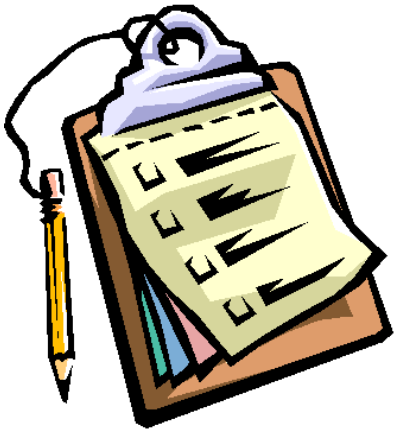
046273 Distributed Functional Programming

Lecture 1: Welcome to Erlang

Hovav Gazit, Winter 2017

Based on slides of Prof' Idit Keidar

Today's Agenda



- A little about this course
  - Motivation
  - Formalities
- Getting started with Erlang
  - What is Erlang
  - "Hello World" in Erlang
  - The Erlang shell
  - Types: numbers, atoms, booleans, tuples, lists
  - Variables & pattern matching
  - Simple functions & recursion

What is This Course About?

1. The Erlang approach to concurrency
   - Functional programming
   - Many lightweight processes
   - Asynchronous message-passing
     - No shared memory
   - Distributed systems
   - Fault-tolerance, robustness
2. The principles behind these features

Let's start with a little motivation …

The (Computing) World is Concurrent

- The multi-core revolution ⇒ computing with multiple concurrent processes/threads - parallel computing
- Wide dispersion, scalability, fault tolerance ⇒ distributed computing
- Inter-operability, interfacing with internal and external systems ⇒ middleware for distributed communication & coordination

The World is Concurrent

"The world is parallel. If we want to write programs that behave as other objects behave in the real world, then these programs will have a concurrent structure."

-- Joe Armstrong

To be continued …

Dealing with Concurrency

- Many tools, languages, paradigms,…
- For parallel computing – most commonly using shared memory
  - pthreads (see SOS), transactional memory, …
  - error prone, hard to get right
- For distributed computing – usually using message-passing
  - sockets, message queues, JMS
  - inter-operability using middleware
  - reliability always a challenge

Dealing with Concurrency Take II

"…

Use a language that was designed for writing concurrent applications, and development becomes a lot easier. Erlang programs model how we think and interact."

-- Joe Armstrong, one of the creators of Erlang

Course Staff

- Lecturer (& TA): Hovav Gazit
  - hovav@technion.ac.il
  - Room 1009 (Inside CGM lab, Room 1011)
  - Appointment by email
  - Phone: 829-5741
- Homework Checker: Itay Tsabary

- Academic Supervisor: Prof' Idit Keidar
  - idish@ee.technion.ac.il
- Course web site: Moodle

Requirements and Grading

- Erlang programming assignments 25%
  - 3 - 4 Wet Exercises
- Exam 75%
  - Allowed material: lecture notes, and your handwritten notes *only*
  - Moed A: 2/2 (Thursday)
  - Moed B: 1/3 (Wednesday)

- In order to pass the course you need to pass the EXAM

Prerequisite Background

- Some programming experience
  - e.g., in C/C++/Java
  - be comfortable reading reference manuals
- Some background in algorithms and data structures highly recommended
  - feel comfortable with recursion
- MAMAT & Operating Systems (In parallel)
- Have your heard about other approaches to concurrency?
  - OS - pthreads

Resources

- Textbooks:
  - Programming Erlang: Software for a Concurrent World, Joe Armstrong
  - I use code examples from the book http://pragmaticprogrammer.com/titles/jaerlang/code.html.
  - Erlang Programming - A Concurrent Approach to Software Development, Francesco Cesarini, Simon Thompson O'Reilly Media 2009

- Erlang official web site http://www.erlang.org/
  - Download Erlang, Reference Manual
- Tutorial (I use pictures, examples): http://learnyousomeerlang.com/contents

What is Erlang?

