Erlang lépésről lépésre

Nyírő Balázs
diogenesz@pergamen.hu
https://balazshobbies.wordpress.com/

Utolsó frissítés: 2018. június 11.

Tartalomjegyzék

1. Bevezető **3**
1.1. Mi a funkcionális programozás? . . . 3
1.2. Könyvek, melyekből tanultam, és amelyek példáin az én példáim is alapulnak . . . 3
1.3. Tudnivalók az Erlang nyelvről . . . 3
1.3.1. Erlang kód fordítása . . . 4
1.3.2. Mintaillesztés . . . 5
1.3.3. Stringek . . . 6
1.3.4. Az atomok . . . 6
1.3.5. A tuple adattípus . . . 6
1.3.6. A lista adattípus . . . 6
1.3.7. Műveletek Stringekkel . . . 8
1.3.8. Komplex adatszerkezet kiírása fájlba . . . 8
1.4. Listagenerátorok . . . 8
1.4.1. Listagenerátor, csak feltétellel . . . 8
1.4.2. Szűrés listagenerátorbel mintaillesztéssel . . . 9
1.4.3. Ha muvelet() sikeres, es nem ures listat ad vissza, akkor kerul bele az eredmenybe. . . . 9
1.4.4. Listabol tuple elmentese es visszaadasa . . . 9
1.5. Tail Recursion . . . 9
1.6. Debuggolás . . . 9
1.7. Metaprogramozás . . . 10
1.8. Példaprogramok . . . 10
1.8.1. Leap year - szökőév kiszámítása . . . 11
1.8.2. Collatz Conjecture - . . . 11
1.8.3. Chessboard, grains . . . 12
1.8.4. Tökéletes számok . . . 12
1.8.5. Twice . . . 13
1.8.6. Ordset implementation . . . 13
1.8.7. ordset:new/0 . . . 14
1.8.8. ordset:is_element/2 . . . 14
1.8.9. ordset:del_element/2 . . . 14
1.8.10. ordset:union/2 . . . 14
1.8.11. ordset:intersection/2 . . . 14
1.8.12. ordset:is_subset/2 . . . 15
1.9. N kiralyno problema . . . 15
1.10. Qsort, fun expressions example: . . . 16
1.11. Processzek . . . 16
1.11.1. Egy nagyon egyszerű processz indítása . . . 17

1.11.2. record to map . . . 17
1.11.3. Map to record . . . 17
1.11.4. Convert a map's binary keys to atoms . . . 18
1.11.5. Egy processz váz, inicializálással, leállással . . . 18
1.12. Nyitott kérdések . . . 19
1.12.1. Erlang applikáció újraindítása, ha a node meghal. . . . 19
1.12.2. Az Erlang kód path bővítése dinamikusan . . . 19
1.12.3. Típusinformáció . . . 20
1.12.4. -include direktiva . . . 20
1.12.5. Tesztelés . . . 20
1.12.6. Ubuntura telepített Erlang library, ahol a gyári függvények is találhatóak: . . . 20
1.13. Rebar 3 . . . 20

1. Bevezető

1.1. Mi a funkcionális programozás?

"... függvény egy kód, amely társítja az inputot az outputhoz, és ezen túl más hatása nincs. „Itt a függvény matematikai definícióját használjuk. Ezeket néha „tiszta" függvénynek nevezik, hogy megkülönböztessék – mondjuk – a C-beli függvényektől." Egy funkcionális nyelv pusztán függvényekkel végzi a programozást, ezáltal egyszerűvé és hatékonnyá téve azt..."
— Odersky

"Széles körben elfogadott tény, hogy a mindenütt jelen lévő állapot és a mellékhatások a programokban lévő komplexitás és bugok elsődleges forrásai"
— Hickey

1.2. Könyvek, melyekből tanultam, és amelyek példáin az én példáim is alapulnak

- learnyousomeerlang.com

1.3. Tudnivalók az Erlang nyelvről

- Változónevek: nagy kezdőbetűvel, vagy alsóvonással kezdődnek.
- Változónak egyszer lehet értéket adni.
- % jellel lehet a sor hátralévő részét kommentelni. Többsoros komment nincs.
- Több utasítás vesszővel elválasztva sorolható fel.
- A függvény visszatérési értéke a legutolsó kifejezés értéke. Nincs külön return utasítás.
- A függvénynevek kisbetűvel kezdődnek.
- Értéket kapott változók listázása shellben: b()
- Változó érték kötés megszüntetése shellben: f(Valtozonev)
- Összes változó kötés megszüntetés shellben: f()
- BIF, Built In Functions: Nem Erlangban, hanem rendszerint C-ben megírt függvények.

