

Az Erlang programozási nyelv

- Deklaratív programozás
- Funkcionális programozás

Hanák Péter

hanak@iit.bme.hu



Tartalom

1 Szekvenciális Erlang

- Deklaratív programozás, funkcionális programozás
- Erlang-bevezetés
- Erlang emulátor
- Típusok
- Szekvenciális programozás
- Típusspecifikáció
- Kivételkezelés
- Rekord
- Rekurzív adatstruktúrák
- Gyakori könyvtári függvények
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Listák használata: futamok
- Listák rendezése
- Kifejezés kiértékelése
- Absztrakció függvényekkel, eljárásokkal



Deklaratív programozás: mi az?

- Egy deklaratív program elsősorban a megoldandó feladatot, a várt eredményt írja le, és csak másodsorban a megoldás módját.
A gyakorlatban mindkét szemponttal foglalkozni kell, de nem egyenlő hangsúllyal.
- Egy program akkor deklaratív, ha tisztán *funkcionális*, *logikai*, vagy *korlátalapú* nyelven van megírva. (Wikipedia; túlzó elvárás)
- A deklaratív program kettős szemantikájú:
 - ▶ deklaratív szemantika:
MIT (milyen feladatot) old meg a program;
 - ▶ procedurális szemantika:
HOGYAN (milyen algoritmussal) oldja meg a feladatot.



Funkcionális programozás: mi az?

- Programozás *függvények alkalmazásával*.
- Kevésbé elterjedten *applikatív programozásnak* is nevezik (vö. function application).
- A függvény: leképezés – az argumentumából állítja elő az eredményt.

A tiszta (matematikai) függvénynek nincs *mellékhatása*.

Példák funkcionális programozási nyelvekre, nyelvcsaládokra:

- Lisp, Scheme
- SML, Caml, Caml Light, OCaml, Alice
- Clean, Haskell
- Erlang
- F#



Tartalom

1 Szekvenciális Erlang

- Deklaratív programozás, funkcionális programozás
- **Erlang-bevezetés**
- Erlang emulátor
- Típusok
- Szekvenciális programozás
- Típusspecifikáció
- Kivételkezelés
- Rekord
- Rekurzív adatstruktúrák
- Gyakori könyvtári függvények
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Listák használata: futamok
- Listák rendezése
- Kifejezés kiértékelése
- Absztrakció függvényekkel, eljárásokkal



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.
ISBN-13 978-1-934356-00-5
<http://www.pragprog.com/titles/jaerlang/programming-erlang>
- Francesco Cesarini, Simon Thompson: Erlang Programming. O'Reilly, 2009. ISBN 978-0-596-51818-9
<http://oreilly.com/catalog/9780596518189/>
- 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` vagy `erl -man <module>`



Tartalom

1 Szekvenciális Erlang

- Deklaratív programozás, funkcionális programozás
- Erlang-bevezetés
- **Erlang emulátor**
- Típusok
- Szekvenciális programozás
- Típusspecifikáció
- Kivételkezelés
- Rekord
- Rekurzív adatstruktúrák
- Gyakori könyvtári függvények
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Listák használata: futamok
- Listák rendezése
- Kifejezés kiértékelése
- Absztrakció függvényekkel, eljárásokkal



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

- Deklaratív programozás, funkcionális programozás
- Erlang-bevezetés
- Erlang emulátor
- **Típusok**
- Szekvenciális programozás
- Típusspecifikáció
- Kivételkezelés
- Rekord
- Rekurzív adatstruktúrák
- Gyakori könyvtári függvények
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Listák használata: futamok
- Listák rendezése
- Kifejezés kiértékelése
- Absztrakció függvényekkel, eljárásokkal



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



- Szövegkonstans (nem fűzér!)
- Kisbetűvel kezdődő, bővített alfanumerikus¹ 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

¹Bővített 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



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



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]`



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

²`Kif  $\mapsto$` jelentése: „`Kif` kiértékelésének eredménye”.

Tartalom

1 Szekvenciális Erlang

- Deklaratív programozás, funkcionális programozás
- Erlang-bevezetés
- Erlang emulátor
- Típusok
- **Szekvenciális programozás**
- Típusspecifikáció
- Kivételkezelés
- Rekord
- Rekurzív adatstruktúrák
- Gyakori könyvtári függvények
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Listák használata: futamok
- Listák rendezése
- Kifejezés kiértékelése
- Absztrakció függvényekkel, eljárásokkal



Minta, mintaillesztés

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

$$Pi = 3.14159 \rightsquigarrow^3 Pi \mapsto 3.14159^4$$

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

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

$$[H2 | [3]] = [1, 2, 3] \rightsquigarrow \text{hiba}$$

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

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

³Ki f $\rightsquigarrow$ jelentése: „Ki f kiértékelése után”.

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

Kifejezés

- Lehet

- ▶ term
- ▶ változó
- ▶ minta

- ★ $\text{term} \subseteq \text{kifejezés}$, $\text{term} \subseteq \text{minta}$
- ★ $\text{változó} \subseteq \text{kifejezés}$, $\text{változó} \subseteq \text{minta}$

továbbá

- ▶ összetett kifejezés
- ▶ szekvenciális kifejezés
- ▶ függvényalkalmazás

- Kifejezés kiértékelése alapvetően: **mohó**.

- ▶ Mit nevezünk mohó kiértékelésnek?
- ▶ Mit nevezünk lusta vagy szükség szerinti kiértékelésnek?



Kifejezés: összetett és szekvenciális

Összetett kifejezés

- Műveleti jelekkel összekapcsolt két vagy több kifejezés

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` $\mapsto [1,2]$



Kifejezés: függvényalkalmazás

Függvényalkalmazás

- Szintaxisa

- ▶ `fnev(arg1, arg2, ..., argn)`

- vagy

- ▶ `modul:fnev(arg1, arg2, ..., argn)`

- Pl. `length([a,b,c])` $\mapsto 3$

- és `erlang:tuple_size({1,a,'A',"1aA"})` $\mapsto 4$



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



Relációk

- Kisebb-nagyobb reláció (automatikus típuskonverzióval, ha kell):
`<, =<, >=, >`
- Egyenlőségi reláció (automatikus típuskonverzióval, ha kell):
`==, /=`
- Azonosság (automatikus típuskonverzió nélkül):
`=:=, /==`
- Az összehasonlítás eredménye:
`a true` vagy `a false` atom
- Automatikus 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 műveletek

- Logikai művelet:
`not`, `and`, `or`, `xor`
- Lusta kiértékelésű („short-circuit”) logikai művelet:
`andalso`, `orelse`
- Példák:
`false and (3 div 0 == 2)  $\rightarrow$  Hiba`
`false andalso (3 div 0 == 2)  $\rightarrow$  false`



Beépített függvények (BIF)

- BIF (Built-in functions)

- ▶ a futtatórendszerbe beépített, rendszerint C-ben írt függvények
- ▶ többségük az **erts**-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: $1 \leq \text{Index} \leq \text{tuple_size}(\text{Tuple})$



További BIF-ek

- Rendszer:

```
date(), time(), erlang:localtime(), halt(),  
exit(Reason)
```

- 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:

$1 + (2 + (3 + (4 + (5 + \text{Acc}))))$

Példa `foldl` kiértékelési sorrendjére:

$5 + (4 + (3 + (2 + (1 + \text{Acc}))))$



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**
 $[a, 'A', [65]] ++ [1+2, 2/1, 'A'] \rightsquigarrow [a, 'A', "A", 3, 2.0, 'A']$
- **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**
 $[a, 'A', [65], 'A'] -- ["A", 2/1, 'A'] \rightsquigarrow [a, 'A']$



Listanézet

- **Listanézet:** `[Kif || Minta <- Lista, Feltétel]`
`Feltétel` tetszőleges logikai kifejezés lehet. A `Minta`-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]`
- **Pitagoraszai számhármások**

```
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])].
```



- Nézzük újra a következő definíciót:

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

- ▶ Mi történik, ha megváltoztatjuk a klózek 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 olyan *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 örök 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



Örkifejezés, ör

Örkifejezés:

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

Ör:

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



Örkifejezés 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)

Örkifejezés **nem** lehet:

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



Ör: 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**

```
Minta1 [when Őr1] -> SzekvenciálisKif1;
```

```
...
```

```
Mintan [when Őr1n] -> SzekvenciálisKifn
```

```
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 őrrrel

- **if**

Őr₁ -> SzekvenciálisKif₁;

...

Őr_n -> SzekvenciálisKif_n

end.

- Kiértékelés: balról jobbra.
- Értéke: az első teljesülő őrr utáni szekvenciális kifejezés
- Ha nincs ilyen őrr, 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 fájlnev kiterjesztése: .erl`
 - ▶ Kívülről is látható függvények listája
`-export ([f1/arity, ..., fn/arity])`
 - ▶ Más modulok modulnév nélkül használható függvényeinek listája
`-import ([f1/arity, ..., fn/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

- Deklaratív programozás, funkcionális programozás
- Erlang-bevezetés
- Erlang emulátor
- Típusok
- Szekvenciális programozás
- **Típusspecifikáció**
- Kivételkezelés
- Rekord
- Rekurzív adatstruktúrák
- Gyakori könyvtári függvények
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Listák használata: futamok
- Listák rendezése
- Kifejezés kiértékelése
- Absztrakció függvényekkel, eljárásokkal



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
 $\{T_1, \dots, T_n\}$ típuskifejezés, ha $T_1, \dots, T_n$ típuskifejezések
 - ▶ Listatípus
 $[T]$ típuskifejezés, ha T típuskifejezés
 - ▶ Uniótípus
 $T_1 \mid T_2$ típuskifejezés, ha T_1 és T_2 típuskifejezések
 - ▶ Függvénytípus
 $\text{fun}(T_1, \dots, T_n) \rightarrow T$ típuskifejezés, ha $T_1, \dots, T_n$ 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

- Deklaratív programozás, funkcionális programozás
- Erlang-bevezetés
- Erlang emulátor
- Típusok
- Szekvenciális programozás
- Típusspecifikáció
- **Kivételkezelés**
- Rekord
- Rekurzív adatstruktúrák
- Gyakori könyvtári függvények
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Listák használata: futamok
- Listák rendezése
- Kifejezés kiértékelése
- Absztrakció függvényekkel, eljárásokkal



- 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

- Deklaratív programozás, funkcionális programozás
- Erlang-bevezetés
- Erlang emulátor
- Típusok
- Szekvenciális programozás
- Típusspecifikáció
- Kivételkezelés
- **Rekord**
- Rekurzív adatstruktúrák
- Gyakori könyvtári függvények
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Listák használata: futamok
- Listák rendezése
- Kifejezés kiértékelése
- Absztrakció függvényekkel, eljárásokkal



Rekord

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



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

- Deklaratív programozás, funkcionális programozás
- Erlang-bevezetés
- Erlang emulátor
- Típusok
- Szekvenciális programozás
- Típusspecifikáció
- Kivételkezelés
- Rekord
- **Rekurzív adatstruktúrák**
- Gyakori könyvtári függvények
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Listák használata: futamok
- Listák rendezése
- Kifejezés kiértékelése
- Absztrakció függvényekkel, eljárásokkal



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
-export([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

- Deklaratív programozás, funkcionális programozás
- Erlang-bevezetés
- Erlang emulátor
- Típusok
- Szekvenciális programozás
- Típusspecifikáció
- Kivételkezelés
- Rekord
- Rekurzív adatstruktúrák
- **Gyakori könyvtári függvények**
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Listák használata: futamok
- Listák rendezése
- Kifejezés kiértékelése
- Absztrakció függvényekkel, eljárásokkal



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`-vel 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)**

<code>~</code>	<code>a ~jel</code>	<code>~c</code>	az adott kódú karakter
<code>~s</code>	fűzér	<code>~f</code> , <code>~e</code> , <code>~g</code>	lebegőpontos szám
<code>~b</code> , <code>~x</code>	egész	<code>~w</code> , <code>~p</code>	Erlang-term
<code>~n</code>	újsor		

- **Példa**

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

```
abc 97 b 2.718282
```

```
ok
```

Tartalom

1 Szekvenciális Erlang

- Deklaratív programozás, funkcionális programozás
- Erlang-bevezetés
- Erlang emulátor
- Típusok
- Szekvenciális programozás
- Típusspecifikáció
- Kivételkezelés
- Rekord
- Rekurzív adatstruktúrák
- Gyakori könyvtári függvények
- **Halmazműveletek (rendezetlen listával)**
- Generikus keresőfák
- Listák használata: futamok
- Listák rendezése
- Kifejezés kiértékelése
- Absztrakció függvényekkel, eljárásokkal



Tagsági vizsgálat

`isMember` igaz, ha az adott érték eleme a halmaznak.

```
%% @spec isMember(X::any(), Ys::[any()]) -> B::bool()
%%   B igaz, ha X eleme Ys-nek
isMember(_, []) ->
    false;
isMember(X, [Y|Ys]) ->
    X==Y orelse isMember(X, Ys).
```

Megjegyzés: `orelse` lusta kiértékelésű.

Háromklózos változat:

```
isMember2(_, []) ->
    false;
isMember2(X, [X|_]) ->
    true;
isMember2(X, [_Y|Ys]) ->
    isMember2(X, Ys).
```



Új elem berakása egy halmazba, listából halmaz

`newMember` új elemet rak egy halmazba, *ha még nincs benne*.

```
%% @spec newMember(X::any(),Xs::[any()]) -> Ys::[any()]
%%   Ys az [X] és az Xs uniója
newMember(X,Xs) ->
    case isMember(X,Xs) of
        true -> Xs;
        false -> [X|Xs]
    end.
```

`listToSet` listát halmazzá alakít a duplikátumok törlésével; *nagyon rossz hatékonyságú*.

```
%% @spec listToSet(Xs::[any()]) -> Ys::[any()]
%%   Ys az Xs lista elemeinek (listaként ábrázolt) halmaz
listToSet([]) ->
    [];
listToSet([X|Xs]) ->
    newMember(X,listToSet(Xs)).
```



Öt művelet halmazokon

- Öt ismert halmazműveletet definiálunk a továbbiakban (rendezetlen listákkal ábrázolt halmazokon):
 - ▶ unió (`union`, $S \cup T$),
 - ▶ metszet (`intersect`, $S \cap T$),
 - ▶ részhalmaza-e (`isSubset`, $T \subseteq S$),
 - ▶ egyenlők-e (`isEqual`, $S = T$),
 - ▶ hatványhalmaz (`powerSet`, pS).
- Otthoni gyakorlásra: halmazműveletek megvalósítása rendezett listákkal, illetve fákkal.
A vizsgán lehetnek ilyen feladatok.



Unió, metszet

```
%% @spec union(Xs::[any()],Ys::[any()]) -> Zs::[any()]
%%   Zs az Xs és Ys halmazok uniója
union([],Ys) -> Ys;
union([X|Xs],Ys) ->
    newMember(X,union(Xs,Ys)).
```

```
%% @spec intersect(Xs::[any()],Ys::[any()]) -> Zs::[any()]
%%   Zs az Xs és Ys halmazok metszete
```

```
intersect([],_) -> [];
intersect(_,[]) -> [];
intersect([X|Xs],Ys) ->
    Zs = intersect(Xs,Ys),
    case isMember(X,Ys) of
        true ->
            [X|Zs];
        false ->
            Zs
    end.
```

```
...
intersect2([X|Xs],Ys) ->
    case isMember(X,Ys) of
        true ->
            [X|intersect2(Xs,Ys)];
        false ->
            intersect2(Xs,Ys)
    end.
```

Részhalmaz-e, egyenlők-e

```
%% @spec isSubset(Xs::[any()],Ys::[any()]) -> B::bool()  
%%   B igaz, ha Xs részhalmaza Ys-nek  
isSubset([],_) ->  
    true;  
isSubset([X|Xs],Ys) ->  
    isMember(X,Ys) andalso isSubset(Xs,Ys).
```

Megjegyzés: andalso lusta kiértékelésű.

```
%% @spec isEqual(Xs::[any()],Ys::[any()]) -> B::bool()  
%%   B igaz, ha Xs és Ys elemei azonosak  
isEqual(Xs,Ys) ->  
    isSubset(Xs,Ys) andalso isSubset(Ys,Xs).
```

A listák egyenlőségének vizsgálata ugyan beépített művelet az Erlangban, halmazokra mégsem használható, mert pl. $[3, 4]$ és $[4, 3]$ listaként különböznek, de halmazként egyenlők.



Halmaz hatványhalmaza

A hatványhalmazt előállító algoritmus vázlata:

- Az S halmaz hatványhalmazának nevezzük az S összes részalmazának a halmazát, beleértve S -t magát és az üres halmazt ($\{\}$) is.
- S hatványhalmazát úgy állítjuk elő, hogy kivesszük S -ből az x elemet, majd *rekurzív módon* előállítjuk az $S - \{x\}$ hatványhalmazát.
- Ha tetszőleges T halmazra $T \subseteq S - \{x\}$, akkor $T \subseteq S$ és $T \cup \{x\} \subseteq S$, azaz mind T , mind $T \cup \{x\}$ eleme S hatványhalmazának.
- Miközben a fentieket rekurzív módon alkalmazzuk (azaz felsoroltatjuk az $S - \{x\}$ stb. részalmazait), a *már kiválasztott* elemeket gyűjtjük. Egy-egy rekurzív lépésben a gyűjtő vagy változatlan (T), vagy kiegészül az x elemmel ($T \cup \{x\}$).



Hatványhalmaz, segédfüggvénnyel

- `pws` kétfelé ágazó rekurziót tartalmaz. Ez gyorsan a verem túlcsoordulásához vezethet.
- Az `insAll` segédfüggvény egy elemet szúr be egy listákból álló lista minden eleme elé