A programming language for concurrent software

Agner Krarup Erlang (1878-1929)
Danish mathematician invented fields of traffic engineering and queuing theory

Ericsson Language

Erlang Language Characteristics

- Functional language
  - High-level, compact declarative programming
  - Supports high-order functions
  - No for, while loops – use recursion instead
  - Immutable state – use tail recursion instead
- Garbage-collected
  - No explicit memory management
- Runs on virtual machine
  - Source compiled into bytecode

Erlang Concurrency Features

- Scalable, lightweight concurrency
- Based on message passing
  - no shared memory
- Soft real-time
- Robust, fault-tolerant
- Supports distributed computations
- Open, integrates with other platforms

Erlang is Best For

- "If your target system is a
  - high-level, concurrent, robust, soft real-time system
  - that will scale in line with demand,
  - make full use of multi-core processors, and
  - integrate with components written in other languages
- Erlang should be your choice."

-- **Cesarini and Thompson**

Erlang is Used By

- Amazon SimpleDB, DB part of EC2
- Yahoo! Delicious social bookmarking
  - > 5 million users and 150 million bookmarked URLs
- Facebook chat service backend,
  - > 100 million active users
- T-Mobile SMS & authentication
- Motorola call processing in public-safety industry
- Ericsson GPRS and 3G mobile networks worldwide
- WhatsApp "How do you support 450 million users with only 32 engineers ?"
- Many open source projects

5 Reasons to Learn Erlang by Joe Armstrong

1. You want to write programs that run faster when you run them on a multi-core computer.
2. You want to write fault-tolerant applications that can be modified without taking them out of service.
3. You've heard about "functional programming" and you're wondering whether the techniques really work.
4. You want to use a language that has been battle tested in real large-scale industrial products that has great libraries and an active user community.
5. You don't want to wear your fingers out by typing lots of lines of code.

6. It is fun!

Hello World in Erlang

```
-module(world).
-export([hello/0]).

% Say hello to the world
hello() -> io:format("Hello world!~n").
```

Module name

"Public" function

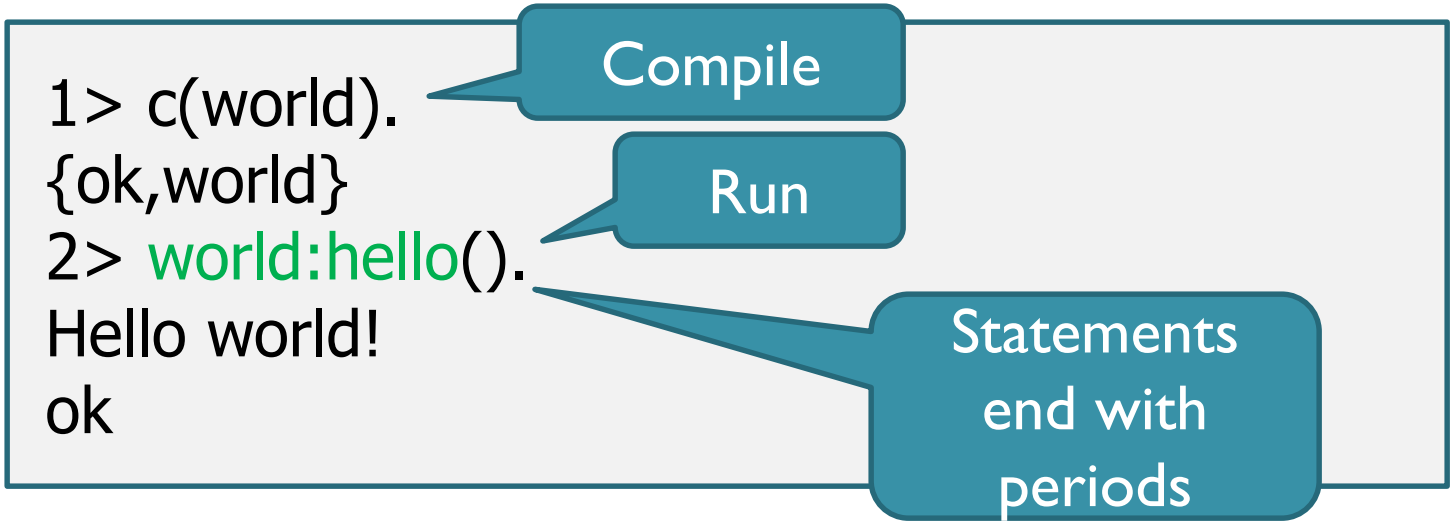
Comment

Statements end with periods

- Erlang code consists of modules
- This module's name is 'world'
  - must be stored in file 'world.erl'

