# Az Erlang programozási nyelv

## Deklaratív programozás, 2008. őszi félév

Hanák Péter

hanak@iit.bme.hu

# Tartalom


1. Szekvenciális Erlang
   - Bevezetés
   - Erlang emulátor
   - Típusok
   - Szekvenciális programozás
   - Típusspecifikáció
   - Kivételkezelés
   - Rekord
   - Rekurzív adatstruktúrák
   - Könyvtári függvények


# Az Erlang nyelv

- 1985: megszületik „Ellemtelben” (Ericsson–Televerket labor)
- 1991: első megvalósítás, első projektek
- 1997: első OTP (Open Telecom Platform)
- 1998-tól: nyílt forráskódú, szabadon használható

- Funkcionális alapú (Functionally based)
- Párhuzamos programozást segítő (Concurrency oriented)
- Gyakorlatban használt

„Programming is fun!”

# Erlang-szakirodalom

- Joe Armstrong: Programming Erlang. Software for a Concurrent World. The Pragmatic Bookshelf, 2007. Nyomtatott és/vagy PDF:
  `http://www.pragprog.com/titles/jaerlang/programming-erlang`
- Joe Armstrong, Robert Virding, Claes Wikström, Mike Williams: Concurrent Programming in Erlang. Second Edition. Prentice Hall, 1996. ISBN 0-13-508301-X. Első része szabadon letölthető:
  `http://erlang.org/download/erlang-book-part1.pdf`
- On-line Erlang documentation (Tutorial, Reference Manual stb.)
  `http://erlang.org/doc.html`
- Wikibooks on Erlang Programming
  `http://en.wikibooks.org/wiki/Erlang_Programming`
- On-line help (csak unix/linux rendszeren)

```
man erl
erl -man <module>
```

# Tartalom


1. Szekvenciális Erlang
   - Bevezetés
   - Erlang emulátor
   - Típusok
   - Szekvenciális programozás
   - Típusspecifikáció
   - Kivételkezelés
   - Rekord
   - Rekurzív adatstruktúrák
   - Könyvtári függvények


# Erlang emulátor

- Erlang shell (interaktív értelmező) indítása

```
$ erl
Erlang (BEAM) emulator version 5.6.3 [source]
  [async-threads:0] [hipe] [kernel-poll:false]
Eshell V5.6.3 (abort with ^G)
1>
```

- Bevitel befejezése, kiértékelés indítása

```
1> 3.2 + 2.1 * 2. % Lezárás és indítás „pont-bevitel”-lel!
7.4
2> atom.
atom
3> 'Atom'.
'Atom'
4> "string".
"string"
5> {ennes, 'A, a, 9.8}.
{ennes,'A',a,9.8}
6> [lista, 'A', a, 9.8].
[lista,'A',a,9.8]
```

# Erlang shell: parancsok

```
6> help().
** shell internal commands **
b() - display all variable bindings
e(N) - repeat the expression in query <N>
f() - forget all variable bindings
f(X) - forget the binding of variable X
h() - history
v(N) - use the value of query <N>
rr(File) - read record information from File (wildcards allowed)
...
** commands in module c **
c(File) - compile and load code in <File>
cd(Dir) - change working directory
help() - help info
l(Module) - load or reload module
lc([File]) - compile a list of Erlang modules
ls() - list files in the current directory
ls(Dir) - list files in directory <Dir>
m() - which modules are loaded
m(Mod) - information about module <Mod>
pwd() - print working directory
q() - quit - shorthand for init:stop()
...
```

# Erlang shell: ^G és ^C

- ^G hatása

```
User switch command
 --> h
  c [nn]    - connect to job
  i [nn]    - interrupt job
  k [nn]    - kill job
  j         - list all jobs
  s         - start local shell
  r [node] - start remote shell
  q         - quit erlang
  ? | h     - this message
 -->
```

- ^C hatása

```
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
(v)ersion (k)ill (D)b-tables (d)istribution
```

# Tartalom


1. Szekvenciális Erlang
   - Bevezetés
   - Erlang emulátor
   - Típusok
   - Szekvenciális programozás
   - Típusspecifikáció
   - Kivételkezelés
   - Rekord
   - Rekurzív adatstruktúrák
   - Könyvtári függvények


# Típus

- Az Erlang erősen típusos nyelv, bár nincs típusdeklaráció
- A típusellenőrzés dinamikus és nem statikus
  - Alaptípusok

| | |
|---|---|
| Szám (egész, lebegőpontos) | Number (integer, float) |
| Atom | Atom |
| Függvény | Fun |
| Ennes (rekord) | Tuple (record) |
| Lista | List |

  - További típusok

| | |
|---|---|
| Pid | Pid (Process identifier) |
| Port | Port |
| Hivatkozás | Reference |
| Bináris | Binary |

# Atom

- Szövegkonstans (nem füzér!)
- Kisbetűvel kezdődő, bővített alfanumerikus[1] karaktersorozat, pl. `sicstus`, `erlang_OTP`
- Bármilyen karaktersorozat is az, ha egyszeres idézőjelbe tesszük, pl. `'SICStus'`,`'erlang OTP'`,`'35 May'`
- Hossza tetszőleges, vezérlőkaraktereket is tartalmazhat, pl.
  `'ez egy hosszú atom, ékezetes betűkkel spékelve'`
  `'formázókarakterekkel \n\c\f\r'`
- Saját magát jelöli

[1] Bővített alfanumerikus: kis- vagy nagybetű, számjegy, aláhúzás (_), kukac (@).