Terminálban függvényobjektum definiálása – `Duplaz()` függvény később már hívható is. `Duplaz` egy függvény objektum lesz, más nyelvek névtelen/lambda függvényeihez hasonló

```
Duplaz = fun(Szam) -> 2*Szam end.
#Fun<erl_eval.6.99386804>
Duplaz(8).
16
```

A programok `modulnev.erl` fajlokban modulokba szervezhetők. Meg kell adjuk a modul nevét, és a külső kódok számára látható függvények listáját a paraméterszámukkal együtt (aritás).

```
-module(modulnev).
-export([function1/1, functionAnother/2]).
```

1.3.1. Erlang kód fordítása

Kódot lehet az `erlc` parancs használatával fordítani, vagy a terminálban a `c()` függvénnyel: ha a modul.erl fájlban lévő kódot szeretnénk fordítani, a `c(modul)` utasítást kell kiadnunk. Ekkor egy `modul.beam` fájl keletkezik a forráskód mellett. A lefordított fájl azért hasznos, mert a shellből kilépve, majd újból belépve is használhatóak a lementett függvények.

Lehetséges olyan escript-et használó shell fájlt készíteni, amit a `chmod +x file.erl` paranccsal futtathatóvá teszünk, és azonnal használható, nem kell külön fordítani:

```
#!/usr/bin/env escript
%% -*- erlang -*-
%%! -smp enable -sname m_factorial -mnesia debug verbose
main([String]) ->
    try
        N = list_to_integer(String),
        F = fac(N),
        io:format("factorial␣~w␣=␣~w\n", [N,F])
    catch
        _:_ ->
            usage()
    end;
main(_) ->
    usage().

usage() ->
    io:format("usage:␣m_factorial␣integer\n"),
    halt(1).

fac(0) -> 1;
fac(N) -> N * fac(N-1).
```

Készítsük el a `modul_szamol` modult, egy példafüggvénnyel, amit aztán erlang shellben betöltve használunk:

```
-module(modul_szamol).
-export([osszead/2]).

osszead(A, B) -> A + B.
```

Az Erlang shellben ezt írjuk be:

```
2> c(modul_szamol).
{ok,modul_szamol}
3> ls().
.idea                    .m_factorial.erl.swp     m_factorial.erl
modul_szamol.beam        modul_szamol.erl         testerlangide.erl

ok
```

```
4>
```

Ha egy új shellt indítunk, a lefordított modul azonnal használható lesz.

```
Eshell V9.0.4  (abort with ^G)
1> modul_szamol:osszead(4,3).
7
```

Érdekes kérdés: mi történik, ha átírjuk a modult és újrafordítjuk? ha egy shellből használunk egy modulbeli függvényt, akkor az átírt változat fog futni, vagy a régi? Módosítottam és újrafordítottam az előbbi modult...

```
osszead(A, B) -> A + B + 2.
```

de a már megnyitott terminálban a régi kód futott.

Viszont a módosított modult egyszerűen újra betölthetjük (2-vel módosítottam az előbb az eredményt):

```
3> modul_szamol:osszead(4,3).
7
4> l(modul_szamol).
{module,modul_szamol}
5> modul_szamol:osszead(4,3).
9
```

Egy modulbeli függvényre akár új néven is hivatkozhatunk a terminálban (az összeadást javítottam):

```
6> Plus = fun modul_szamol:osszead/2.
#Fun<modul_szamol.osszead.2>
7> Plus(1,2).
3
```

Az import utasítással más modul függvényét emelhetjük be az aktuális névtérbe:

```
-import(modulnev, fuggveny/parameterek_szama)
```

Ha saját modult írunk, a `-export(fuggveny/1, masikfuggveny/3)` módon exportálni kell azokat a függvényeket, amelyek kívülről látszódnak. A függvény neve mögött per jellel a paraméterek számát soroljuk fel (aritás). Nem szép, de lehetőség van egy modulból az összes függvényt automatikusan exportálni a `-compile(export_all)` direktívával.

A programban kommenteket a % jellel lehet készíteni. Ha egymás mögött két százalékjel van, a komment az őt befoglaló kóddal van egy szinten indentálva, a három százalékjel jelentése: bal oldalra igazítva. (??)

A függvények típusát is megadhatjuk:

```
-module(fv_param_tipus).
-export([dupla/1]).
-spec(dupla(number()) -> number()).

dupla(A) -> 2*A.
```

1.3.2. Mintaillesztés

Változókhoz értékek kötése mintaillesztésen keresztül történik (más nyelven értékadásnak hívnánk). Az = jel itt nem értékadást jelent, hanem az = jel bal oldalán álló minta és a jobb oldali term összehasonlítását. Ha a minta szerkezete megegyezik a term szerkezetével, a nem kötött változónevekhez a term-beli párjuk értéke hozzákötődik. Ezt a kötődést értékadásnak is felfoghatjuk

Ugyan azt a változónevet többször is felhasználhatjuk a mintában, például ha egy olyan szerkezetet várunk, aminek az első és utolsó számjegye megegyezik – ez tulajdonképpen egy vizsgálat, hogy a szerkezet illeszkedik-e az adott mintára. Ha igen, az A értéket kap.

Ha egy mintaillesztésben meg kell adnunk egy változót, de azt nem akarjuk később felhasználni, például itt a belső számjegyek értékét, akkor alsóvonással jelezhetjük ezt. A sima alsóvonás az egy névtelen változó. Ha egy változónevet alsóvonással kezdünk, akkor jelezzük, hogy mi az adott érték jelentése, de az alsóvonás jelzi, hogy később nem szándékozzuk az értékét felhasználni.