The Erlang Shell

- See erlang.org for installing and running
- Let's compile and run our new program:



We'll now learn about Erlang types and variables, and evaluate some expressions in the shell

Erlang Basic Types, Variables, Pattern Matching

Chapter 2 of Francesco Cesarini

Chapter 2 of Armstrong

Numbers

- Integers, can be *any* size
  - e.g., 18446744073709551616 (2^{64}), -100
- Floats: double-precision
  - e.g., 6.148914691236517e18
- Evaluating arithmetic expressions in shell:

```
1> -234 + 1.
-233
2> 5-
2> 4.
1
3> 2#111.
7
4> (12+3)/5 + 7*2.
17.0
```

Erlang evaluates the expression only when you enter a period

111 in base 2

The result is a float

Arithmetic Expressions

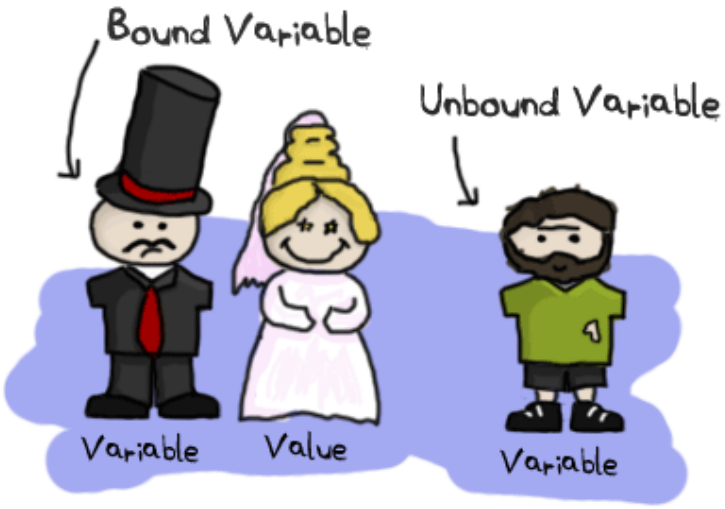
Op	*Description*	*Argument Type*	*Priority*
+ *X*	+ X	Number	1
- *X*	- X	Number	1
X * *Y*	X * Y	Number	2
X / *Y*	X / Y (floating-point division)	Number	2
bnot X	Bitwise not of X	Integer	2
X div Y	Integer division of X and Y	Integer	2
X rem Y	Integer remainder of X divided by Y	Integer	2
X band Y	Bitwise and of X and Y	Integer	2
X + *Y*	X + Y	Number	3
X - *Y*	X - Y	Number	3
X bor Y	Bitwise or of X and Y	Integer	3
X bxor Y	Bitwise xor of X and Y	Integer	3
X bsl N	Arithmetic bitshift left of X by N bits	Integer	3
X bsr N	Bitshift right of X by N bits	Integer	3

Operators with equal priorities are left associative - evaluated from left to right.

Variables Don't Vary

- Variables begin with an upper case letter (or _), include letters, digits, _
- Erlang is a single-assignment language

```
1>  X = 500.
500
2> X.
500
3> Y.
* 1: variable 'Y' is unbound
4> Y= X*X.
250000
5> X = 400.
** exception error: no match of right hand side value 400
```



Think of = in Algebraic Equations

- $X^2 = X + 2$
  - This must be the same X on both sides
- $X = X + 1$
  - This is impossible!
- $\begin{cases} X = 500 \\ Y = X^2 \end{cases}$
  - Y has to stay the square of X

Pattern Matching

- = is *not* an assignment operator
- It's a pattern matching operator:

Pattern = Term

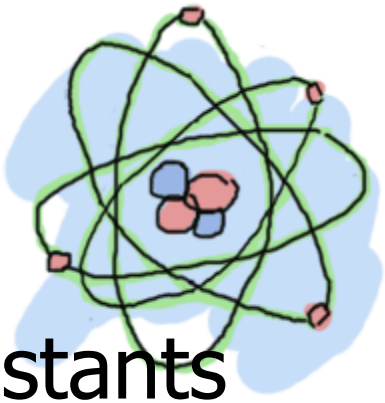
```
1>  X = 5.
5
2> Y = 2 +3.
5
3> Y= X.
5
4> X= Y+1.
** exception error: no match of right hand side value 6
5> f(X).
ok
6> X= Y+1.
6
```