# Term, változó

## Term

- Tovább nem egyszerűsíthető, ún. *kanonikus* kifejezés, érték a programban
- Minden termnek van típusa
- Termek összehasonlítási sorrendje (v.ö. típusok)
  `number < atom < reference < fun < port < pid < tuple < list < binary`

## Változó

- Nagybetűvel kezdődő, bővített alfanumerikus karaktersorozat, más szóval *név*
- A változó lehet *szabad* vagy *kötött*
- A szabad változónak nincs értéke, típusa
- A kötött változó értéke, típusa valamely konkrét term értéke, típusa
- Minden változóhoz csak egyszer köthető érték, azaz kötött változó nem kaphat értéket

# Szám

- Egész
  - Pl. `2008`, `-9`, `0`
  - Tetszőleges számrendszerben `radix#szám` alakban, pl. `2#101001`, `16#fe`
  - Az egész korlátlan pontosságú, pl. `12345678901234567890123456789012345678901234`
- Lebegőpontos
  - Pl. `3.14159`, `0.2e-22`
- Karakterkód
  - Ha nyomtatható: `$z`
  - Ha vezérlő: `$\n`
  - Oktális számmal: `$\012`

# Ennes

- Rögzített számú, tetszőleges kifejezésből álló sorozat
- Példák: `{2008, erlang}`, `{'Joe', 'Armstrong', 16.99}`
- Nullas: `{}`
- Egyelemű ennes $\neq$ ennes eleme, pl. `{elem}` $\neq$ `elem`

# Lista

- Korlátlan számú, tetszőleges kifejezésből álló sorozat
- Lineáris rekurzív adatszerkezet:
  - vagy üres (`[]` jellel jelöljük),
  - vagy egy elemből áll, amelyet egy lista követ: `[Elem|Lista]`
- Első elemét, ha van, a lista *fejének* nevezzük
- Első eleme utáni, esetleg üres, részét a lista *farkának* nevezzük
  - Egyelemű lista: `[elem]`
  - Fejből-farokból létrehozott lista: `[elem|[]]`, `['első'|['második']]`
  - Többelemű lista: `[elem,'elem',123,3.14]`, `[elem,123|[3.14]]`, `[elem,123|[3.14,'elem']]`
- A konkatenáció műveleti jele: `++`
  Pl. `['egy'|['két']] ++ [elem,123|[3.14,'elem']] =`
  `[egy,két,elem,123,3.14,elem]`

# Füzér

- Csak rövidítés, tkp. karakterkódok listája, pl.
  `"erl"` = `[$e,$r,$l]` = `[101,102,103]`
- A nyomtatható karakterkódok listáját füzérként írja ki:
  `[101,114,108]` $\rightarrowtail$[2] `"erl"`
- Ha más érték is van a listában, listaként írja ki:
  `[31,101,114,108]` $\rightarrowtail$ `[31,101,114,108]`
  `[a,101,114,108]` $\rightarrowtail$ `[a,101,114,108]`
- A konkatenáció műveleti jele: `++`, pl.
  `"erl"++"ang"` = `"erlang"`
- Füzérek esetén a `++` egymás mellé írással helyettesíthető, pl.
  `"erl""ang"` = `"erl" "ang"` = `"erl"++"ang"`

[2] `Kif` $\rightarrowtail$ jelentése: „`Kif` kiértékelésének eredménye”.

# Tartalom


1. Szekvenciális Erlang
   - Bevezetés
   - Erlang emulátor
   - Típusok
   - Szekvenciális programozás
   - Típusspecifikáció
   - Kivételkezelés
   - Rekord
   - Rekurzív adatstruktúrák
   - Könyvtári függvények


# Minta, mintaillesztés

- Minta: term alakú kifejezés, amelyben szabad változó is lehet
- Sikeres illesztés esetén a szabad változók kötötté válnak, értékük a megfelelő részkifejezés lesz.
- A mintaillesztés (!) műveleti jele: $=$
- Mintaillesztés $\neq$ értékadás!
- Példák:

  `Pi = 3.14159` $\rightsquigarrow$[3] `Pi` $\mapsto$ `3.14159`[4]

  `[H1|T1] = [1,2,3]` $\rightsquigarrow$ `H1` $\mapsto$ `1`, `T1` $\mapsto$ `[2,3]`

  `[1,2|T2] = [1,2,3]` $\rightsquigarrow$ `T2` $\mapsto$ `[3]`

  `[H2|[3]] = [1,2,3]` $\rightsquigarrow$ hiba

  `{A1,B1} = {{a},'Beta'}` $\rightsquigarrow$ `A1` $\mapsto$ `{a}`, `B1` $\mapsto$ `'Beta'`

  `{{a},B2} = {{a},'Beta'}` $\rightsquigarrow$ `B2` $\mapsto$ `'Beta'`

[3] `Kif` $\rightsquigarrow$ jelentése: „`Kif` kiértékelése után”.

[4] `X` $\mapsto$ `V` jelentése: „`X` a `V` értékhez van kötve”.

# Kifejezés

- Lehet
  - term, változó, minta
    - ⋆ term $\subseteq$ kifejezés, term $\subseteq$ minta
    - ⋆ változó $\subseteq$ kifejezés, változó $\subseteq$ minta

  továbbá
  - összetett kifejezés, függvényalkalmazás, szekvenciális kifejezés
- Kifejezés kiértékelése alapvetően: **mohó**.

Összetett kifejezés

- Műveleti jelekkel összekapcsolt két vagy több kifejezés

# Kifejezés (folyt.)

## Függvényalkalmazás

- Szintaxisa
  - `fnev(arg1, arg2, ..., argn)`
  - `modul:fnev(arg1, arg2, ..., argn)`
- Pl. `length([a,b,c])` $\rightarrowtail$ `3`
  és `erlang:tuple_size({1,a,'A',"1aA"})` $\rightarrowtail$ `4`

# Kifejezés (folyt.)

## Szekvenciális kifejezés

- Szintaxisa
  - **`begin`** `exp1, exp2, ..., expn` **`end`**
  - `exp1, exp2, ..., expn`
- A `begin` és `end` párt akkor kell kiírni, ha az adott helyen egyetlen kifejezésnek kell állnia
- Értéke az utolsó kifejezés értéke
- Pl. `begin a, "a", 5, [1,2] end` $\rightarrowtail$ `[1,2]`