A mintaillesztés azért jó, mert adott függvények adott mintájú adatszerkezetek érkezésekor futnak le.

```
{X, Y} = {1, 2}
{A, _, _EzSemKell, A} = {8, 7, 8}  %
```

A mintaillesztés lehetőségei korlátozottak, nem lehet vele számtartományra vagy bizonyos típusú adatra szűrni. Ha erre van szükség, `guard`-okat kell használni.

1.3.3. Stringek

Erlangban nincs külön string adattípus. A string az egymás utáni karakterek listáját jelenti. A listában egy karakter az az ascii kódjának megfelelő számmal van reprezentálva. Vagyis egy string az tulajdonképpen számok listája.

1.3.4. Az atomok

Az atomok kisbetűs string konstansok. A kódban idézőjelek nélkül írjuk le, alsóvonást és kukac jelet tartalmazhat. Egy atom értéke az maga a neve. (olvasnivaló) Felmerül a kérdés, hogy mi szükség van rájuk: a legtöbb programozási nyelvben stringeket használunk, ahol azok egyenlőségét `"kutya" == "kutya"` módon vizsgáljuk, és betűről betűre összehasonlítjuk a két szöveget. Az Erlang nyelvben nagyon sok string összehasonlítást végzünk a mintaillesztések miatt. Hogy ne kelljen mindig a betűről betűre összehasonlítást elvégezni, atomokat használunk, melyeknek elég csak a pointereit összehasonlítani, hogy kiderüljön: azonos értékűek-e. Ráadásul minden egyes atom csak egy példányban létezik a memóriában, míg a stringek ellenőrzésekor valószínűleg két külön példányt hasonlítunk össze. Az atomokra érdemes hasonlóképp tekinteni, mint az integer típusra számok esetében.

Fontos: véges számú atomot tud az erlang interpreter tárolni, ezért nem szabad list_to_atom() vagy más atomra alakító hívást használni, ha ez elkerülhető, olyan kódban, ahol nem tudjuk garantálni a létrejövő atomok számának felső korlátosságát. Ilyen esetben mintaillesztéssel ha ez a bemenet, akkor ez az atom keletkezik jellegű megoldást érdemes használni.

Ha nagybetűket, vagy szóközt szeretnénk használni, akkor egyszeres idézőjelek közé kell rakni:

```
'Ez␣egy␣nagybetuket␣es␣szokozt␣tartalmazo␣atom'
```

1.3.5. A tuple adattípus

A tuple elemszáma létrehozáskor rögzített. A benne lévő elemeket kiolvashatjuk, ha a tuple szerkezetének megfelelően építjük fel a változókat a bal oldalon (ugyanez listákkal is működik):

```
Coord = {1, 2}.
{X, Y} = Coord.
```

1.3.6. A lista adattípus

A lista változó számú elemet tartalmaz, üres is lehet – a lista hossza az elemek száma. A lista első eleme a `Head` (fej), a további részei a `Tail` (farok). Amikor a farokról beszélünk, az is egy lista, hiszen elemek vannak benne - csak az eredeti lista fej része nélkül.

Egy lista bármilyen típusú elemet tartalmazhat. Az alábbi példában ++ jellel L listából és a másik listából egy új lista készül.

Fontos: a következő példában az = jel azért nem működik, mert az nem értékadás. A bal oldali `L`, amihez kötve van a `[1,2,3]` lista, annak a szerkezete és értékei nem illeszkednek a jobb oldalon lévő öt elemű `[1,2,3,4,5]` listával. Egy három elemű lista szerkezete nem ugyanaz, mint egy öt elemű listáé:

```
1> L=[1,2,3].
[1,2,3]
2> L ++ [4,5].
[1,2,3,4,5]
3> L = L ++ [4,5].
** exception error: no match of right hand side value
                    [1,2,3,4,5]
4> L = L ++ [].
[1,2,3]
```

Erlangban a szövegek azok karakterkódokat tartalmazó listák, nincs külön string adattípus. Kiíratjuk ugyanazt a Txt változót, kétféleképp:

```
1> Txt = "almafa".
"almafa"
2> Txt.
"almafa"
3> io:format("~p~n", [Txt]).
"almafa"
ok
4> io:format("~w~n", [Txt]).
[97,108,109,97,102,97]
```

Új listát meglévő listákból a ++ és -- operátorokkal építhetünk fel, mindkettő jobb asszociatív:

```
5> [2, 4, 6] ++ [8, 10].
[2,4,6,8,10]
6> [2, 4, 6] -- [2, 4] -- [6].
[6]
7> [2, 4, 6] -- [2, 6] -- [4].
[4]
8> [2, 4, 6, 2, 6] -- [2, 6] -- [4].
[4,2,6]
```