```
%% @spec insAll(X::any(),Yss::[[any()]],
%%           Zss)::[[any()]] -> Wss::[[any()]]
%%   Wss az Yss lista Ys elemeinek Zss elé fűzött
%%   listája, ahol minden Ys elé egy X van beszúrva
insAll(_X,[],Zss) ->
    Zss;
insAll(X,[Ys|Yss],Zss) ->
    insAll(X,Yss,[X|Ys]|Zss).
```



Hatványhalmaz, segédfüggvénnyel (folyt.)

- `powerSet insAll`-t használó, lineárisan rekurzív processzt előállító változata

```
powerSet21([]) -> [[]];  
powerSet21([X|Xs]) ->  
    Pws = powerSet21(Xs),  
    Pws ++ insAll(X,Pws, []).
```

- `powerSet insAll`-t használó, iteratív processzt előállító változata

```
powerSet22([]) -> [[]];  
powerSet22([X|Xs]) ->  
    Pws = powerSet22(Xs),  
    insAll(X,Pws,Pws).
```

Példa:

```
powerSet21([1,2,3,4,5,6,7,8,9,10,11,  
            12,13,14,15,16,17,19,20,21,22]).
```

Hatványhalmaz, kétfelé ágazó rekurzióval

- A `pws` függvény a `Zs` argumentumban gyűjti a halmaz *már kiválasztott* elemeit; kezdetben üres.
- $\text{pws}(Xs, Zs) = \{S \cup Zs \mid S \subseteq Xs\}$, azaz az $Xs \cup Zs$ azon részhalmazainak a listája, amelyek teljes egészében tartalmazzák a `Zs` halmazt.

```
%% @spec pws(Xs::[any()], Zs::[any()]) -> Rs::[any()]
%%   Rs azoknak a halmazoknak a listája, amelyeket Xs
%%   egy-egy részhalmaza és Zs uniója alkot
pws([], Zs) -> [lists:reverse(Zs)]; %% [Zs]
pws([X|Xs], Zs) -> pws(Xs, Zs) ++ pws(Xs, [X|Zs]).
```

- A második klózban a `pws(Xs, Zs)` rekurzív hívás állítja elő az $S - \{x\}$ hatványhalmazát (hiszen `[X|Xs]` felel meg `S`-nek), azaz az összes olyan halmazt, amely az `x`-et nem tartalmazza.
- A `pws(Xs, [X|Zs])` rekurzív hívás a `Zs`-ben gyűjti az `X`-eket, azaz előállítja az összes olyan halmazt, amely az `x`-et tartalmazza.



Hatványhalmaz, kétfelé ágazó rekurzióval (folyt.)

`powerSet1`-nek már csak megfelelő módon hívnia kell `pws`-t:

```
%% @spec powerSet1(Xs::[any()] -> Yss::[[any()]])
%%   Yss az Xs halmaz hatványhalmaza
powerSet1(Xs) ->
    pws(Xs, []).
```

Példa:

```
powerSet1([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,19,20]).
```



Tartalom

1 Szekvenciális Erlang

- Deklaratív programozás, funkcionális programozás
- Erlang-bevezetés
- Erlang emulátor
- Típusok
- Szekvenciális programozás
- Típusspecifikáció
- Kivételkezelés
- Rekord
- Rekurzív adatstruktúrák
- Gyakori könyvtári függvények
- Halmazműveletek (rendezetlen listával)
- **Generikus keresőfák**
- Listák használata: futamok
- Listák rendezése
- Kifejezés kiértékelése
- Absztrakció függvényekkel, eljárásokkal



Generikus keresőfák Erlangban

Lásd a leírást a

<http://dp.iit.bme.hu/dp08a/gtree.pdf>,

a futtatható példaprogramokat a

<http://dp.iit.bme.hu/dp08a/gtree.erl>
fájlban.



Tartalom

1 Szekvenciális Erlang

- Deklaratív programozás, funkcionális programozás
- Erlang-bevezetés
- Erlang emulátor
- Típusok
- Szekvenciális programozás
- Típusspecifikáció
- Kivételkezelés
- Rekord
- Rekurzív adatstruktúrák
- Gyakori könyvtári függvények
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- **Listák használata: futamok**
- Listák rendezése
- Kifejezés kiértékelése
- Absztrakció függvényekkel, eljárásokkal



Futamok előállítása (1)

- Futam: olyan lista, amelynek szomszédos elemei adott feltételnek megfelelnek.
- A feltételt az előző és az aktuális elemre alkalmazandó *predikátumként* adjuk át a futamot előállító függvénynek.
- Feladat: írjunk olyan Erlang-függvényt, amely egy lista egymás utáni elemeiből képzett futamok listáját adja eredményül – az elemek eredeti sorrendjének megőrzésével.
- Az első változatban egy-egy segédfüggvényt írunk egy lista *első* (prefix) *futamának*, valamint a *maradéklistának* az előállítására.



Futamok előállítása (2)

- A `futam1` segédfüggvénynek két argumentuma van: az első egy predikátum, amely a kívánt feltételt megvalósítja, a második pedig egy pár.
A pár első tagja az előző elem, a második tagja pedig az a lista, amelynek az előző elemmel induló futamát kell `futam1`-nek előállítania.
- A `maradek1` segédfüggvény két argumentuma azonos `futam1` argumentumaival. Eredményül azt a listát kell visszaadnia, amelyet az első futam leválasztásával állít elő a pár második tagjaként átadott listából.



Futamok előállítása két segédfüggvénnyel

```
%% @spec futam:futamok1(Pred::pred(),List::[elem()]) ->
%%                                     Lists::[[elem()]]
%% @type pred() = (elem(), elem()) -> bool()
%% @type elem() = any()
%%   Lists a List szomszédos elemeiből álló,
%%   Pred-et kielégítő futamok listája
futamok1(_P,[]) ->
    [];
futamok1(P,[X|Xs]) ->
    Fs = futaml(P,X,Xs),
    Ms = maradekl(P,X,Xs),
    case Ms of
        [] ->
            [Fs];
        Zs ->
            [Fs|futamok1(P,Zs)]
    end.
```



Futamok előállítása két segédfüggvénnyel (folyt.)

```
futam1(_P,X,[]) -> [X];  
futam1(P,X,[Y|Ys]) ->  
    case P(X,Y) of  
        true -> [X|futam1(P,Y,Ys)];  
        false -> [X]  
    end.
```

```
maradek1(_P,_X,[]) -> [];  
maradek1(P,X,[Y|Ys]=YYs) ->  
    case P(X,Y) of  
        true -> maradek1(P,Y,Ys);  
        false -> YYs  
    end.
```

Példa

```
11> futam:futamok1(fun(A,B) -> A<B end,  
                    [1,3,9,5,7,2,5,9,1,6,0,0,3,5,6,2])).  
[[1,3,9],[5,7],[2,5,9],[1,6],[0],[0,3,5,6],[2]]
```

A futamok1 függvény hatékonyságáról

Hatékonyságot rontó tényezők

- 1 `futamok1` kétszer megy végig a listán: először `futam1`, azután `maradek1`;
- 2 a három függvény közül csak `maradek1` jobbrekurzív.

Javítási lehetőségek

- 1 `futam1` javított változata egy párt adjon eredményül, ennek első tagja legyen a `futam`, második tagja pedig a `maradék`;
- 2 `futam1` javított változata legyen jobbrekurzív és használjon akkumulátort;
- 3 a `case Ms of [] -> [Fs]` programrész elhagyható: a rekurzió egy hívással később mindenképpen leáll;
- 4 `futamok1` javított változata is legyen jobbrekurzív és használjon akkumulátort (ezt később oldjuk meg).



Futamok előállítása egy segédfüggvénnyel

```
futamok2(_P, []) -> [];  
futamok2(P, [X|Xs]) ->  
    {Fs, Ms} = futamok2(P, X, Xs, []),  
    [Fs|futamok2(P, Ms)].  
  
futam2(_P, X, [], Zs) -> {lists:reverse([X|Zs]), []};  
futam2(P, X, [Y|Ys]=YYs, Zs) ->  
    case P(X, Y) of  
        true -> futam2(P, Y, Ys, [X|Zs]);  
        false -> {lists:reverse([X|Zs]), YYs}  
    end.
```

Példa

```
19> futam:futamok2(fun(A,B) -> A>=B end,  
                    [1,3,9,5,7,2,5,9,1,6,0,0,3,5,6,2])).  
[[1],[3],[9,5],[7,2],[5],[9,1],[6,0,0],[3],[5],[6,2]]
```



Futam és futamok gyűjtése egyszerre

Javítás: `futamok2`-t jobbrekurzívvá tesszük.