# Aritmetikai műveletek

- Matematikai műveletek
  - Előjel: `+`, `-` (precedencia: 1)
  - Multiplikatív: `*`, `/`, `div`, `rem` (precedencia: 2)
  - Additív: `+`, `-` (precedencia: 3)
- Bitműveletek
  - `bnot`, `band` (precedencia: 2)
  - `bor`, `bxor`, `bsl`, `bsr` (precedencia: 3)
- Megjegyzések
  - `+`, `-`, `*` és `/` egész és lebegőpontos operandusokra is alkalmazhatók, a típuskonverzió automatikus
  - `+`, `-` és `*` eredménye egész, ha mindkét operandusuk egész, egyébként lebegőpontos
  - `/` eredménye mindig lebegőpontos
  - `div` és `rem`, valamint a bitműveletek operandusai csak egészek lehetnek

# Logikai műveletek

- Összehasonlítás
  - Kisebb-nagyobb reláció (típuskonverzióval, ha kell): `<`, `=<`, `>=`, `>`
  - Egyenlőségi reláció (típuskonverzióval, ha kell): `==`, `/=`
  - Azonosság (típuskonverzió nélkül): `=:=`, `=/=`
  - Összehasonlítás eredménye: a `true` vagy a `false` atom
  - Típuskonverzió: `float`-tá alakít, ha az egyik operandus `float`, a másik `integer`
  - Termek összehasonlítási sorrendje (v.ö. típusok): `number < atom < reference < fun < port < pid < tuple < list < binary`
- Logikai művelet: `not`, `and`, `or`, `xor`
- Lusta kiértékelésű („short-circuit”) logikai művelet: `andalso`, `orelse`
- Példák:

  `false and (3 div 0 == 2)` $\rightarrowtail$ Hiba

  `false andalso (3 div 0 == 2)` $\rightarrowtail$ `false`

# Beépített függvények

- BIF (Built-in functions)
  - a futtatórendszerbe beépített, rendszerint C-ben írt függvények
  - többségük a kernel-könyvtár `erlang` moduljának része
  - többnyire rövid néven (az `erlang:` modulnév nélkül) hívhatók
- Az alaptípusokon alkalmazható leggyakoribb BIF-ek:
  - Számok:
    `abs(Num), trunc(Num), round(Num), float(Num)`
  - Lista:
    `length(List), hd(List), tl(List)`
  - Ennes:
    `tuple_size(Tuple), element(Index, Tuple),`
    `setelement(Index, Tuple, Value)`
    Megjegyzés: $1 \leq$ `Index` $\leq$ `tuple_size(Tuple)`
- Rendszer:
  `date(), time(), erlang:localtime(), halt(),`
  `exit(Reason)`

# Típusvizsgálat, típuskonverzió BIF-ekkel

- Típusvizsgálat
  - `is_integer(Term)`, `is_float(Term)`,
  - `is_number(Term)`, `is_atom(Term)`,
  - `is_boolean(Term)`,
  - `is_tuple(Term)`, `is_list(Term)`,
  - `is_function(Term)`, `is_function(Term,Arity)`
- Típuskonverzió
  - `atom_to_list(Atom)`, `list_to_atom(String)`,
  - `integer_to_list(Int)`, `list_to_integer(String)`, `erlang:list_to_integer(String, Base)`,
  - `float_to_list(Float)`, `list_to_float(String)`,
  - `tuple_to_list(Tuple)`, `list_to_tuple(List)`

# Magasabb rendű függvények alkalmazása

- Leképzés: `lists:map(Fun,List)`
  A `List` lista `Fun`-nal transzformált elemeiből álló lista
- Szűrés: `lists:filter(Pred,List)`
  A `List` lista `Pred`-et kielégítő elemeinek listája
- Redukálás
  Jobbról balra haladva: `lists:foldr(Fun,Acc,List)`
  Balról jobbra haladva: `lists:foldl(Fun,Acc,List)`

  A `List` lista elemeiből és az `Acc` elemből a kétoperandusú `Fun`-nal képzett érték

  Példa `foldr` kiértékelési sorrendjére:
  `1+(2+(3+(4+(5+Acc))))`
  Példa `foldl` kiértékelési sorrendjére:
  `5+(4+(3+(2+(1+Acc))))`

# Függvénydeklaráció

- Egy vagy több, pontosvesszővel (;) elválasztott *klózból* állhat.
- Alakja:

```
fnev(A11, ..., A1m) [when Őr1] -> SzekvenciálisKif1;
...;
fnev(An1, ..., Anm) [when Őrn] -> SzekvenciálisKifn.
```

- A függvényt a neve, az „aritása” (paramétereinek száma), valamint a moduljának a neve azonosítja.
- Az azonos nevű, de eltérő aritású függvények nem azonosak!
- Példák:

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

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

# Függvényérték

- A funkcionális nyelvekben a függvény is *érték:*
  - leírható (jelölhető),
  - van típusa,
  - névhez (változóhoz) köthető,
  - adatszerkezet eleme lehet,
  - paraméterként átadható,
  - függvényalkalmazás eredménye lehet.
- Magasabb rendű függvény: paramétere $\vee$ eredménye függvény
- Névtelen függvény (függvényjelölés) mint érték

```
fun (M11, ..., M1m) [when Őr1] -> SzekvenciálisKif1;
    ...;
    (Mn1, ..., Mnm) [when Őrn] -> SzekvenciálisKifn
end.
```

- Már deklarált függvény mint érték

```
fun Fnev/Aritas % az Fnev-et deklaráló a modulban
fun Modul:Fnev/Aritas
```

# Függvényérték: példák

```
Area1 = fun ({circle,R}) -> R*R*3.14159;
            ({rectan,A,B}) -> A*B;
            ({square,A}) -> A*A
        end.

Area1({circle,2}).

Area2 = fun dpr:area/1.

Area2({square,2.5}).

fun dpr:area/1({circle,5.2}).

F1 = [Area1, Area2, fun dpr:area/1, 12, area].

(lists:nth(1,F1))({circle,2}).

(lists:nth(3,F1))({circle,2}).
```