A lista első elemét `Head`-nek (Fej), a többi részét `Tail`-nek (Farok) nevezzük, függvénnyel lekérdezhetők, illetve mintaillesztésnél használjuk ezt a fogalmat: L = [2, 4, 6, 8]. [2,4,6,8] hd(L). 2 tl(L). [4,6,8]

```
Feldolgoz = fun([H|T]) -> H  end.
#Fun<erl_eval.6.99386804>
16> Feldolgoz([2,4,6]).
2
```

Egy listából több elemet is kivehetünk. Ha itt `A, A` lenne, akkor két ugyanolyan elemet várnánk a lista első két elemeként.

```
KetSzam = fun([A, B| T]) -> {A,B}  end.
KetSzam([2,4,6,8]).
{2,4}

% a shellben is tudunk mintaillesztest csinalni:
% azonos-e az elso ket elem?
1> [A, A | T] = [1,1,2,3].
```

```
[1,1,2,3]

% es ha nem azonosak:
2> [A, A | T] = [1,2,2,3].
** exception error: no match of right hand side value
                    [1,2,2,3]

New = [2 | [4,6,8]].
[2,4,6,8]
```

1.3.7. Műveletek Stringekkel

```
% ket string osszefuzese:
A = "X" ++ "Y".

% Tuple atalakitasa string-e:
4> Tuple={5,43}.
{5,43}
5> TupleToString=io_lib:format("~p", [Tuple]).
[[123,["5",44,"43"],125]]
6> TupleToString.
[[123,["5",44,"43"],125]]
```

1.3.8. Komplex adatszerkezet kiírása fájlba

Forrás

```
file:write_file("/tmp/foo", io_lib:fwrite("~p.\n", [Data])).
```

1.4. Listagenerátorok

A listagenerátorokkal egy kiinduló listából adott feltételnek megfelelő elemeken műveletet végezhetünk és az eredményt listában visszaaadjuk: L listából kiválogatjuk a 3-nál nagyobb elemeket, elvégezzük mindegyiken az X*X műveletet, és az eredményeket egy listában visszaadjuk.

```
L = [1,3,5,7].
[1,3,5,7]
7> [X*X || X <- L, X > 3].
[25,49]
```

Összetettebb listagenerátor, két listából dolgozik, mindkettőre feltétellel szűr. Először veszi X-ből a 4-et, majd Y-ból az 5ös és 6-os értékeket, utána a következő körben X értéke 5, és újra a az 5, 6-os értékek Y-ból:

```
Eredmeny=[X+Y || X <- [1,2,3,4,5], Y<-[3,4,5,6],  X > 3, Y > 4].
"\t\n\n\v"
22> io:format("~w~n", [Eredmeny]).
[9,10,10,11]
ok
```

1.4.1. Listagenerátor, csak feltétellel

```
["Pling" || 6 rem 3 == 0].
```

A listákról tudni kell, hogy mivel a karakterláncok is listák, ahol a karakterek számkódja van a listában, fura feliratok jelenhetnek meg a lista megjelenítésekor. Az előző példában az `io:format()` függvény garantálta, hogy számokkal lesz a lista tartalma megjelenítve.

A c karakter számkódját a `$c` kifejezéssel kaphatjuk meg. Az ”abc” string tulajdonképpen ez a lista: `[$a, $b, $c]`

```
1> S=[$a, $b, $c].
"abc"
2> S
2> .
"abc"
3> io:format("~w~n", [S]).
[97,98,99]
ok
```

1.4.2. Szűrés listagenerátorbel mintaillesztéssel

Az Idojaras lista atomokat tartalmaz (az atomok string konstansok).

```
1> Idojaras = [{budapest,esos}, {becs,napos}, {parizs,esos},{london,napos}].
[{budapest,esos},{becs,napos},{parizs,esos},{london,napos}]
2> [Varos || {Varos, esos} <-Idojaras].
[budapest,parizs]
```

1.4.3. Ha muvelet() sikeres, es nem ures listat ad vissza, akkor kerul bele az eredmenybe.

N kiralyno problemanal jott elo. [X || Y <- lists:seq(1,8), X <-muv(Y)].

1.4.4. Listabol tuple elmentese es visszaadasa

16> FuncList=[1,2,3]. [1,2,3] 17> [EGYBE, anotherdata || EGYBE = A, B, C <- FuncList]. [1,2,3] 18>

1.5. Tail Recursion

Ha egy függvény utolsó hívása egy függvényhívás, akkor beszélhetünk ’Tail Recursion’-ról. Ebben az esetben kevesebb erőforrást igényel a rekurzió, akár végtelen ciklus is készíthető így.

