Yes	Yes	Yes
Scheme	Strict	Dynamic	No	No	Yes/ Macro
Clojure	Strict + Lazy	Dynamic	No	Destructuring and macros	Yes/ Macro
Scala	Strict	Static	Yes	Yes	Yes
Erlang	Strict	Dynamic	?	Yes	Yes
JavaScript	Strict	Dynamic	No	No	No
R	Strict	Dynamic	No	No	No
Mathematica	Strict	Dynamic	Yes	?	?

Notes:

- AGDT - Algebraic Data Types
- GIL - Global Interpreter Locking. Languages with GIL cannot take advantage of multi-core processors.
- TCO - Tail Call Optimization. Languages without TCO cannot perform recursion safely. It can lead to a stack overflow for a big number of iterations.
- JVM - Java Virtual Machine / Java Platform
- .NET - Dot Net Platform: CLR - Virtual Machine
- NAT - Native Code. Native code is not portable like a virtual machine, its execution is constrained to the processor architecture and to the system calls of the operating system.
- VM - Virtual Machine
- OO - Object Orientated

- Currying - Curried functions like in Haskell, F# and Ocaml
- DSL - Domain Specific Language
- Syntax Sugars help increase expressiveness and to write shorter, concise and more readable code.
  - Short lambda functions:
    - Haskell: (\ x y -> 3 * x + y)
    - Ocaml and F# (fun x y -> x + y)
  - Monad bind operator: »= from Haskell
  - Function application.
    - The operator (|>) from F# that pipes an argument into a function. 10 |> sin |> exp |> cos is equivalent to: (sin (exp (cos 10)))
    - Clojure (->) macro: (-> 10 Math/sin Math/exp Math/cos) which is expanded to: (Math/cos (Math/exp (Math/sin 10)))
  - Function composition operator: (») from F# and (.) dot from Haskell
- JavaScript: Is single-thread with event loop and uses asynchronous IO (non blocking IO) with callbacks.
- It is controversial that Javascript is based on scheme. According to Douglas Crockford JavaScript is Scheme on a C clothe. With a C-like syntax {Reference}.

More Information: Comparison of Functional Programming Languages
See also: ML Dialects and Haskell: SML, OCaml, F#, Haskell

3 Influential People

A selection of people who influenced functional programming:

- Alonzo Church, Mathematician -> Lambda Calculus
- Haskell Curry, Mathematician -> Concept of currying
- Robin Milner, Computer Scientist -> Type inference

  - Hindley–Milner type system
  - ML language

- John McCarthy, Computer Scientist -> Creator of Lisp and the father of Artificial intelligence research.

  - Guy Steele Interviews John McCarthy, Father of Lisp

- John Backus, Computer Scientist -> Backus-Naur form (BNF), Fortran Language,

  - Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

- Philip Wadler, Theory behind functional programming and the use of monads in functional programming, the design of the purely functional language Haskell.

  - The essence of functional programing
  - Philip Wadler on Functional Programming - Interview

- Eugenio Moggi, Professor of computer science at the University of Genoa, Italy. - He first described the general use of monads to structure programs.

  - Notions of computation and monads - Eugenio Moggi

- Simon Peyton Jones, Computer Scientist -> Major contributor to the design of the Haskell programming language.

  - Interview with Peyton Jones - The A-Z of Programming Languages: Haskell - Techworld
  - Simon Peyton Jones - Microsoft Research

- John Hughes), Computer Scientist -> One of the most influentials papers in FP field: Why functional programing matters.

- Gerald Jay Sussman, Mathematician and Computer Scientist

  - Scheme Lisp) Language

- Book: Structure and Interpretation of Computer Programs
- Book: Structure and Interpretation of Classical Mechanics
- Lambda Papers: A series of MIT AI Memos published between 1975 and 1980, developing the Scheme programming language and a number of influential concepts in programming language design and implementation.

4 Miscellaneous

4.1 Selected Wikipedia Articles

General

- List of functional programming topics
- Comparison of Functional Programming Languages
- Functional programming
- Declarative programming
- Aspect-oriented programming

Functions
First Class Functions

- First-class function
- Pure function
- Side effect (computer science)
- Purely functional
- Referential transparency (computer science)
- Function type
- Arity
- Variadic function

Composition

- Function composition (computer science)
- Function composition - Mathematics
- Composability
- Functional decomposition

Scope

- Scope (computer science)

Currying and Partial Evaluation

- Currying
- Partial evaluation

Higher Order Functions, Closures, Anonymous Functions

- Anonymous function
- Closure (computer programming)
- Higher-order function
- Fixed-point combinator
- Defunctionalization
- Closure (computer programming))
- Callback (computer programming))
- Coroutine

Recursion

- Recursion (computer science)
- Tail call
- Double recursion
- Primitive recursive function

- Ackermann function
- Tak (function)

Lambda Calculus and Process Calculus

- Lambda calculus
- Typed lambda calculus
- Process calculus
- Futures and promises
- Combinatory logic

Evaluation

- Evaluation strategy
- Eager Evaluation
- Short-circuit evaluation

Related to Lazy Evaluation

- Lazy Evaluation
- Thunk

Monads

- Monads Functional Programming)
- Haskell/Understanding monads
- Monad transformer

Continuations

- Continuation
- Continuation-passing style

Fundamental Data Structures

- List (abstract data type)
- Array data structure
- Array data type

Types

- Category theory
- Type Theory
- Type System
- Algebraic data type
- Type signature
- Enumerated type
- Product type
- Tagged union
- Dependent type
- Recursive data type
- Generalized algebraic data type
- Disjoint union

Concurrency

- Thread (computing)
- Concurrency (computer science)
- Concurrent computing
- Actor model
- Event loop
- Channel (programming)
- MapReduce

- Futures and promises
- Asynchronous I/O
- Multicore processor

Miscellaneous

- Call stack
- Call graph
- Reflection (computer programming)
- Function object
- Memoization
- Garbage collection (computer science)

Functional Languages

- Lisp (programming language)
- Scheme Lisp
- Haskell
- ML (programming language)
- Standard ML
- OCaml
- F# - Fsharp

4.2 Selected Rosettacode Pages

4.2.1 Concepts Examples

- Call a function
- Higher-order functions
- Closures/Value capture
- Function composition

- Partial function application
- Currying
- Catamorphism - Fold/Reduce
- Null object
- Y combinator

Recursion:

- Anonymous recursion
- Ackermann function

4.2.2 Languages

- Haskell
- Ocaml
- F# - Fsharp
- Scheme
- Racket
- Clojure
- Scala
- JavaScript / ECMAScript

4.3 Libraries and Frameworks

Python

- Python Libraries Useful for functional programming:
  - functools — Higher-order functions and operations on callable objects
  - itertools — Functions creating iterators for efficient looping
  - operator — Standard operators as functions

- Functional Programming

Javascript

- Underscore.js: "Underscore is a JavaScript library that provides a whole mess of useful functional programming helpers without extending any built-in objects."