# Listaműveletek

- Listák összefűzése (konkatenációja): `++`,
  `append(Lst1,Lst2)`, `append(LstOfLsts)`
  `Cs = As++Bs` $\rightsquigarrow$ `Cs` $\mapsto$ az `As` összes eleme a `Bs` elé fűzve az eredeti sorrendben
- Példa
  `[a,'A',[65]]++[1+2,2/1,'A']` $\rightarrowtail$
  `[a,'A',"A",3,2.0,'A']`
- Listák különbsége: `--`, `subtract(Lst1,Lst2)`
  `Cs = As--Bs` $\rightsquigarrow$ `Cs` $\mapsto$ az `As` olyan másolata, amelyből ki van hagyva a `Bs`-ben előforduló összes elem balról számított első előfordulása, feltéve, hogy volt ilyen elem `As`-ben
- Példa
  `[a,'A',[65],'A']--["A",2/1,'A']` $\rightarrowtail$ `[a,'A']`

# Listanézet

- Listanézet: `[Kif || Minta <- Lista, Feltétel]`
  `Feltétel` tetszőleges logikai kifejezés lehet. A `Mintá`ban előforduló változónevek elfedik a listakifejezésen kívüli azonos nevű változókat.
- Kis példák

  `[2*X || X <- [1,2.0,3]]` ⟼ `[2,4.0,6]`

  `[2*X || X <- [1,2,3], X rem 2 /= 0, X > 2]` ⟼ `[6]`
- Pitagoraszi számhármasok

```
pitag(N) ->
  [{A,B,C} ||
     A <- lists:seq(1,N),
     B <- lists:seq(1,N),
     C <- lists:seq(1,N),
     A+B+C =< N,
     A*A+B*B =:= C*C
  ].
```

# Listanézet: érdekes példák

- Quicksort

```
qsort([]) -> [];
qsort([Pivot|Tail]) ->
  qsort([X || X <- Tail, X < Pivot])
  ++ [Pivot] ++
  qsort([X || X <- Tail, X >= Pivot]).
```

- Permutáció

```
perms([]) -> [[]];
perms(L) ->
  [[H|T] || H <- L, T <- perms(L--[H])].
```

# Őr

- Nézzük újra a következő definíciót:

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

  - Mi történik, ha megváltoztatjuk a klózok sorrendjét?
  - Mi történik, ha `-1`-re alkalmazzuk?
  - És ha `2.5`-re?

  A baj az, hogy a `fac(N) -> ...` klóz túl általános.
- Megoldás: korlátozzuk a mintaillesztést őr alkalmazásával

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

# Őr, őrszekvencia, őrkifejezés

- Az őrrel ún. *nem-strukturális tulajdonságot* írunk elő, amit mintaillesztéssel nem tudunk leírni.
- Az őrt a `when` kulcsszó vezeti be.
- Az őrben előforduló összes változónak *kötöttnek* kell lennie.

Elnevezések:

- **Őrszekvencia:** egyetlen őr vagy őrök pontosvesszővel (;) elválasztott sorozata
  - `true`, ha legalább egy őr `true` (vagy-kapcsolat)
- **Őr:** egyetlen őrkifejezés vagy őrkifejezések vesszővel (,) elválasztott sorozata
  - `true`, ha az összes őrkifejezés `true` (és-kapcsolat)
- **Őrkifejezés:** olyan speciális kifejezés, amelyben csak mellékhatás nélküli Erlang-kifejezések lehetnek

# Őrkifejezés, őr

Őrkifejezés:

- Nem Erlang-kifejezés, csak ahhoz hasonló
- Vagy sikerül, vagy meghiúsul
- Hibát (kivételt) **nem** jelezhet; ha hibás az argumentuma, meghiúsul

Őr:

- A mintával együtt választja ki a kiértékelendő kifejezést
- A kiválasztásnak hatékonynak és előre látható eredményűnek kell lennie

# Őrkifejezés

Őrkifejezés lehet:

- `true` atom $\rightsquigarrow$ *sikerül*, bármely más term és kötött változó $\rightsquigarrow$ *meghiúsul*
- Típust vizsgáló predikátumok, aritmetikai és logikai kifejezések (korábban volt már róluk szó)

Őrkifejezés **nem** lehet:

- Mellékhatással járó kifejezés
- Felhasználó által definiált függvény, mert esetleg mellékhatása lehet

# Őr: példák

```
X when is_integer(X), X > 0

X when is_number(X), X =< 0

X when is_tuple(X), tuple_size(X) >= 2,
       element(2,X) =:= valami

X when is_list(X),
       (length(X) >= 4 orelse hd(X) =:= 'A')

{X,Y,Z} when X > Y; X > Z

{X,Y} when integer(X), X > 0; integer(Y), Y < 0

[X,Y|Z] when is_integer(X), is_integer(Y)
       andalso X>0; Y>0 andalso Z =/= []

[X,Y|Z]) when is_integer(X), is_integer(Y),
       is_integer(hd(Z))
```

# Feltételes kifejezés mintaillesztéssel

- **case** `Kif` **of**
  `Minta`$_1$ [**when** `Őr`$_1$] -> `SzekvenciálisKif`$_1$;
  ...
  `Minta`$_n$ [**when** `Őr`$_{1n}$] -> `SzekvenciálisKif`$_n$
  **end**.
- Kiértékelés: balról jobbra
- Értéke: az első illeszkedő minta utáni szekvenciális kifejezés
- Ha nincs ilyen minta, hibát jelez
- Példák:

```
X=2, case X of 1 -> "1"; 3 -> "3" end.
  ** exception error: no case clause matching 2

10> X=2, case X of 1 -> "1"; 2 -> "2" end.
  "2"

X=2, Y=3, case X of 2 when Y == 2 -> "1+3"; 3 -> "3" end.
  ** exception error: no case clause matching 2

X=2, Y=3, case X of 2 when Y == 3 -> "1+3"; 3 -> "3" end.
  "1+3"
```