1.6. Debuggolás

A debugger használatához `debug_info` opcióval kell a fordítást végrehajtani, ezt többféleképp tehetjük meg Erlang shellből::

```
c(modulnev,[debug_info])
compile:file("path/to/file.erl",[debug_info]).
```

Linux shellből egy harmadik módszer:

```
erlc +debug_info x.erl
```

A modul fordítása után kövessük a http://erlang.org/doc/apps/debugger/debugger_chapter.html lépéseit:

- `debugger:start().`

- a megjelenő monitor ablakban a Module/Interpret menüpontban válaszd ki a figyelendő modult
- Modul/Figyelendő modulnév/View menüpont hatására megjelenik a kód.
- a shellben a modulod belépő függvényét indítsd el, `modul:fv(Params)` módon

Példa debuggolásra:

1.7. Metaprogramozás

Példakód http://erlang.org/pipermail/erlang-questions/2003-November/010546.html

```
Small example:

1> FunStr = "fun␣(A)␣->␣A+B␣end.".
...
2> {ok, Tokens, _} = erl_scan:string(FunStr).
...
3> {ok, [Form]} = erl_parse:parse_exprs(Tokens).
...
4> Bindings = erl_eval:add_binding('B', 2, erl_eval:new_bindings()).
...
5> {value, Fun, _} = erl_eval:expr(Form, Bindings).
..
6> Fun(1).
3
```

1.8. Példaprogramok

Erlang programot fordíthatunk a shellben a `c(modulnev)` függvénnyel, és futtathatjuk a `modulnev:fv()` utasítással, ha a megfelelő függvény exportálva lett, és ezért kívülről hívható.

Erlang program indítható Linux héjból így is:

```
#!/usr/bin/env escript

% Start program:
% chmod +x m_factorial_cli.erl
% ./m_factorial_cli.erl 5
main([String]) ->
    N = list_to_integer(String),
    F = fac(N),
        io:format("factorial␣~w␣=␣~w\n", [N,F]).

fac(0) -> 1;
fac(N) -> N * fac(N-1).
```

Egy másik indítási lehetőség, ha a modult terminálból lefordítjuk a `erlc modul.erl` paranccsal, majd lefuttatjuk az `erl -run m_factorial_run parameteres 4` utasítással le tudunk futtatni. Az Erlang shell-ből a kilépést a `halt()` utasítás biztosítja.

Fontos: ha a `-run` kapcsolóval indítjuk az erlangot, a paraméterek stringként adódnak át, a `-s` esetében pedig atomként, egy listában. A változónév előtti aláhúzással jelezzük, hogy az adott változót szándékosan nem használjuk később. Ha egy függvényt meghívunk, több paramétert is átadhatunk neki - és egy paraméter egy lista (string) jelen esetben, vagyis a paraméterek listájában az első elemet, egy stringet akarunk használni, ami az Erlangban szintén lista.

```
-module(m_factorial_run).
-export([fac/1, pelda/0, parameteres/1]).

pelda() ->
        io:format("Linux␣hejbol␣hivott␣fuggveny\n"),
        halt().

% receive one number as a string
parameteres(_Parameters=[P1 | _P_others]) ->
        io:format("Parameter␣1:␣>~p<~n", P1),
    Num = list_to_integer(P1),
        io:format("string␣converted␣to␣num:␣~p~n", [Num]),
    Fac = fac(Num),
    io:format("factorial␣~w␣=␣~w\n", [Num,Fac]),
        halt().

fac(0) -> 1;
fac(N) -> N * fac(N-1).
```

1.8.1. Leap year - szökőév kiszámítása

```
-module(exercism_leap_year).
-export([main/1]).

main([Par1 | _P_others]) ->
    Year = list_to_integer(Par1),
    io:format("Is␣~w␣a␣leap␣year?␣␣~w\n", [Year, leap_year(Year)]),
    halt().

leap_year(Year) when Year rem 400 == 0 -> true;
leap_year(Year) when Year rem 100 == 0 -> false;
leap_year(Year) when Year rem 4 == 0 -> true;
leap_year(_) -> false.
```

1.8.2. Collatz Conjecture -

A feladat részletes leírása

```
-module(exercism_collatz_conjecture).
-export([main/1]).

main([Par1 | _]) ->
    Num = list_to_integer(Par1),
    io:format("Collatz␣conjecture␣~w:␣~w\n", [Num, calc(Num, 0)]),
    halt().

calc(N, _Level) when N < 1        -> { error , "Only␣positive␣numbers␣are␣allowed" };
```

```
calc(N, Level) when N == 1       -> Level;

calc(N, Level) when N rem 2 == 0 ->
        %io:fwrite("N: ~p  Level: ~p~n", [N, Level]),
        calc(N div 2, Level+1);

calc(N, Level) ->
        %io:fwrite("N: ~p  Level: ~p~n", [N, Level]),
        calc((3*N+1), Level+1).
```

1.8.3. Chessboard, grains

A feladat részletes leírása

A programot le kell fordítani, majd futtatni:

```
erlc exercism_grains.erl
erl -run exercism_grains main
...
Grains on chessboard: 18446744073709551615
```

A forráskód:

```
-module(exercism_grains).
-export([square/1, total/0, sum_all_element/2, main/0]).

main() ->
    io:format("Grains␣on␣chessboard:␣~w\n", [total()]),
    halt().

square(1) -> 1;
square(N) -> 2*square(N-1).

total() ->
    All64value = [square(X) || X <- lists:seq(1,64) ],
    sum_all_element(All64value, 0).

sum_all_element([H|[]], Sum) ->
    Sum + H;
sum_all_element([H|T], Sum)  ->
    sum_all_element(T, Sum+H).
```