```
futamok3(_P, []) -> [];  
futamok3(P, [X|Xs]) -> futamok3(P, X, Xs, [], []).  
  
futamok3(_P, X, [], Zs, Zss) ->  
    lists:reverse([lists:reverse([X|Zs])|Zss]);  
futamok3(P, X, [Y|Ys], Zs, Zss) ->  
    case P(X, Y) of  
        true ->  
            futamok3(P, Y, Ys, [X|Zs], Zss);  
        false ->  
            futamok3(P, Y, Ys, [], [lists:reverse([X|Zs])|Zss])  
    end.
```

Példa

```
25> futam:futamok3(fun(A,B) -> 2*A>B end,  
    [1,3,9,5,7,2,5,9,1,6,0,0,3,5,6,2]).  
[[1],[3],[9,5,7,2],[5,9,1],[6,0],[0],[3,5,6,2]]
```



Példák a futam-függvények alkalmazására

```
futam:t4(0) -> futam1(fun(A,B) -> A=<B end, 1,  
    [9,19,3,4,24,34,4,11,45,66,13,45,66,99]);  
%%% [1,9,19].
```

```
futam:t4(1) -> maradek1(fun(A,B) -> A=<B end, 1,  
    [9,19,3,4,24,34,4,11,45,66,13,45,66,99]);  
%%% [3,4,24,34,4,11,45,66,13,45,66,99].
```

```
futam:t4(2) -> futamok1(fun(A,B) -> A=<B end,  
    [1,9,19,3,4,24,34,4,11,45,66,15,45,66,99]);  
%%% [[1,9,19], [3,4,24,34], [4,11,45,66], [15,45,66,99]].
```

```
futam:t4(3) -> futamok1(fun(A,B) -> A=<B end, [299,1]);  
%%% [[299], [1]].
```

```
futam:t4(4) -> futamok1(fun(A,B) -> A=<B end, [299]);  
%%% [[299]].
```

```
futam:t4(5) -> futamok1(fun(A,B) -> A=<B end, []).  
%%% [].
```



Tartalom

1 Szekvenciális Erlang

- Deklaratív programozás, funkcionális programozás
- Erlang-bevezetés
- Erlang emulátor
- Típusok
- Szekvenciális programozás
- Típusspecifikáció
- Kivételkezelés
- Rekord
- Rekurzív adatstruktúrák
- Gyakori könyvtári függvények
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Listák használata: futamok
- **Listák rendezése**
- Kifejezés kiértékelése
- Absztrakció függvényekkel, eljárásokkal



Beszűrő rendezés

`insl` az `X` elemet a megfelelő helyre szúrja be az `Ys` listában:

```
%% @spec insl(X::any(), Ys::[any()]) -> Zs::[any()]
%% @pre: Ys az =< reláció szerint rendezve van
%%      Zs az =< reláció alapján beszűrt X-szel bővített Ys
insl(X, []) -> [X];
insl(X, [Y|Ys]) when X=<Y ->
    [X, Y|Ys];
insl(X, [Y|Ys]) ->
    [Y|insl(X, Ys)].
```

`inssortl1`-gyel rekurzívan rendezzük a lista maradékát; végrehajtási ideje $O(n^2)$

```
inssortl1([]) -> [];
inssortl1([X|Xs]) ->
    insl(X, inssortl1(Xs)).
```



Beszűrő rendezés, generikus változat

Az `inssort` függvényt generikussá tesszük: az `ins` függvényt paraméterként adjuk át

```
%% @spec inssort12(F:ins(),Xs::[any()]) -> Zs::[any()]
%% @type ins() = (any(),[any()]) -> [any()]
%%   Zs az F beszűrő függvénnnyel az =<
%%   reláció szerint rendezett Ys
inssort12(_F,[]) ->
    [];
inssort12(F,[X|Xs]) ->
    F(X,inssort12(F,Xs)).
```



Beszűrő rendezés, generikus változat (folyt.)

„Generikusabb”, ha a rendezési relációt adjuk át paraméterként:

```
%% @spec ins2(F::pred(),X::any(),Ys::[any()]) ->
%%                                     Zs::[any()]
%% @type pred() = (any(), any()) -> bool()
%% @pre Ys az F reláció szerint rendezve van
%%     Zs az F reláció alapján beszűrt X-szel bővített Ys
ins2(_F,X,[]) ->
    [X];
ins2(F,X,[Y|Ys]) ->
    case F(X,Y) of
        true ->
            [X,Y|Ys];
        false ->
            [Y|ins2(F,X,Ys)]
    end.
```



Beszűrő rendezés, generikus változat (folyt.)

```
insort21(_F, []) ->
    [];
insort21(F, [X|Xs]) ->
    ins2(F, X, insort21(F, Xs)) .
```

Jobbrekurzív változat

```
insort22(F, Zs) ->
    insort22(F, [], Zs) .

insort22(_F, [], Zs) ->
    Zs;
insort22(F, [X|Xs], Zs) ->
    insort22(F, Xs, ins3(F, X, Zs)) .
```

Szükségünk van hozzá `ins3`-ra, `ins` jobbrekurzív változatára –
meghagyjuk gyakorló feladatnak.



Beszúró rendezés, példák

```
ti11()-> rend:inssort11([9,7,8,3,1,5]).
```

```
ti12()->  
  rend:inssort12(fun(A,Ls)->ins1(A,Ls) end,[9,7,8,3,1,5]).
```

```
ti21()->  
  rend:inssort21(fun(A,B)->A<B end,[9,7,8,3,1,5]).
```

```
ti22()->  
  rend:inssort22(fun(A,B)->A>=B end,[9,7,8,3,1,5]).
```

```
ti33()->  
  rend:inssort21(fun(A,B)->A<B end,[4.24, 4.1, 5.67]).
```

```
ti34()->  
  rend:inssort22(fun(A,B)->A>=B end,[4, 4, 5, 1, 0, 8]).
```

```
ti35()-> rend:inssort22(fun(A,B)->A<B end,"qwer").
```



Beszűrő rendezés *fold*-dal

```
inssortR(F,Xs) ->  
  lists:foldr(fun(A,Ls) -> ins2(F,A,Ls) end,[],Xs).
```

A foldl-t alkalmazó változat jobbrekurzív:

```
inssortL(F,Xs) ->  
  lists:foldl(fun(A,Ls) -> ins3(F,A,Ls) end,[],Xs).
```

```
ti31()->  
  rend:inssortR(fun(A,B)->A<B end,[9,7,8,3,1,5])).
```

```
ti32()->  
  rend:inssortR(fun(A,B)->A>=B end,[9,7,8,3,1,5])).
```



Generikus kiválasztó rendezés

```
%% @spec selsort(F:pred(),Xs::[any()]) -> Zs::[any()]
%% @type pred() = (any(), any()) -> bool()
%%   Zs az F reláció szerint rendezett Xs
selsort(F,Xs) ->
    ssort(F,Xs,[]).

%% @spec ssort(F::pred(),Xs::[any()],Ys::[any()]) ->
%%                                     Zs::[any()]
%% @type pred() = (any(), any()) -> bool()
%%   Zs az F szerinti sorrendben az Ys elé fűzött Xs
ssort(_F,[],Ws) ->
    Ws;
ssort(F,[X|Xs],Ws) ->
    {M,Ms} = maxSelect(F,X,Xs,[]),
    ssort(F,Ms,[M|Ws]).
```



Generikus kiválasztó rendezés (folyt.)

```
%% @spec maxSelect(F::pred(),X::any(),Ys::[any()],
%%               Zs::[any()]) -> {M::any(),Ms::[any()]}
%% @type pred() = (any(), any()) -> bool()
%%   M az [X|Ys] lista F szerinti legnagyobb eleme, Ms az
%%   [X|Ys] többi eleméből és a Zs elemeiből álló lista
maxSelect(_F,X,[],Zs) ->
    {X,Zs};
maxSelect(F,X,[Y|Ys],Zs) ->
    maxSelect(F,max(F,X,Y),Ys,[min(F,X,Y)|Zs]).

max(F,X,Y) ->
    case F(X,Y) of
        true ->
            X;
        false ->
            Y
    end.

min(F,X,Y) ->
    case F(X,Y) of
        true ->
            Y;
        false ->
            X
    end.
```



Generikus kiválasztó rendezés, példák

```
ts1() ->
    rend:selsort (fun (A,B) -> A=<B end,
                  [1,2,3,4,5,6,7,8,9]) .
```

```
ts2() ->
    rend:selsort (fun (A,B) -> A=<B end,
                  [9,8,7,6,5,4,3,2,1]) .
```

```
ts3() ->
    rend:selsort (fun (A,B) -> A=<B end,
                  [4.5,6.7,3.6,4.3,1.2,0.9,8.9,9.8,2.0]) .
```

```
ts4() ->
    rend:selsort (fun (A,B) -> A=<B end,
                  "Ej mi a ko tyukanyo") .
```



Gyorsrendezés

A *Quicksort* algoritmus:

$qs [] = []$

$qs(m::xs) = qs([...,x_i,...] \mid x_i \leq m) @ [m] @ qs([...,x_j,...] \mid x_j > m),$

ahol $x_i, x_j \in xs$

Egy n elemű sorozatot $O(n \cdot \log n)$ lépésben rendez.