# Feltételes kifejezés őrrel

- **if**
  Őr$_1$ -> SzekvenciálisKif$_1$;
  ...
  Őr$_n$ -> SzekvenciálisKif$_n$
  **end**.
- Kiértékelés: balról jobbra.
- Értéke: az első teljesülő őr utáni szekvenciális kifejezés
- Ha nincs ilyen őr, futáskor hibát jelez.
- Példák

```
X=2, if 2<X -> "<" end.
  ** exception error: no true branch ...

X=2, if 2<X -> "<"; X>2 -> ">" end.
  ** exception error: no true branch ...

X=2, if 2<X -> "<"; X>=2 -> ">=" end.
  ">="
```

# Modul

- Modul: attribútumok és függvénydeklarációk sorozata
- Attribútumok
  - Modulnév (atom; a fájl nevével azonosnak kell lennie):
    `-module(name)` % A fájlnév kiterjesztése: `.erl`
  - Kívülről is látható függvények listája
    `-export([`$f_1$`/arity,...,`$f_n$`/arity])`
  - Más modulok modulnév nélkül használható függvényeinek listája
    `-import([`$f_1$`/arity,...,`$f_n$`/arity])`
    `-include("filename.ext")`
    `-define(makrónév, helyettesítés)`
  - Függvénydeklaráció: lásd ott.
- Példák

Deklarációk

```
-define(pi,3.14).  % Csak modulban!
area({circle,R}) -> R*R*pi;
area({rectan,A,B}) -> A*B;
area({square,A}) -> A*A.
```

Hívások

```
dpr:area({circle,2}).
dpr:area({rectan,4.5,2}).
dpr:area({square,2.5}).
```

# Rekurzió

- Lineáris rekurzió
  Példa: lista hosszának meghatározása

```
len([]) -> 0;
len([_|T]) -> 1 + len(T).
```

- Elágazó rekurzió
  Példa: Fibonacci-számok

```
fib(0) -> 0;
fib(1) -> 1;
fib(N) -> fib(N-2) + fib(N-1).
```

  Mindkettőből *rekurzív processz* jön létre, ha alkalmazzuk: minden egyes rekurzív hívás mélyíti a vermet. (Hogyan?)

# Jobbrekurzió, iteráció

A rekurzió gyakran, esetleg egy vagy több argumentum (ún. akkumulátor) bevezetésével, jobbrekurzióvá alakítható.

- Példa: lista hosszának meghatározása

```
leni(L) -> leni(L,0).

leni([],N) -> N;
leni([_|T],N) -> leni(T,N+1).
```

- Példa: Fibonacci-számok

```
fibi(0) -> 0;
fibi(N) -> fibi(N,0,1).

fibi(1,Prev,Curr) -> Curr;
fibi(N,Prev,Curr) -> fibi(N-1,Curr,Prev+Curr).
```

- A jobbrekurzióból *iteratív processz* hozható létre, amely nem mélyíti a vermet.
- A segédfüggvényt jobb nem exportálni, hogy elrejtsük az akkumulátort.

# Lista bejárása

A lista rekurzív adatszerkezet, nyilvánvaló, hogy rekurzív módon érdemes bejárni.

- Mindig kell egy klóz az üres lista feldolgozására.
- Kell legalább egy klóz a nem üres lista feldolgozására.

Példa: Egészlista átlagának kiszámítása

```
average1(L) -> sumi(L) / leni(L).

sumi(L) -> sumi(L,0).

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

Hatékonyabb, ha csak egyszer járjuk be a listát:

```
average2(L) -> average2(L,0,0).

average2([],Sum,Len) -> Sum/Len;
average2([H|T],Sum,Len) ->
    average2(T,Sum+H,Len+1).
```

# Lista bejárása (folyt.)

Vannak esetek, amikor a nem üres listára több klózt kell írni.

- Példa: `X` eleme-e a nem rendezett `L` listának

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

- Példa: Lista legnagyobb eleme

```
max(X,Y) when X >= Y -> X; max(_X,Y) -> Y.

listmax([]) -> undefined;
listmax([X]) -> X;
listmax([X|T]) -> max(X,listmax(T)).

listmaxi([]) -> undefined;
listmaxi([X]) -> X;
listmaxi([X,Y|T]) -> listmaxi([max(X,Y)|T]).
```

# Lista feldolgozása magasabb rendű függvényekkel

Gyakran nem a kód, hanem a kódolás hatékonysága, valamint a megbízhatóság a legfontosabb cél.

- Átlagolás magasabb rendű függvényekkel

```
average(L) ->
    lists:foldl(fun(X,Y) -> X+Y end,0,L) / length(L).
```

- Lista maximuma, minimuma (vö. `lists:max/1`, `lists:min/1`)

```
listMax([]) -> undefined;
listMax([H|T]) -> lists:foldl(fun max/2,H,T).

listMin([]) -> undefined;
listMin([H|T]) ->
    F = fun(X,Y) when X >= Y -> Y; (X,_Y) -> X end,
    lists:foldl(F,H,T).
```

- Listák összefűzése (vö. `lists:append/2`, `lists:reverse/2`)

```
listAppend(L1,L2)->
    foldr(fun(E,L) -> [E|L] end,L2,L1).

listRevAppend(L1,L2)->
    foldl(fun(E,L) -> [E|L] end,L2,L1).
```

# Gyakori magasabb rendű függvények definíciója

```
map(_,[]) -> [];
map(Fun,[H|T]) -> [Fun(H)|map(Fun,T)].

filter(_,[]) -> [];
filter(Pred,[H|T]) ->
    case Pred(H) of
      true  -> [H|filter(Pred,T)];
      false -> filter(Pred,T)
    end.