1.8.4. Tökéletes számok

Wikipédia leírás

```
-module(nyiro_balazs_tokeletes_szamok).
-compile(export_all).

findPerfects() ->
    MaxN = 20,
    Perfects = [ calcPerfect(N) || N <-lists:seq(2, MaxN, 1), genPerfectNum(N) > 0],
    io:fwrite("MaxN:␣~w,␣Perfects:␣~w", [MaxN, Perfects]).
```

```
genPerfectNum(N) ->
    MaybeMersenne = pow(2,N)-1,
    io:fwrite("Maybe␣Mersenne:␣~w~n", [MaybeMersenne]),
    case isPrime(MaybeMersenne) of
        true -> calcPerfect;
        false-> 0
    end.

calcPerfect(N) ->
        pow(2,N-1) * ( pow(2,N)-1 ).

pow(_N, 0) -> 1;
pow(N, Level) -> N * pow(N, Level-1).

isPrime(1)->true;
isPrime(2)->true;
isPrime(3)->true;
isPrime(N)->
    Dividers = [ X || X <- lists:seq(2, ceil(N/2), 1), N rem X == 0],
    case length(Dividers) of
        0 -> true;
        _ -> false
    end.
```

1.8.5. Twice

Definiálj egy függvényt, ami egy kapott függvényt az adott értéken kétszer lefuttat.

```
-module(nyiro_balazs__elte_1028_twice).
-compile(export_all).


% Define a function twice/2 that applies its function parameter to an arbitrary value twice.
%
% For instance:
%
% 1> function:twice(fun (N) -> N + 2 end, 0).
% 4
% 2> function:twice(fun (X) -> " knock" ++ X end, ". Who's there?").
% " knock knock. Who's there?"

twice(Func, Val) -> Func(Func(Val)).
```

1.8.6. Ordset implementation

In this exercise we will implement a set data structure and define functions over it. The set will be a list which stores the elements in an increasing order and contains no duplicates.

In this exercise you may not use functions defined in ordsets, lists, and proplists.

Create the module ordset, which exports the following functions.

1.8.7. ordset:new/0

The new function returns an empty set, that is, an empty list.

1.8.8. ordset:is_element/2

The is_element/2 function expects an element and an ordset, and recursively decides whether the ordset contains the element.

Take advantage of the fact the set is sorted. Search until the actual element in the set is greater than the first argument.

For instance:

```
1> ordset:is_element(1, []).
false
2> ordset:is_element(b, [a,b,c]).
true
3> ordset:is_element(d, [a,b,c]).
false
```

1.8.9. ordset:del_element/2

This function removes an element from an ordset. If the set does not contain the element, the set remains unchanged.

For example:

```
1> ordset:del_element(4, []).
[]
2> ordset:del_element(d, [a,b,c]).
[a,b,c]
3> ordset:del_element(2, [0,2,4,5]).
[0,4,5]
```

1.8.10. ordset:union/2

The union/2 expects two ordsets as parameters, and returns the union of them. The result is also an ordset.

```
1> ordset:union([1,2,3], []).
[1,2,3]
2> ordset:union([1,3,5], [2,4,6]).
[1,2,3,4,5,6]
3> ordset:union([2], [0,1,5]).
[0,1,2,5]
```

1.8.11. ordset:intersection/2

The function expects two ordsets as arguments, and returns a new ordset containing common elements.

```
1> ordset:intersection([1,2,3],[]).
[]
2> ordset:intersection([1,2,3],[0,1,3,4]).
[1,3]
```

1.8.12. ordset:is_subset/2

The function is_subset expects two ordsets as arguments, and decides whether the second argument contains every element of the first argument.

```
1> ordset:is_subset([], [2,3,5]).
true
2> ordset:is_subset([2,3], []).
false
3> ordset:is_subset([3,5], [2,3,5,6]).
true
```

```
-module(nyiro_balazs__elte_1034_ordset).
-compile(export_all).


new()->[].

% TODO:
is_element(Wanted, [Wanted|_T])->true;
is_element(Wanted, [_|T])->is_element(Wanted, T);
is_element(_, [])->false.

add_element(E, [])->[E];
add_element(E, [H|T]) when H >  E -> [H | add_element(E, T)];
add_element(E, Stack=[H|_]) when H == E -> Stack;
add_element(E, Stack=[H|_]) when H <  E -> [E|Stack].

del_element(_E, [])->[];
del_element(E, [H|T]) when H == E -> T;
del_element(E, [H|T]) when H /= E -> [H | del_element(E, T)].

union(A, []) -> A;
union(A, [H|T]) -> union(add_element(H, A), T).

% TODO
% intersection(A, B)
% is_subset(A, B)
```