Can include unbound variables - will become bound

No unbound variables

Forget X

Atoms



- Atoms are non-numerical constants
  - Like enums, or #defined symbols in C
- Start with lower case letter, include letters, digits, @, _, .
  - examples: january, start_with_lower_case, erlang@ee.technion.ac.il
- Or anything inside quotes
  - examples: ‘January’ ‘spaces allowed’

Booleans

- The atoms true and false.
- Boolean expressions:

```
1>  1 == 2.
false
2> 1 =< 2.
true
3> 1.0 == 1.
true
4> 1.0 =:= 1.
false
5> a < z.
true
6> (1<3) and (3 <4).
true
```

== means equals
=:= means is identical to

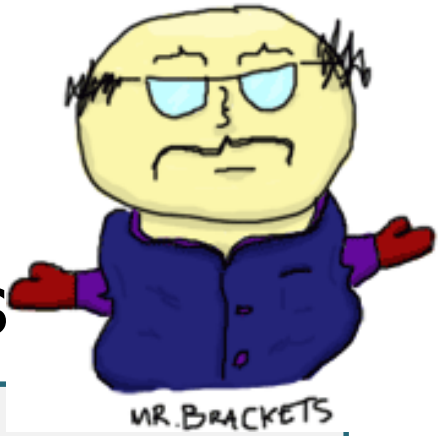
Atoms compared in lexicographical order

Boolean Expressions and Term Comparisons

- not B1: Logical not
- B1 and B2: Logical and
- B1 or B2: Logical or
- B1 xor B2: Logical xor

Operator	*Meaning*
X > Y	X is greater than Y.
X < Y	X is less than Y.
X =< Y	X is equal to or less than Y.
X >= Y	X is greater than or equal to Y.
X == Y	X is equal to Y.
X /= Y	X is not equal to Y.
X =:= Y	X is identical to Y.
X =/= Y	X is not identical to Y.

Tuples



- Like struct, but anonymous

```
1> F = {firstName, joe}.
{firstName,joe}
2> L = {lastName, armstrong}.
{lastName,armstrong}
3> P = {person, F, L}.
{person,{firstName,joe},{lastName,armstrong}}
4> tuple_size(P).
3
5 > element(2, P).
{firstName,joe}
6> setelement(2, P, {firstname, joey}).
{person,{firstname,joey},{lastName,armstrong}}
```

Built-in functions (BIFs) manipulating tuples

Pattern Matching with Tuples

- We can use pattern matching to extract values from tuples

```
1> Point = {point, 7.3, 45.2}.
{point, 7.3, 45.2}.
2> {point, X, Y} = Point.
{point,7.3,45.2}
3> X.
7.3
4> {point, C, C} = Point.
** exception error: no match of right hand side value
{point,7.3,45.2}
```

Anonymous Variables in Patterns

- _ is an anonymous variable
- Unlike regular variables, several occurrences of _ in the same pattern don't have to bind to the same value

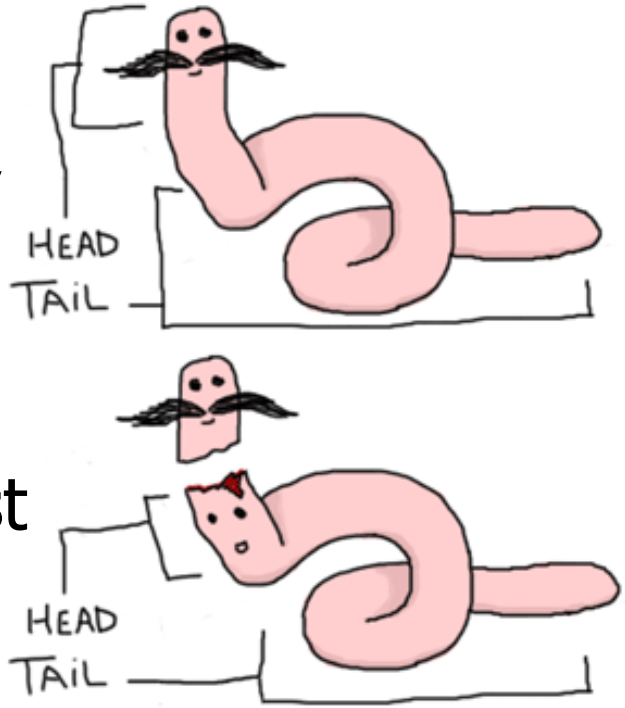
```
1> Guy ={person,{name,{first,joe},{last,armstrong}},{foot,42}}.
{person,{name,{first,joe},{last,armstrong}},{foot,42}}
2> {_,{_,{_,Who},_},_} = Guy.
{person,{name,{first,joe},{last,armstrong}},{foot,42}}
3> Who.
joe
```