map(Fun,L) -> [Fun(X) || X <- L].

filter(Pred,L) -> [X || X <- L, Pred(X)].

foldr(_,Acc,[]) -> Acc;
foldr(Fun,Acc,[H|T]) -> Fun(H,foldr(Fun,Acc,T)).

foldl(_,Acc,[]) -> Acc;
foldl(Fun,Acc,[H|T]) -> foldl(Fun,Fun(H,Acc),T).
```

# Tartalom


1. Szekvenciális Erlang
   - Bevezetés
   - Erlang emulátor
   - Típusok
   - Szekvenciális programozás
   - Típusspecifikáció
   - Kivételkezelés
   - Rekord
   - Rekurzív adatstruktúrák
   - Könyvtári függvények


# Típusspecifikáció

- Csak *dokumentációs konvenció,* nem nyelvi elem az Erlangban
- Készülnek programok a típusspecifikáció és a programkód összevetésére
- A *typeName* típust is jelöljük: `typeName()`.
- Típusok: előre definiált és felhasználó által definiált

# Előre definiált típusok

- `any()`, `term()`: bármely Erlang-típus
- `atom()`, `binary()`, `float()`, `function()`, `integer()`, `pid()`, `port()`, `reference()`: Erlang-alaptípusok
- `bool()`: a `false` és a `true` atomok
- `char()`: az `integer` típus karaktereket ábrázoló része
- `iolist() = [char()|binary()|iolist()]`[5]: karakter-io
- `tuple()`: ennestípus
- `list(L)`: `[L]` listatípus szinonimája
- `nil()`: `[]` üreslista-típus szinonimája
- `string()`: `list(char())` szinonimája
- `deep_string() = [char()|deep_string()]`
- `none()`: a „nincs típusa” típus; nem befejeződő függvény „eredményének” megjelölésére

[5] `...|...` választási lehetőség a szintaktikai leírásokban.

# Új (felhasználó által definiált) típusok

- Szintaxis: `@type newType() = TypeExpression.`
- Definíciós szabályok
  - Ennestípus
    `{T1,...,Tn}` típuskifejezés, ha `T1`,...,`Tn` típuskifejezések
  - Listatípus
    `[T]` típuskifejezés, ha `T` típuskifejezés
  - Uniótípus
    `T1|T2` típuskifejezés, ha `T1` és `T2` típuskifejezések
  - Függvénytípus
    `fun(T1,...,Tn) -> T` típuskifejezés, ha `T1`,...,`Tn` és `T` típuskifejezések
  - Típuskifejezés az előre definiált típus, a felhasználó által definiált típus és az előre definiált típus bármely példánya

# Függvénytípus specifikálása

Egy függvény típusát az argumentumainak (formális paramétereinek) és az eredményének (visszatérési értékének) a típusa határozza meg.

- Szintaxis: `@spec funcName(T1,...,Tn) -> Tret.`
- `T1,...,Tn` és `Tret` háromféle lehet:
  - `TypeVar`
    Típusváltozó, tetszőleges típus jelölésére
  - `TypeVar::Type`
    Típusváltozó, előírt típus jelölésére
  - `Type`
    Típuskifejezés

# Típusspecifikáció: példák

```
@type onOff()  = on | off.
@type person() = {person, name(), age()}.
@type people() = [person()].
@type name()   = {firstname, string()}.
@type age()    = integer().

@spec file:open(FileName, Mode) ->
                      {ok, Handle} | {error, Why}.
@spec file:read_line(Handle) -> {ok, Line} | eof.

@spec lists:map(fun(A) -> B, [A]) -> [B].
@spec lists:filter(fun(X) -> bool(), [X]) -> [X].

@type sspec() = size(), board().
@type size()  = integer().
@type field() = integer().
@type board() = [[field()]].
@spec sudoku(SudokuSpec::sspec) -> [SudokuSpec::sspec].
```

# Tartalom


1. Szekvenciális Erlang
   - Bevezetés
   - Erlang emulátor
   - Típusok
   - Szekvenciális programozás
   - Típusspecifikáció
   - Kivételkezelés
   - Rekord
   - Rekurzív adatstruktúrák
   - Könyvtári függvények


# Kivételkezelés

- Kivétel jelzése háromféleképpen lehetséges
  - `throw(Why)`
    Olyan hiba jelzésére, amelynek kezelése várható az alkalmazástól
  - `exit(Why)`
    A futó processz befejezésére
  - `erlang:error(Why)`
    Súlyos rendszerhiba jelzésére, amelynek kezelése nem várható az alkalmazástól
- Kivétel elkapása kétféleképpen lehetséges
  - **`try`** ... **`catch`** kifejezéssel
  - **`catch`** kifejezéssel

# Kivételkezelés: `try ... catch`

```
try FuncOrExprSeq of
    Pat1 [when Grd1] -> Expr1;
    Pat2 [when Grd2] -> Expr2;
    ...
catch
    ExcType1: ExcPat1 [when ExcGrd1] -> ExcExpr1;
    ExcType2: ExcPat2 [when ExcGrd2] -> ExcExpr2;
    ...
after
    AfterExprs
end
```

- Ha a `FuncOrExprSeq` kiértékelése sikeres, az értékét az Erlang megpróbálja az `of` és `catch` közötti mintákra illeszteni.
- Ha a kiértékelés sikertelen, az Erlang a jelzett kivételt próbálja meg illeszteni a `catch` és `after` közötti mintákra.
- Minden esetben kiértékeli az `after` és `end` közötti kifejezést.

# Példa `try ... catch` és `catch` használatára

```
genExc(A,1) -> A;
genExc(A,2) -> throw(A);
genExc(A,3) -> exit(A);
genExc(A,4) -> erlang:error(A).

tryGenExc(X,I) ->
    try genExc(X,I) of
        Val -> {I, 'Lefutott', Val}
    catch
        throw:X -> {I, 'Kivetelt dobott', X};
        exit:X  -> {I, 'Befejezodott', X};
        error:X -> {I, 'Sulyos hibat jelzett', X}
    end.