1.9. N kiralyno problema

```
% Author: Balazs Nyiro
-module(nyiro_balazs__nqueen).
-compile(export_all).

nq()->
    Yall=lists:seq(1,8),

    % only the order is the difference:
    % TODO: delete duplicated coords from solutions: [{1,1}, {2,3}], [{2,3}, {1,1}]
    Solutions1 = [sit({X,Y},[],[])          || X<-[1],Y<-Yall],
    Solutions2 = [sit({X,Y},Queens, Queens) || X<-[2],Y<-Yall, Queens<-Solutions1],
```

```
    Solutions3 = [sit({X,Y},Queens, Queens) || X<-[3],Y<-Yall, Queens<-Solutions2, length(Que
    Solutions4 = [sit({X,Y},Queens, Queens) || X<-[4],Y<-Yall, Queens<-Solutions3, length(Que
    Solutions5 = [sit({X,Y},Queens, Queens) || X<-[5],Y<-Yall, Queens<-Solutions4, length(Que
    Solutions6 = [sit({X,Y},Queens, Queens) || X<-[6],Y<-Yall, Queens<-Solutions5, length(Que
    Solutions7 = [sit({X,Y},Queens, Queens) || X<-[7],Y<-Yall, Queens<-Solutions6, length(Que
    Solutions8 = [sit({X,Y},Queens, Queens) || X<-[8],Y<-Yall, Queens<-Solutions7, length(Que

    Solutions = [S || S <-Solutions8, length(S) > 0],
    io:format("Solutions:␣~w~n",[Solutions]),
    io:format("length1:␣~w~n", [length(Solutions1)]),
    io:format("length2:␣~w~n", [length(Solutions2)]),
    io:format("length3:␣~w~n", [length(Solutions3)]),
    io:format("length4:␣~w~n", [length(Solutions4)]),
    io:format("length5:␣~w~n", [length(Solutions5)]),
    io:format("length6:␣~w~n", [length(Solutions6)]),
    io:format("length7:␣~w~n", [length(Solutions7)]),
    io:format("length8:␣~w~n", [length(Solutions8)]),
    io:format("length␣solutions:␣~w~n", [length(Solutions)]),
    io:format("END␣~n").

% Can the new Queen sit down?
% give back the Queens list
sit({X,Y},_TestQueens=[],          AllQueens) -> [{X,Y}|AllQueens];
sit({X,Y},_TestQueens=[{ QX, QY}|_],_AllQueens) when
    QX == X;
    QY == Y;
    (QX-X==QY-Y);
    (QX-X==-(QY-Y)) -> [];
sit({X,Y},_TestQueens=[{_QX,_QY}|T], AllQueens) -> sit({X,Y}, T, AllQueens).
```

[1]

1.10. Qsort, fun expressions example:

```
L=[4,89,2,267,7,78,15].

1> L=[4,89,2,267,7,78,15].
[4,89,2,267,7,78,15]
2> Sort = fun(F, [Elem|Others])->
2>   F(F, [A || A <- Others, A=<Elem]) ++
2>   [Elem] ++
2>   F(F, [B || B <- Others, B>Elem]);
2> (F, [])-> [] end.
#Fun<erl_eval.12.99386804>
3> Sort(Sort, L).
[2,4,7,15,78,89,267]
```

1.11. Processzek

[1] labjegyzet

1.11.1. Egy nagyon egyszerű processz indítása

```
1> c(processes__simple_draft).
processes__simple_draft.erl:2: Warning: export_all flag enabled - all functions will be expor
{ok,processes__simple_draft}
2> processes__simple_draft:start().
Loop...
true
3> server ! {handle, "first␣message"}.
handle msg: [102,105,114,115,116,32,109,101,115,115,97,103,101]
{handle,"first␣message"}
Loop...
```

```
-module(processes__simple_draft).
-compile(export_all).

start() ->
    register(server, spawn_link(?MODULE, loop, [])).

stop() ->
    server ! stop.

loop()->
    io:format("Loop...~n"),
    receive
        stop -> do_cleanup;
        {handle, Msg} ->
            io:format("handle␣msg:␣~w~n", [Msg]),
            loop();
        _Other ->
            an_unhandled_message
    end.
```

1.11.2. record to map

```
% this isn't general because we transform atom to binary.
% TODO: separate key conversion and create an independent record_to_map
record_to_map(Rec, Fields)->
  {NewMap, _} = lists:foldl(
    fun(Field, {M, N}) ->
       Val = element(N, Rec),
       {M#{ erlang:atom_to_binary(Field, utf8) => Val}, N + 1}
    end,
    {#{}, 2},
    Fields
  ),
  NewMap.
```

1.11.3. Map to record

```
% In a normal map, StartIndex=2 because 1=the name of the record.
```

```
% But: timestamp is in the record now, so the first new element will be the 3rd.
map_to_record(M, BaseRecord, StartIndex)->
  {NewRec, _} = lists:foldl(
    fun(Key, {Rec, Idx}) ->
      Val = maps:get(Key, M),
      RecUpdated = setelement(Idx, Rec, Val),
      ct:pal("RecUpdated␣=␣~p~n", [RecUpdated]),
      {RecUpdated, Idx + 1}
    end,
    {BaseRecord, StartIndex},
    maps:keys(M)
  ),
  NewRec.
```