Uses of Pattern Matching

- Assigning values to variables
  - see "Variables ..." slide
- Extracting values from data structures
  - see last two slides
- Controlling execution flow
  - later – see "Functions with Multiple Clauses"

Lists

- Lists look a lot like tuples
  - but are processed differently
- Example: [a,b,c,d,177]
  - a is the *head* of the list,
  - [b,c,d,177] is its *tail*
- The tail of a well-formed list
  - is a list
- [] is the empty list
- List *constructor* (*cons* for short):
  - [H|T] constructs a list with head H and tail T
  - e.g., [1,2,3] = [1|[2,3]] = [1,2|[3]] = [1,2|[3|[]]]



List Concatenation, Subtraction

```
1> Part1 = [1,2,3],
1> Part2 = [4,5,6],
1> List = Part1 ++ Part2.
[1,2,3,4,5,6]
2> [a, a, b, b, c, c] -- [a, b, c].
[a,b,c]
3> [a, a, b, b, c, c] -- [a, b, c, b].
can you guess?
```

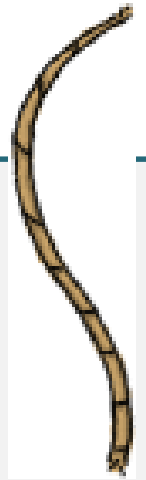
Multiple expressions separated by commas

List concatenation

List subtraction

- ++ and -- are right associative
- Note: these operators are inefficient
  - cons is more efficient
  - no need to worry about it in this course ☺

Strings are Lists



```
1> Name = "Hello".
"Hello"
2> [1,2,3].
[1,2,3]
3> [83,117,114,112,114,105,115,101].
"Surprise"
4> [1,83,117,114,112,114,105,115,101].
[1,83,117,114,112,114,105,115,101].
5> I = $s.
115
6> [I-32,$u,$r,$p,$r,$i,$s,$e].
"Surprise"
7> "A long string"
7> " is split".
"A long string is split"
```

Pattern Matching

Pattern	Term	Result?
{X,abc}	= {123,abc}	Succeeds, X → 123
{X,Y,Z}	= {222,def,"cat"}	
{X,Y}	= {333,ghi,"cat"}	
X	= true	
{X,Y,X}	= {{abc,12},42,{abc,12}}	
{X,Y,X}	= {{abc,12},42,true}	
[H|T]	= [1,2,3,4,5]	
[H,T]	= [1,2,3,4,5]	
[H|T]	= "cat"	
[A,B,C|T]	= [a,b,c,d,e,f]	

Call f() after each line, so all variables are unbound.

Erlang Functions

Chapters 2, 3 of Francesco Cesarini

Chapter 3 of Armstrong

Defining Functions

- Defined in modules (not in shell)
- Uniquely defined by
  - module name, function name, and arity
  - arity = number of arguments
- Argument, return types not specified
  - dynamic typing
  - checked at run-time

Hello Function Example

- Let's recall our example:

```
hello() -> io:format("Hello world!~n").
```

- This function is defined using 1 clause
- The head includes:
  - 'hello' – the function name (an Atom)
  - '()' –there are no arguments, the *arity* is 0
- After -> comes the body
  - it invokes the function 'format/1' (arity 1) from the module 'io'

Function Example 2

```
-module(demo).
-export([print_square/1]).
print_square(X) ->
   Sq = X * X,
   io:format("The square of ~p is ~p.~n", [X, Sq] ).
```

The arity of print_square is 1

Expressions are separated by commas

The 2nd argument of format/2 is a list

Last expression ends with period

```
1> c(demo).
{ok,demo}
2> demo:print_square(7).
The square of 7 is 49.
ok
3>demo:print_square(1.5).
The square of 1.5 is 2.25.
```