[catch dpr:genExc(X,I)
    || {X,I} <- [{'No',1},{'Th',2},{'Ex',3},{'Er',4}]].
```

# Tartalom


1. Szekvenciális Erlang
   - Bevezetés
   - Erlang emulátor
   - Típusok
   - Szekvenciális programozás
   - Típusspecifikáció
   - Kivételkezelés
   - Rekord
   - Rekurzív adatstruktúrák
   - Könyvtári függvények


# Rekord

- Ha egy ennesnek sok a tagja, nehéz felidézni, melyik mit tag jelent
- Ezért vezették be a rekordot - bár önálló rekordtípus nincs
- Rekord = címkézett ennes; szintaktikai édesítőszer
- Rekord deklarálása (csak modulban!):
  `-record(rn,{`$p_1$`=`$d_1$`,...,`$p_n$`=`$d_1$`}),`
  ahol
  - `rn`: rekordnév,
  - $p_i$: mezőnév,
  - $d_i$: alapértelmezett érték (opcionális).
- Rekord létrehozása és változóhoz kötése:
  `X=#rn{`$m_1$`=`$v_1$`,...,`$m_n$`=`$v_n$`}`
- Egy mezőérték lekérdezése: `X#rn.`$m_i$
- Egy/több mezőérték változóhoz kötése: `#rn{`$m_2$`=V,`$m_4$`=W} = X`

# Rekord: példák

- A `dprrec.hrl` rekorddefiníciós fájl tartalma:

```
-record('TODO',{sts=remind,who='HP',txt}).
```

  Csak így használhatja több Erlang modul ugyanazt a rekorddefiníciót.

- Deklaráció beolvasása

```
1> rr("dprrec.hrl").
['TODO']
```

- Új, alapértelmezett rekord (`X`) létrehozása

```
2> X = #'TODO'{}.
#'TODO'{sts = remind,who = 'HP',txt = undefined}
```

- `X1` is új

```
3> X1 = #'TODO'{sts=urgent,txt="Diák!"}.
#'TODO'{sts = urgent,who = 'HP',txt = "Diák!"}
```

- Rekord (`X1`) másolása frissítéssel; `X2` is új

```
4> X2 = X1#'TODO'{sts=done}.
#'TODO'{sts = done,who = 'HP',txt = "Diák!"}
```

# Rekord: további példák

- Mezőértékek lekérdezése

```
5> #'TODO'{who=W,txt=T} = X2.
#'TODO'{sts = done,who = HP,txt = "Diák!"}

6> W.
'HP'

7> T.
"Diák!"

>8 X1#'TODO'.sts.
urgent
```

- Rekorddeklaráció elfelejtetése

```
9> rf('TODO').
ok

10> X2.
{'TODO',done,'HP',"Diák!"}
```

A rekord az Erlangon belül: ennes.

# Tartalom


1. Szekvenciális Erlang
   - Bevezetés
   - Erlang emulátor
   - Típusok
   - Szekvenciális programozás
   - Típusspecifikáció
   - Kivételkezelés
   - Rekord
   - Rekurzív adatstruktúrák
   - Könyvtári függvények


# Lineáris rekurzív adatstruktúrák (pl. verem)

- Verem: ennessel valósítjuk meg, listával triviális lenne
- Műveletek: üres verem létrehozása, verem üres voltának vizsgálata, egy elem berakása, utoljára berakott elem kivétele

```
-module(stack).          %% stack.erl
-compile(export_all).

%% @type stack() = empty | {any(),stack()}

empty() -> empty.

is_empty(empty) -> true;
is_empty({_,_}) -> false.

insert(X,empty) -> {X,empty};
insert(X,{_V,_S}=VS) -> {X,VS}.

return(empty) -> error;
return({V,S}) -> {V,S}.

st(3) -> insert(1,(insert(2,insert(3,empty())))).
```

- `{_V,_S}=VS`: réteges minta

# Elágazó rekurzív adatstruktúrák (pl. bináris fa)

- Műveletek bináris fákon: létrehozása, mélysége, leveleinek száma

```
-module(tree).          %% tree.erl
-compile(export_all).

%% @type btr() = leaf | {any(),btr(),btr()}.

empty() -> leaf.

node(V, Lt, Rt) -> {V,Lt,Rt}.

max(X,Y) when X>Y -> X; max(_X,Y) -> Y.

depth(leaf) -> 0;
depth({_,Lt,Rt}) -> 1+max(depth(Lt),depth(Rt)).

leaves(leaf) -> 1;
leaves({_,Lt,Rt}) -> leaves(Lt)+leaves(Rt).
```

# Bináris fa (folyt.): listából fa, fából lista

```
listToTree([]) -> empty();
listToTree(L) ->
    {L1,L2} = lists:split(length(L) div 2, L),
    node(hd(L2), listToTree(L1), listToTree(tl(L2))).

ilist(N) -> lists:seq(1,N,1).

alist(F,N) -> lists:map(fun(X) ->
    list_to_atom([X-1+F]) end, ilist(N)).

treeToList_in(leaf) -> [];
treeToList_in({V,Lt,Rt}) ->
    treeToList_in(Lt) ++ [V] ++ treeToList_in(Rt).