1.11.4. Convert a map's binary keys to atoms

```
% the map's keys are binary elements. convert them to atoms
binmap_to_atommap(Mbin) ->
  lists:foldl(
    fun(Key, Map) ->
      Val = maps:get(Key, Mbin),
      maps:put(erlang:binary_to_atom(Key, utf8), Val, Map)
    end,
    #{},
    maps:keys(Mbin)
  ).
```

1.11.5. Egy processz váz, inicializálással, leállással

```
-module(processes__simple_skeleton).
-compile(export_all).

% call it with something as Args: start(firstcall)
start(Args) ->
    %         process_id=spawn_link(modulname, funcname, args)
    % register(ProcessName, ProcessId)
    register(server, spawn_link(?MODULE, init, [Args])).


stop() ->
    server ! stop.

init(Args)->
    io:format("init,␣Args:␣~w~n", [Args]),
    InitState = initialize_state(Args),
    loop(InitState).

% from terminal you can send messages to the server:
% server ! {handle, message}
loop(State)->
    io:format("Loop,␣State:␣~w~n", [State]),
```

```
    receive
        stop ->
            terminate(State);
        {handle, Msg} ->
            NewState = handle_req(Msg, State),
            loop(NewState);
        _Other ->
            an_unhandled_message
    end.

handle_req(Msg, State)->
    io:format("Msg:␣~w␣␣State:␣~w~n", [Msg, State]),
    State.

initialize_state(Args) ->
    io:format("initialize␣state:␣~w~n", [Args]),
    do_init.

terminate(State)->
    io:format("terminate:␣~w~n", [State]),
    do_cleanup.
```

1.12. Nyitott kérdések

Miért van általában state?A state az micsoda? allapot tarolas??

A supervisor tudja, hogy a rendszeremben lévő processzek milyen kapcsolatban vannak. Konkurrencia kezelő fogalom. Supervisort exit szignállal lehet leállítani, nincs stop() függvénye.

Application: forrásfájlokkal, adatokkal, külső függőségekkel együtt egy egész, ami az alkalmazást magába foglalja. Application/application: van main supervisor, ami inicializálja az applicationt és elindulnak processzek. A kernel modul kernel.app fájljában van `{mod, {kernel, []}}` elem, emiatt elindul a kernel nevű modul, üres paraméterrel. A `kernel-5.4.1/src/kernel.erl` fájlban a `start/2` indítja a `supervisor:start_link()`et.

Application/library: beam fájlok halmaza. Ad egy szolgáltatást, de nincs fő supervisor, utility gyűjtemény, csak függvényeket hívogatunk belőlük.

A .app fájlt, az application leíró fájlt az applikáció ebin mappájába kell másolni.

Az application fület nézzük meg:

```
observer:start()
```

Hogy mukodik a monitorozas? mi a link? mi a genserver? megnezni: global:registered_names() Cowboy, yaws, rebar3

1.12.1. Erlang applikáció újraindítása, ha a node meghal.

```
erl -heart
export HEART_COMMAND=path/bin/start
```

1.12.2. Az Erlang kód path bővítése dinamikusan

```
code:get_path().
code:add_patha("path/dir").
```

```
% from command line, you can start erlang with modified path:
erl -pa path/dir
```

plt: típusinformációt tartalmazó fájl, generálni kell

1.12.3. Típusinformáció

-spec fuggveny(map(), (number())->number())

Ne legyen a kódban export all, mert a dialyzer akkor pontosabb, mert akkor nem lehet kívülről mindent hívni.

```
dialyzer --build_plt --apps erts stdlib kernel
```

Megnezni: global:whereis_name(modulnev). percept:profile()... melyik folyamat meddig futott cprof, fprof - folyamat elemzes? coverity checker: mely sorokat erintette a vegrehajtas?

1.12.4. -include direktiva

`include(fajl.hrl)` utasítással fordításkor az adott fájlból bemásolódik annak a tartalma az include helyére. Létezik `-include_lib()` utasítás is, például a kernelnek van egy file.hrl fájlja. Ha nem akarok statikus útvonalat használni, mondhatom, hogy a kernel library adott fájlját akarom módosítani, így ha a kernelnek újabb verziója érkezik, amitől a path-ja változik, akkor is működik az include: `-include_lib(kernel/include/file.hrl)`

1.12.5. Tesztelés

Eunit: függvények egyedi tesztelése Ct, Common test: nagyobb rendszer tesztelése: bonyolultabb, nagyobb tesztek. a tesztelő függvényeket _suite.erl fájlba kell tenni.

Tulajdonság alapú tesztelés: a programkódra tulajdonságokat fogalmazol meg. Proper: opensource valtozata a quickChecknek QuickCheck: fizetős, alap nem fizetős

1.12.6. Ubuntura telepített Erlang library, ahol a gyári függvények is találhatóak:

1.13. Rebar 3

A Rebar 3 egy Make-hez hasonó szkriptgyűjtemény, amivel template-ből lehet új appokat létrehozni, buildelni, tesztelni.

Alapvető használat - új app létrehozása: `rebar3 new app uj_applikacio_neve`

```
/usr/lib/erlang/lib
```