Functions with Multiple Clauses

```
-module(geometry).
-export([area/1]).
area({rectangle, Width, Height}) ->
        Width * Height;
area({square, Edge}) ->
        Edge * Edge;
area({circle, R}) ->
        math:pi() * R * R.
```

The arity of area is 1 as it takes a single tuple

Pattern matching : find first matching clause

Clauses separated by semicolons ;

Last clause ends with period

```
1> geometry:area({circle, 1.4}).
6.157521601035994
2> geometry:area(1.4).
guess what?
```

How Would We Write This in C?

```
enum ShapeType { Rectangle, Circle, Square };
struct Shape {
    enum ShapeType kind;
    union {
        struct { int width, height; } rectangleData;
        struct { int radius; } circleData;
        struct { int side;} squareData;
    } shapeData;
};
double area(struct Shape* s) {
    if( s->kind == Rectangle ) {
        int width, ht;
        width = s->shapeData.rectangleData.width;
        ht = s->shapeData.rectangleData.ht;
        return width * ht;
} else if ( s->kind == Circle ) {
...
```

Recursion


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

Start off by defining base case(s)

Other cases should converge to base case(s)

A Run of Factorial

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

factorial (3) ⇒ 3 * factorial(3-1)

factorial (2) ⇒ 2 * factorial(2 -1)

factorial (1) ⇒ 1 * factorial(1-1)

factorial (0) ⇒1

⇒1 * 1 = 1

⇒2 * 1 = 2

⇒3 * 2 = 6

The first clause matches

Recursion Over Lists

```
sum([]) -> 0;
sum([Head | Tail ]) -> Head + sum(Tail).
```

Base case

Converge to base case

What is the arity of sum?

sum([1,2,3]) ⇒ 1 + sum([2, 3])
sum([2,3]) ⇒ 2 + sum([3])
sum([3]) ⇒ 3 + sum([])
sum([]) ⇒ 0
3 + 0 ⇒ 3
2 + 3 ⇒ 5
1+ 5 ⇒ 6

Checking Membership in A List

```
member(_, [ ])        -> false;
member(H, [H | _]) -> true;
member(X, [_ | T ]) -> member(X, T).
```

What is the arity of member?

What would happen if we swapped the first two lines?

How about the last two?

Recursively Building Lists

```
bump([]) -> [];
bump([H | T]) -> [H+1 | bump(T)].
```

bump([2,4,6]) ⇒ [3 | bump ([4,6])]
bump([4,6]) ⇒ [5 | bump([6])]
bump([6]) ⇒ [7 | bump([])]
bump([]) ⇒ []
[7 | []] ⇒ [7]
[5 | [7]] ⇒ [5, 7]
[3 | [5, 7]] ⇒ [3, 5, 7]

Some List Functions

```
1> lists:max([1,2,3, 4, 3]).
4
2> lists:reverse ([1,2,3]).
[3,2,1]
3> lists:split(2, [3,4,10,7,9]).
{[3,4],[10,7,9]}
4> lists:zip([1,2,3],[5,6,7]).
[{1,5},{2,6},{3,7}]
5> lists:delete(2, [1,2,3,4,2,4,2]).
[1,3,4,2,4,2]
6> lists:last([1,2,3,4,2,4,24]).
24
7> lists:length([1,2,3,4,2,4,24]).
** exception error: undefined function lists:length/1
8> length([1,2,3,4,2,4,24]).
7
```

length/1 is a BIF

Homework for Next Week

- Download and install Erlang
- Make sure you understand the results of all examples in the Pop Quiz
  - check them in the shell!
- Open the Erlang Reference Manual
  - downloadable with the code
- Read the man page of the 'lists' module
  - experiment with some list functions
- Solve the problems on the next slide

Problems for Next Week

1. Write a module 'fib' exporting 'fib/1'

   fib(N) ->

   the Nth element in the Fibonacci series

2. Write mymath:power(x, y) ->

   x to the power of y for integers

3. Write mylists:nth(Nth, List) ->

   the Nth element in List

Not for submission, but I'll ask about them next week ☺

Next Week

- Sequential Erlang programming
  - control flow – guards, case, if
  - looping with tail recursion
  - list comprehension
  - high order functions, funs