treeToList_pre(leaf) -> [];
treeToList_pre({V,Lt,Rt}) ->
    [V] ++ treeToList_pre(Lt) ++ treeToList_pre(Rt).
```

# Tartalom


1. Szekvenciális Erlang
   - Bevezetés
   - Erlang emulátor
   - Típusok
   - Szekvenciális programozás
   - Típusspecifikáció
   - Kivételkezelés
   - Rekord
   - Rekurzív adatstruktúrák
   - Könyvtári függvények


# Füzérkezelő függvények (`string` modul)

- `len(Str)`, `equal(Str1,Str2)`, `concat(Str1,Str2)`
- `chr(Str,Chr)`, `rchr(Str,Chr)`, `str(Str,SubStr)`, `rstr(Str,SubStr)`
  A karakter / részfüzér első / utolsó előfordulásának indexe, vagy 0, ha nincs benne
- `span(Str,Chrs)`, `cspan(Str,Chrs)`
  Az `Str` ama prefixumának hossza, amelyben kizárólag a `Chars`-beli karakterek fordulnak / nem fordulnak elő
- `substr(Str,Strt,Len)`, `substr(Str,Strt)`
  Az `Str` specifikált részfüzére

# További füzérkezelő függvények (`string` modul)

- `tokens(Str,SepList)`
  A `SepList` karakterei mentén füzérek listájára bontja az `Str`-t
- `join(StrList,Sep)`
  Füzérré fűzi össze, `Sep`-pel elválasztva, az `StrList` elemeit
- `strip(Str)`, `strip(Str,Dir)`, `strip(Str,Dir,Char)`
  A formázó / `Char` karaktereket levágja a füzér elejéről / végéről

Részletek és továbbiak: Reference Manual.

# Listakezelő függvények (`lists` modul)

- `nth(N,Lst)`, `nthtail(N,Lst)`, `last(Lst)`
  A `Lst` `N`-edik karaktere / ott kezdődő farka / utolsó eleme
- `append(Lst1,Lst2)`, `append(LstOfLsts)`
  Az `Lst1` és `Lst2` / `LstOfLsts` elemei egy listába fűzve
- `concat(Lst)`
  Az `Lst` összes eleme füzérré alakítva és egybefűzve
- `reverse(Lst)`, `reverse(Lst,Tl)`
  Az `Lst` megfordítva / megfordítva a `Tl` elé fűzve (más deklaratív nyelvekben `reverse/2`-nek `revAppend` a neve)
- `flatten(DeepList)`, `flatten(DeepList,Tail)`
  A `DeepList` kisimítva / kisimítva `Tail` elé fűzve
- `max(Lst)`, `min(Lst)`
  Az `Lst` legnagyobb / legkisebb eleme

# További listakezelő függvények (`lists` modul)

- `filter(Pred,Lst)`, `delete(Elem,Lst)`
  A `Lst` `Pred`-et kielégítő elemek / `Elem` nélküli másolata
- `takewhile(Pred,Lst)`, `dropwhile(Pred,Lst)`
  Az `Lst` `Pred`-et kielégítő prefixumát tartalmazó / nem tartalmazó másolata
- `partition(Pred,Lst)`, `split(N,Lst)`
  A `Lst` elemei `Pred` / `N` szerint két listába válogatva
- `member(Elem,Lst)`, `all(Pred,Lst)`, `any(Pred,Lst)`
  Igaz, ha `Elem` / `Pred` szerinti minden / `Pred` szerinti legalább egy elem benne van az `Lst`-ben
- `prefix(Lst1,Lst2)`, `suffix(Lst1,Lst2)`
  Igaz, ha az `Lst2` az `Lst1`-gyel kezdődik / végződik

# Továbbra is: listakezelő függvények (`lists` modul)

- `sublist(Lst,Len)`, `sublist(Lst,Strt,Len)`
  Az `Lst` 1-től / `Strt`-től kezdődő, `Len` hosszú része
- `subtract(Lst1,Lst2)`
  Az `Lst1` `Lst2` elemeinek első előfordulását nem tartalmazó másolata
- `zip(Lst1,Lst2)`, `unzip(Lst)`
  Az `Lst1` és `Lst2` elemeiből képzett párok listája; az `Lst`-ben lévő párok szétválasztásával létrehozott két lista
- `sort(Lst)`, `sort(Fun,Lst)`
  Az `Lst` alapértelmezés / `Fun` szerint rendezett másolata
- `merge(LstOfLsts)`
  Az `LstOfLsts` listában lévő rendezett listák alapértelmezés szerinti összefuttatása

# Még mindig: listakezelő függvények (`lists` modul)

- `merge(Lst1,Lst2)`, `merge(Fun,Lst1,Lst2)`,
  A rendezett `Lst1` és `Lst2` listák alapértelmezés / `Fun` szerinti összefuttatása

- `map(Fun,Lst)`
  Az `Lst` `Fun` szerinti átalakított elemeiből álló lista

- `foreach(Fun,Lst)`
  Az `Lst` elemeire a mellékhatást okozó `Fun` alkalmazása

- `sum(Lst)`
  Az `Lst` elemeinek összege, ha az összes elem számot eredményező kifejezés

- `foldl(Fun,Acc,Lst)`, `foldr(Fun,Acc,Lst)`
  Az `Acc` akkumulátor és az `Lst` elemeinek `Fun` szerinti redukálása, balról jobbra, illetve jobbról balra haladva

Részletek és továbbiak: Reference Manual.

# Néhány további könyvtári modul és függvény

- `math` modul: `pi()`, `sin(X)`, `acos(X)`, `tanh(X)`, `asinh(X)`, `exp(X)`, `log(X)`, `log10(X)`, `pow(X,Y)`, `sqrt(X)`
- `io` modul: `write([IoDev,]Term)`, `fwrite(Format)`, `fwrite([IoDev,]Format,Data)`, `nl([IoDev])`, `format(Format)`, `format([IoDev,]Format,Data)`, `get_line([IoDev,]Prompt)`, `read([IoDev,]Prompt)`
- Formázójelek (`io` modul)

| | | | |
|---|---|---|---|
| `~~` | a ~ jel | `~c` | az adott kódú karakter |
| `~s` | füzér | `~f`, `~e`, `~g` | lebegőpontos szám |
| `~b`, `~x` | egész | `~w`, `~p` | Erlang-term |
| `~n` | újsor | | |

- Példa

```
11> io:format("~s ~b ~c ~f~n",
      [[$a,$b,$c],$a,$b,math:exp(1)]).
abc 97 b 2.718282
ok
```