```
qsort1([]) ->
    [];
qsort1([X|Xs]) ->
    Ls = [E | E <- Xs, E <= X],
    Rs = [E | E <- Xs, E > X],
    qsort1(Ls) ++ [X] ++ qsort1(Rs).
```



Gyorsrendezés, módosított változat

Ha azonos értékű elemek sorozatban fordulnak elő a rendezendő listában, gyorsabb a következő változat

```
qsort2([]) ->
    [];
qsort2([X|Xs]) ->
    Ls = [E || E <- Xs, E<X],
    Ms = [E || E <- Xs, E==X],
    Rs = [E || E <- Xs, E>X],
    qsort2(Ls) ++ [X|Ms] ++ qsort2(Rs).
```



Gyorsrendezés akkumulátorral

```
qsort3(Xs) ->
    qsort3(Xs, []).

%% @spec qsort3(Xs::[any()],Zs::[any()]) -> Ws::[any()]
%%   Ws az =< reláció szerint rendezett Xs a Zs elé fűzve
qsort3([],Zs) ->
    Zs;
qsort3([X|Xs],Zs) ->
    Ls = [E || E <- Xs, E<X],
    Ms = [E || E <- Xs, E==X],
    Rs = [E || E <- Xs, E>X],
    qsort3(Ls,[X|Ms++qsort3(Rs,Zs)]).
```

Jobb az előzőeknél?



Gyorsrendezés, generikus változatok

```
%% @spec gqsort2(F:pred(),Xs::[any()],Zs::[any()]) ->
%%                                     Ws::[any()]
%% @type pred() = (any(), any()) -> bool()
%%   Ws az F reláció szerint rendezett Ys a Zs elé fűzve
gqsort2(_F,[]) -> [];
gqsort2(F,[X|Xs]) ->
    Ls = [E || E <- Xs, F(E,X)],
    Ms = [E || E <- Xs, E==X],
    Rs = [E || E <- Xs, F(X,E)],
    gqsort2(F,Ls) ++ [X|Ms] ++ gqsort2(F,Rs).

gqsort3(F,Xs) -> gqsort3(F,Xs,[]).

gqsort3(_F,[],Zs) -> Zs;
gqsort3(F,[X|Xs],Zs) ->
    Ls = [E || E <- Xs, F(E,X)],
    Ms = [E || E <- Xs, E==X],
    Rs = [E || E <- Xs, F(X,E)],
    gqsort3(F,Ls,[X|Ms++gqsort3(F,Rs,Zs)]).
```



Gyorsrendezés, példák

```
tq1() ->  
    rend:qsort1("abrakadabra").
```

```
tq2() ->  
    rend:qsort2("abrakadabra").
```

```
tq3() ->  
    rend:qsort3("abrakadabra").
```

```
tq4() ->  
    rend:gqsort2(fun(A,B) -> A<B end, "abrakadabra").
```

```
tq5() ->  
    rend:gqsort3(fun(A,B) -> A<B end, "abrakadabra").
```



Összefésülő rendezések

Kell hozzá egy segédfüggvény két lista megfelelő sorrendű összefuttatására

```
%% @spec merge(Xs::[any()],Ys::[any()]) -> Zs::[any()]
%%   az Xs és az Ys <= reláció szerinti összefésülése Zs
merge([],Ys) ->
    Ys;
merge(Xs,[]) ->
    Xs;
merge([X|Xs]=XXs,[Y|Ys]=YYs) ->
    if
        X=<Y    ->
            [X|merge(Xs,YYs)];
        true ->
            [Y|merge(XXs,Ys)]
    end.
```

Könyvtári változata `lists:merge/2`, generikus verziója `lists:merge/3`. Más `merge` verziók is vannak a `lists` modulban.



Fölről lefelé haladó összefésülő rendezés

A fölről lefelé haladó összefésülő rendezés (*top-down merge sort*) akkor hatékony, ha közel azonos hosszúságú az a két lista, amelyekre a rendezendő listát szétszedjük.

```
%% @spec tmsort(Xs::[any()]) -> Zs::[any()]
```

```
%%   Zs az =< reláció szerint rendezett Ys
```

```
tmsort(Xs) ->
  H = length(Xs),
  K = H div 2,
  if
    H>1 ->
      merge(tmsort(take(Xs,K)),
            tmsort(drop(Xs,K)));
    true ->
      Xs
  end.
```

```
take(Xs,K) ->
  lists:sublist(Xs,K).
```

```
drop(Xs,K) ->
  lists:nthtail(K,Xs).
```

A legrosszabb esetben $O(n \cdot \log n)$ lépésre van szükség.



Alulról fölfelé haladó összefésülő rendezés

A *bottom-up merge sort* legegyszerűbb változata az eredeti k hosszú listát k darab egyelemű listára bontja, majd a szomszédos listákat összefuttatja, így 2, 4, 8, 16 stb. elemű listákat állít elő.

R. O'Keefe algoritmus (1982) lépésről lépésre futtatja össze az egyforma hosszú részlistákat, de csak a végén rendezi az összeset.

A	B	C	D	E	F	G	H	I	J	K
AB		C	D	E	F	G	H	I	J	K
AB		CD		E	F	G	H	I	J	K
ABCD				E	F	G	H	I	J	K
ABCD				EF		G	H	I	J	K
ABCD				EF		GH		I	J	K
ABCD				EFGH				I	J	K
ABCDEFGH								I	J	K
ABCDEFGH								IJ		K
...										

A példában az összefuttatott részlistákat *egymás mellé írással* jelöljük.

Alulról fölfelé haladó összefésülő rendezés (folyt.)

```
%% @spec bmsort(Xs::[any()]) -> Zs::[any()]  
%%   Zs az =< reláció szerint rendezett Ys  
bmsort(Xs) ->  
    sorting(Xs, [], 0).
```

A `sorting` segédfüggvény

- első argumentuma a rendezendő lista,
- második argumentuma a már rendezett részlistákból álló lista akkumulátora,
- harmadik argumentuma az adott lépésben feldolgozandó elem sorszáma.



Alulról fölfelé haladó összefésülő rendezés (folyt.)

```
%% @spec sorting(Xs::[any()], Lss::[[any()]], K) ->
%%                                     Zs::[any()]
%% @pre K>=0
%%   Zs a még rendezetlen Xs és a már K rendezett
%%   részlistát tartalmazó Lss összefűzésének eredménye
sorting([X|Xs], Lss, K) ->
    sorting(Xs, mergepairs([X] | Lss), K+1), K+1);
sorting([], Lss, _K) ->
    hd(mergepairs(Lss, 0)).
```

- Ha a rendezendő lista (Xs) még nem fogyott el, soron következő eleméből `sorting` egyelemű listát ($[X]$) képez, és ezt a már rendezett részlisták listája (Lss) elé fűzve meghívja a `mergepairs` segédfüggvényt.
- Ha a rendezendő lista kiürült, `sorting` a kétszintű lista egyetlen elemét, a rendezett Lss listát adja eredményül – `mergepairs` speciális ($K==0$!) meghívásával.



Alulról fölfelé haladó összefésülő rendezés (folyt.)

```
%% @spec mergepairs(Lss::[[any()]],K) -> Zss::[[any()]]
%% @pre K>=0
%%     Zss az Lss-nek olyan változata, amely az Lss első
%%     két részlistája helyett, ha egyforma a hosszuk, az
%%     összefuttatásuk eredményét tartalmazza
mergepairs(LLlss=[L1s,L2s|Lss],K) ->
    %% legalább kételemű a lista
    if K rem 2 == 1 ->
        LLlss;
    true ->
        mergepairs([merge(L1s,L2s)|Lss],K div 2)
    end;
mergepairs(Lss,_K) ->
    Lss. % egyelemű a lista
```



Alulról fölfelé haladó összefésülő rendezés (folyt.)

- `mergepairs` az argumentumként átadott lista két egyforma hosszú bal oldali részlistáját fűzi egybe, feltéve persze, hogy vannak ilyenek. K az átadott elem sorszáma.
- `mergepairs` egyetlen listában gyűjti a már összefuttatott részlistákat. Az éppen átadott elem K sorszámából dönti el, hogy mit kell csinálnia a következő részlistával.
- Ha K páratlan, `mergepairs` a listát változtatás nélkül adja vissza, ha páros, akkor az `LLSs` lista elején álló két, egyforma hosszú listát egyetlen rendezett listává futtatja össze.
- $K==0$ -ra `mergepairs` az összes listák listáját olyan listává futtatja össze, amelynek egyetlen eleme maga is lista.
- A legrosszabb esetben $O(n \cdot \log n)$ lépésre van szükség.



Alulról fölfelé haladó összefésülő rendezés (folyt.)

A függvények működését egy példán is bemutatjuk. A kezdőhívás legyen

```
bmsort([1,2,3,4,5,6,7,8,9])  
    ---> sorting([1,2,3,4,5,6,7,8,9], [], 0)  
  
sorting([X|Xs], Lss, K) ->  
    sorting(Xs, mergepairs([[X]|Lss], K+1), K+1);  
sorting([], Lss, _K) ->  
    hd(mergepairs(Lss, 0)).
```

Amíg `sorting` első argumentuma a nem üres `[X|Xs]` lista, `sorting` saját magát hívja meg. A rekurzív hívás

- 1. argumentuma a lépésenként egyre rövidülő `Xs` lista,
- 2. argumentuma a `mergepairs([[X]|Lss], K+1)` függvényalkalmazás eredménye, ahol kezdetben `Lss == []`,
- 3. argumentuma a már feldolgozott listaelemek száma (`K+1`).

Alulról fölfelé haladó összefésülő rendezés (folyt.)

- A következő dián táblázatos elrendezés mutatja
 - ▶ `mergепairs` mindkét argumentumát,
 - ▶ a rekurzív `sorting` hívás itt J -vel jelölt 3. argumentumát, $K+1$ -et, és
 - ▶ bináris számként K -t lépésről lépésre.
- A `sorting` függvény hívja `mergепairs`-t azokban a sorokban, amelyekben a J új értéket vesz föl, a többi helyen `mergепairs` hívása rekurzív.
- Ne feledjük, hogy `mergепairs`-nek listák listája az első argumentuma.
- A táblázat utolsó oszlopa a későbbi magyarázatra hivatkozik.
- Vegyük észre, hogy kapcsolat van az `LLLSs` első eleme utáni listaelemek hossza és a K bitjei között! Ha K valamelyik bitje 1, akkor (balról jobbra haladva) az `LLLSs` megfelelő listaelemének a hossza az adott bit helyiértékével egyenlő. A 0 értékű biteknek megfelelő listaelemek „hiányoznak” `LLLSs`-ből.



Alulról fölfelé haladó összefésülő rendezés (folyt.)

LLLss	N	J	K	
[[1]]	1	1	0	m1
[[2],[1]]	2	2	1	m2
[[1,2]]	1			m3
[[3],[1,2]]	3	3	10	m3
[[4],[3],[1,2]]	4	4	11	m2
[[3,4],[1,2]]	2			m2
[[1,2,3,4]]	1			m3
[[5],[1,2,3,4]]	5	5	100	m3
[[6],[5],[1,2,3,4]]	6	6	101	m2
[[5,6],[1,2,3,4]]	3			m3
[[7],[5,6],[1,2,3,4]]	7	7	110	m3
[[8],[7],[5,6],[1,2,3,4]]	8	8	111	m2
[[7,8],[5,6],[1,2,3,4]]	4			m2
[[5,6,7,8],[1,2,3,4]]	2			m2
[[1,2,3,4,5,6,7,8]]	1			m3
[[9],[1,2,3,4,5,6,7,8]]	9	9	1000	m3
[[9],[1,2,3,4,5,6,7,8]]	0	0		m4
[[1,2,3,4,5,6,7,8,9]]				



Alulról fölfelé haladó összefésülő rendezés (folyt.)

- m1:** Az argumentumként átadott listának egyetlen eleme van (maga is lista), ezért az argumentumot `mergepairs` második klóza változtatás nélkül visszaadja az őt hívó `sorting`-nak.
- m2:** N páros, ez azt jelzi, hogy az argumentumként átadott lista első két eleme egyforma hosszú lista, amelyeket `merge` egyetlen rendezett listává futtat össze, majd az eredménnyel `mergepairs` első klóza meghívja saját magát.
- m3:** N páratlan, ez azt jelzi, hogy az argumentumként átadott lista első két eleme nem egyforma hosszú lista, ezért az argumentumot `mergepairs` első klóza változtatás nélkül visszaadja az őt hívó `sorting`-nak.
- m4:** $N==0$ azt jelenti, hogy az összes listák listáját olyan listává kell összefuttatni, amelynek egyetlen lista az eleme.

Simarendezés

Az applikatív simarendezés (*smooth sort*) algoritmus a O'Keefe alulról fölfelé haladó rendezéséhez hasonló, de nem egyforma hosszú listákat, hanem növekvő *futamokat* állít elő.

Ha a futamok száma n -től, a lista hosszától független, azaz a lista majdnem rendezve van, akkor az algoritmus végrehajtási ideje $O(n)$, és a legrosszabb esetben is legfeljebb csak $O(n \cdot \log n)$.



Simarendezés (folyt.)

```
%% @spec nextrun(Run::[any()],Xs::[any()]) ->
%%      {Rs::[any()],Ms::[any()]}
%%      Rs az Xs egy, a < reláció szerint növekvő
%%      futama a Run elé fűzve, Ms pedig az Xs maradéka
nextrun(Run,[X|Xs]) ->
    if X < hd(Run) ->
        {lists:reverse(Run),[X|Xs]};
    true ->
        nextrun([X|Run],Xs)
    end;
nextrun(Run,[]) ->
    {lists:reverse(Run),[]}.
```

- `nextrun` eredménye egy pár, amelynek első tagja a futam (egy növekvő számsorozat), a második tagja pedig a rendezendő lista maradéka.
- A futam csökkenő sorrendben bővül, kilépéskor a futamot meg kell fordítani.



Simarendezés (folyt.)

`msorting` a futamokat ismételten előállítja és összefuttatja

```
%% @spec msorting(Xs::[any()], Lss::[[any()]], K) ->
                                         Zs::[any()]

%% @pre K >= 0
%%   Zs a még rendezetlen Xs és a már K rendezett
%%   részlistát tartalmazó Lss összefűzésének eredménye
msorting([X|Xs], Lss, K) ->
    {Run, Tail} = nextrun([X], Xs),
    msorting(Tail, mergepairs([Run|Lss], K+1), K+2);
msorting([], Lss, _K) ->
    hd(mergepairs(Lss, 0)).

%% @spec msort(Xs::[any()]) -> Zs::[any()]
%%   Zs az =< reláció szerint rendezett Ys
msort(Xs) ->
    msorting(Xs, [], 0).
```



Alulról fölfelé haladó összefésülő rendezés: példák

```
tm1() ->  
    rend:tmsort("abrakadabra").
```

```
tm2() ->  
    rend:bmsort("abrakadabra").
```

```
tm3() ->  
    rend:smsort("abrakadabra").
```



T-D összefésülő rendezés, generikus változat

```
tmsort(F,Xs) ->
    H = length(Xs),
    K = H div 2,
    if
        H>1 ->
            lists:merge(F,tmsort(F,take(Xs,K)),
                        tmsort(F,drop(Xs,K)));
        true ->
            Xs
    end.
```



B-U összefésülő rendezés, generikus változat

```
bmsort(F,Xs) ->
    sorting(F,Xs,[],0).

sorting(F,[X|Xs],Lss,K) ->
    sorting(F,Xs,mergepairs(F,[X|Lss],K+1),K+1);
sorting(F,[],Lss,_K) ->
    hd(mergepairs(F,Lss,0)).

mergepairs(F,LLLss=[L1s,L2s|Lss],K) ->
    %% legalább kételemű a lista
    if K rem 2 == 1 ->
        LLLss;
    true ->
        mergepairs(F,[lists:merge(F,L1s,L2s)|Lss],
                    K div 2)

end;

mergepairs(_F,Lss,_K) ->
    Lss. % egyelemű a lista
```

Simarendezés, generikus változat

```
nextrun(F, Run, [X|Xs]) ->
    %%! if F(X,hd(Run)) ->
    case F(X,hd(Run)) of
        true->
            {lists:reverse(Run), [X|Xs]};
        false ->
            nextrun(F, [X|Run], Xs)
    end;
nextrun(_F, Run, []) -> {lists:reverse(Run), []}.

smsorting(F, [X|Xs], Lss, K) ->
    {Run, Tail} = nextrun(F, [X], Xs),
    smsorting(F, Tail, mergepairs(F, [Run|Lss], K+1), K+2);
smsorting(F, [], Lss, _K) ->
    hd(mergepairs(F, Lss, 0)).

smsort(F, Xs) ->
    smsorting(F, Xs, [], 0).
```



Generikus összefésülő rendezések: példák

```
tgm11() ->  
    rend:tmsort(fun(A,B) -> A<B end, "abrakadabra").
```

```
tgm12() ->  
    rend:tmsort(fun(A,B) -> A>B end, "abrakadabra").
```

```
tgm21() ->  
    rend:bmsort(fun(A,B) -> A<B end, "abrakadabra").
```

```
tgm22() ->  
    rend:bmsort(fun(A,B) -> A>B end, "abrakadabra").
```

```
tgm31() ->  
    rend:smsort(fun(A,B) -> A<B end, "abrakadabra").
```

```
tgm32() ->  
    rend:smsort(fun(A,B) -> A>B end, "abrakadabra").
```



Futási idők mérése: segédfüggvények

```
randlist(N,Max) ->  
    randlist(N,Max, []).
```

```
randlist(0,_Max,Zs) -> Zs;  
randlist(N,Max,Zs) ->  
    randlist(N-1,Max,[random:uniform(Max) | Zs]).
```

```
randlist(N) ->  
    randlist(N,1000000).
```

```
runlist(0,_Max,Zs) -> Zs;  
runlist(N,Max,Zs) ->  
    runlist(N-1,Max,  
            lists:seq(1,random:uniform(Max))++Zs).
```

```
runlist(N) ->  
    runlist(N,100, []).
```



Futási idők mérése

```
runtime(S,F,Xs) ->  
    T1 = now(),  
    F(Xs),  
    T2 = now(),  
    T = timer:now_diff(T2,T1),  
    io:fwrite("~w. Length: ~w, time: ~w ms~n",  
              [S,length(Xs),T div 1000]).
```



Futási idők mérése: példák

```
measure() ->
  R3000 = randlist(3000),
  Rr70  = runlist(70),
  Lt = fun(A,B) -> A =< B end,
  Gt = fun erlang:'>=' /2,
  runtime(inssortl1,fun inssortl1/1,R3000),
  runtime(inssortl2,fun(Xs) ->
                                inssortl2(fun ins1/2, Xs) end,R3000),
  runtime(inssort21,fun(Xs) -> inssort21(Lt, Xs) end,R3000),
  runtime(inssort22,fun(Xs) -> inssort22(Gt, Xs) end,R3000),
  runtime(inssortR,fun(Xs) -> inssortR(Lt, Xs) end,R3000),
  runtime(inssortL,fun(Xs) -> inssortL(Lt, Xs) end,R3000),
  runtime(selsort,fun(Xs) -> selsort(Lt, Xs) end,R3000),
  runtime(qsort1,fun(Xs) -> qsort1(Xs) end,R3000),
  runtime(qsort2,fun(Xs) -> qsort2(Xs) end,R3000),
  runtime(qsort3,fun(Xs) -> qsort3(Xs) end,R3000),
  runtime(gqsort2,fun(Xs) -> gqsort2(Lt,Xs) end,R3000),
  runtime(gqsort3,fun(Xs) -> gqsort3(Lt,Xs) end,R3000),
```



Futási idők mérése: példák (folyt.)

```
runtime(tmsort, fun(Xs)      -> tmsort(Xs) end, R3000),
runtime(bmsort, fun(Xs)      -> bmsort(Xs) end, R3000),
runtime(smsort, fun(Xs)      -> smsort(Xs) end, R3000),
runtime(smsort, fun(Xs)      -> smsort(Xs) end, Rr70),
runtime(gtmsort, fun(Xs)     -> tmsort(Lt, Xs) end, R3000),
runtime(gbmsort, fun(Xs)     -> bmsort(Lt, Xs) end, R3000),
runtime(gsmsort, fun(Xs)     -> smsort(Lt, Xs) end, R3000),
runtime(gsmsort, fun(Xs)     -> smsort(Lt, Xs) end, Rr70),
runtime('lists:sort/1', fun lists:sort/1, R3000),
runtime('lists:sort/2', fun(Xs) ->
                                lists:sort(Lt, Xs) end, R3000),
runtime('lists:usort/1', fun lists:usort/1, R3000),
runtime('lists:usort/2', fun(Xs) ->
                                lists:usort(Lt, Xs) end, R3000).
```



Futási idők mérése: eredmények

```
2> rend:measure().  
inssort11. Length: 3000, time: 418 ms  
inssort12. Length: 3000, time: 414 ms  
inssort21. Length: 3000, time: 747 ms  
inssort22. Length: 3000, time: 733 ms  
inssortR. Length: 3000, time: 728 ms  
inssortL. Length: 3000, time: 699 ms  
selsort. Length: 3000, time: 3574 ms  
qsort1. Length: 3000, time: 13 ms  
qsort2. Length: 3000, time: 17 ms  
qsort3. Length: 3000, time: 16 ms  
gqsort2. Length: 3000, time: 41 ms  
gqsort3. Length: 3000, time: 39 ms
```



Futási idők mérése: eredmények (folyt.)

```
tmsort. Length: 3000, time: 15 ms
bmsort. Length: 3000, time: 10 ms
smsort. Length: 3000, time: 382 ms
smsort. Length: 3645, time: 19 ms
gtmsort. Length: 3000, time: 18 ms
gbmsort. Length: 3000, time: 26 ms
gmsort. Length: 3000, time: 463 ms
gmsort. Length: 3545, time: 24 ms
'lists:sort/1'. Length: 3000, time: 4 ms
'lists:sort/2'. Length: 3000, time: 9 ms
'lists:usort/1'. Length: 3000, time: 4 ms
'lists:usort/2'. Length: 3000, time: 11 ms
ok
```



Tartalom

1 Szekvenciális Erlang

- Deklaratív programozás, funkcionális programozás
- Erlang-bevezetés
- Erlang emulátor
- Típusok
- Szekvenciális programozás
- Típusspecifikáció
- Kivételkezelés
- Rekord
- Rekurzív adatstruktúrák
- Gyakori könyvtári függvények
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Listák használata: futamok
- Listák rendezése
- **Kifejezés kiértékelése**
- Absztrakció függvényekkel, eljárásokkal



Összetett kifejezés kiértékelése

Egy összetett kifejezést az Erlang két lépésben értékeli ki, ún. mohó kiértékeléssel. Az alábbi kiértékelési szabály rekurzív, azért ilyen egyszerű.

- 1 Először kiértékeli az operátort (műveleti jelet, függvényjelet) és az argumentumait (aktuális paramétereit),
- 2 majd alkalmazza az operátort az argumentumokra.

Az egyszerű kifejezés kiértékelési szabályai:

- 1 az állandó (jelölés) értéke az, amit jelöl,
- 2 a belső (beépített) művelet a megfelelő gépi utasításokat aktivizálja,
- 3 a név értéke az, amihez az adott *környezet* köti a nevet.

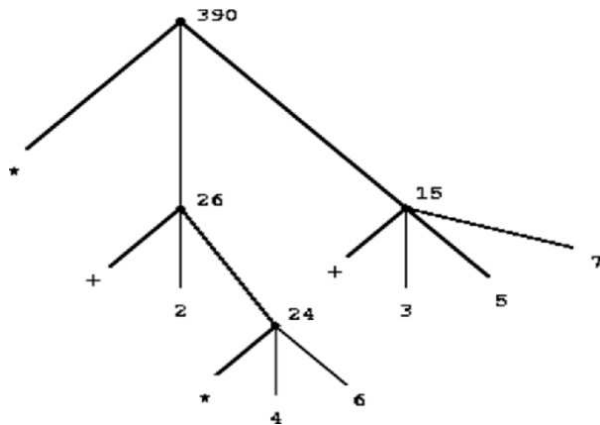
Megjegyzés: a 2. pont a 3. pont speciális esetének is tekinthető.



Összetett kifejezés kifejezésfája

A kifejezéseket ún. kifejezésfával ábrázolhatjuk. Példa:

$$(2+4*6) * (3+5+7) == (2+(4*6)) * ((3+5)+7) ==$$
$$(*) ((+) (2, (*)(4, 6)), (+)((+) (3, 5), 7))$$



A levelek
operátorok vagy
primitív kifejezések
(állandók, nevek), a
csomópontok
részkifejezések
eredményei.

A kiértékelés során az operandusok alulról fölfelé „terjednek”.



Függvényalkalmazás kiértékelése

- Egy felhasználói függvényeket tartalmazó kifejezést az összetett kifejezéshez hasonlóan értékeli ki az Erlang.
- Feltettük, hogy az Erlang tudja, hogyan alkalmazza a belső függvényeket az aktuális paramétereikre.
- Ezt most azzal egészítjük ki, hogy az Erlang egy függvény alkalmazásakor
 - ▶ a függvény törzsében a formális paraméterek összes előfordulását lecseréli a megfelelő aktuális paraméterre, majd kiértékeli a függvény törzsét.

Nézzük pl. a következő egyszerű függvények definícióját!

$$\text{sq}(X) \rightarrow X * X.$$
$$\text{sumsq}(X, Y) \rightarrow \text{sq}(X) + \text{sq}(Y).$$
$$f(A) \rightarrow \text{sumsq}(A + 1, A * 2).$$


Mohó és lusta kiértékelés

Mohó kiértékelés esetén minden lépésben egy rész kifejezést egy vele egyenértékű kifejezéssel helyettesítünk. Nézzük pl. az $f(5)$ mohó kiértékelését!

$$\begin{aligned} f(5) &\rightarrow \text{sumsq}(5+1, 5*2) \rightarrow \text{sumsq}(6, 5*2) \rightarrow \\ &\text{sumsq}(6, 10) \rightarrow \text{sq } 6 + \text{sq } 10 \rightarrow 6*6 + \text{sq } 10 \\ &\rightarrow 36 + \text{sq } 10 \rightarrow 36 + 10*10 \rightarrow 36 + 100 \rightarrow 136 \end{aligned}$$

- A függvényalkalmazás itt bemutatott *helyettesítési modellje*, az ún. egyenlők helyettesítése egyenlőkkel (*equals replaced by equals*) segíti a függvényalkalmazás *jelentésének* megértését.
- Olyan esetekben alkalmazható, amikor egy függvény *jelentése független* a környezetétől.
- Az értelmezők/fordítók rendszerint bonyolultabb modell szerint működnek.



Mohó és lusta kiértékelés (folyt.)

- Az Erlang tehát először kiértékeli az operátort és az argumentumait, majd alkalmazza az operátort az argumentumokra. Ezt a kiértékelési sorrendet *mohó* (eager) vagy *applikatív sorrendű* (applicative order) kiértékelésnek nevezzük.
- Van más lehetőség is: a kiértékelést addig halogatjuk, ameddig csak lehetséges. Ezt *lusta* (lazy), *szükség szerinti* (by need) vagy *normál sorrendű* (normal order) kiértékelésnek nevezzük.
- Nézzük az $f(5)$ lusta kiértékelését!

$$\begin{aligned} f(5) &\rightarrow \text{sumsq}(5+1, 5*2) \rightarrow \text{sq}(5+1) + \\ &\text{sq}(5*2) \rightarrow (5+1) * (5+1) + (5*2) * (5*2) \rightarrow \\ &6 * (5+1) + (5*2) * (5*2) \rightarrow 6 * 6 + \\ &(5*2) * (5*2) \rightarrow 36 + (5*2) * (5*2) \rightarrow 36 + \\ &10 * (5*2) \rightarrow 36 + 10 * 10 \rightarrow 36 + 100 \rightarrow 136 \end{aligned}$$



Mohó és lusta kiértékelés (folyt.)

- Igazolható, hogy olyan függvények esetén, amelyek jelentésének megértésére a helyettesítési modell alkalmas, a kétféle kiértékelési sorrend azonos eredményt ad.
- Vegyük észre, hogy lusta (szükség szerinti) kiértékelés mellett egyes részkifejezéseket néha többször is ki kell értékelni.
- A többszörös kiértékelést jobb értelmezők/fordítók (pl. Alice, Haskell) úgy kerülik el, hogy az azonos részkifejezéseket megjelölik, és amikor egy részkifejezést először kiértékelnek, *az eredményét megjegyzik*, a többi előfordulásakor pedig ezt az eredményt veszik elő. E módszer hátránya a nyilvántartás szükségessége. Ma általában ezt nevezik *lusta* kiértékelésnek.



Tartalom

1 Szekvenciális Erlang

- Deklaratív programozás, funkcionális programozás
- Erlang-bevezetés
- Erlang emulátor
- Típusok
- Szekvenciális programozás
- Típusspecifikáció
- Kivételkezelés
- Rekord
- Rekurzív adatstruktúrák
- Gyakori könyvtári függvények
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Listák használata: futamok
- Listák rendezése
- Kifejezés kiértékelése
- Absztrakció függvényekkel, eljárásokkal



Matematikai vs. funkcionális nyelvi függvények

- A funkcionális nyelvi függvények sokban hasonlítanak a matematikai függvényekhez: egy vagy több argumentumtól függő értéket adnak eredményül.
- Egy dologban azonban mindenképpen különböznek: a funkcionális nyelvi függvényeknek *hatékonyaknak* is kell lenniük.
- Nézzük pl. a négyzetgyök következő definícióját: $\sqrt{x} = y$, ahol $y \geq 0$ és $y^2 = x$.
- Ez az egyenletrendszer alkalmas pl. annak *ellenőrzésére*, hogy egy szám egy másiknak a négyzetgyöke-e, de nem alkalmas a négyzetgyök *előállítására*.
- A matematikai függvénnyel egy bizonyos tulajdonságot *deklarálunk*, a funkcionális nyelvi függvénnyel (eljárással) azt is megmondjuk, *hogyan kell kiszámítani* az adott értéket.
- A deklaratív programozás tehát csak az imperatív programozáshoz *képest* tekinthető deklaratívnek, vö. MIT és HOGYAN.



Négyzetgyökvonás Newton-módszerrel

- A négyzetgyökszámítás legismertebb módszere, a *szukcesszív approximáció* azon alapul, hogy

ha y az x négyzetgyökének egy közelítése, akkor az y és az x/y átlaga a négyzetgyök egy jobb közelítése.

A sorozat akkor ér véget, amikor a közelítő érték már elég jó.

- Írjuk le az algoritmust Erlang-nyelven.

```
sqrtIter(Guess, X) ->
    case goodEnough(Guess, X) of
        true ->
            Guess;
        false ->
            sqrtIter(improved(Guess, X), X)
    end.
```



Négyzetgyökvonás Newton-módszerrel (folyt.)

- A megoldási stratégiát, a *fölről lefelé haladó* (top down) módszert jól tükrözi a fenti programrészlet: kezdetben nem foglalkozunk a részletekkel, föltesszük, hogy minden megvan, ami kell, és később megírjuk.
- Definiáljuk a hiányzó részeket.

```
improved(Guess,X) -> average(Guess, X/Guess).  
average(X,Y) -> (X+Y)/2.  
goodEnough(Guess,X) ->  
    abs(sq(Guess) - X) < 0.0001.  
sq(X) -> X * X.
```

- Most már meghívhatjuk `sqrtIter`-t a négyzetgyök első közelítő értékével.

```
sqrt(X) -> sqrtIter(1.0, X).
```



Kifejező nevek, mellékhatás nélküli függvények

Azzal, hogy értelmes, kifejező nevet adunk az egyes programelemeknek, könnyebbé, egyszerűbbé tesszük

- „az ügyek szétválasztásával” (*separation of concerns*) a program kidolgozását,
- az olvasóknak a program megértését,
- a későbbiekben a program javítását.

Ha arra is ügyelünk, hogy a segédfüggvényeknek ne legyen *mellékhatásuk*, a specifikáció megtartása mellett később bármikor lecserélhetők lesznek.



Lineáris rekurzió és iteráció

- A faktoriális matematikai definíciójának hű tükörképe a következő Erlang-program:

$0! = 1$		<code>%% @spec factorial1(N::integer()) -></code>
$n! = n(n-1)!$		<code>%% F::integer()</code>
		<code>%% @pre N >= 0</code>
		<code>%% F == N!</code>
		<code>factorial1(0) -> 1;</code>
		<code>factorial1(N) when N>0 -></code>
		<code> N * factorial1(N-1).</code>

- Helyettesítési modellünket alkalmazva látható, hogy a program által létrehozott folyamat az összes tényezőt N-től 1-ig eltárolja, mielőtt az első szorzást végrehajtaná („késlelteti” a szorzásokat) – ez a program tehát *lineáris-rekurzív folyamatot* hoz létre.



Lineáris rekurzió és iteráció (folyt.)

- Ha először 1-et szoroznánk 2-vel, majd a részszorzatokat 3-mal, 4-gyel s.í.t., akkor az N érték meghaladásakor az utolsó részszorzat éppen N faktoriálisa lenne!
- Ehhez a programban szükségünk van egy olyan *formális paraméterre* (tkp. lokális változóra), amely tárolja a részszorzat aktuális értékét, és egy másikra, amely 1-től N -ig számlál. Az így létrehozott folyamat *lineáris-iteratív folyamat* lesz.

```
factorial2(N) when N >= 0 ->  
    factorial2(N,1,1).
```

```
factorial2(N,Product,Counter) ->  
    if (Counter > N) ->  
        Product;  
    true ->  
        factorial2(N, Product*Counter, Counter+1)  
end.
```



Lineáris rekurzió és iteráció (folyt.)

- További ór alkalmazásával:

```
factorial21(N) when N >= 0 ->  
    factorial21(N,1,1).
```

```
factorial21(N,Product,Counter) when Counter > N ->  
    Product;  
factorial21(N,Product,Counter) ->  
    factorial21(N, Product*Counter, Counter+1).
```

- factorial2/3 egyszerűbb szerkezetű, factorial3/2 változatát kapjuk, ha a számlálót *lefelé* számláltatjuk.

```
%% @pre : N >= 0  
factorial3(N) ->  
    factorial3(1,N).
```

```
factorial3(Product, 0) ->  
    Product;  
factorial3(Product, Counter) ->  
    factorial3(Product*Counter, Counter-1).
```

Eljárások, függvények és folyamatok

- Az eljárások, függvények olyan *minták*, amelyek megszabják a számítási folyamatok, processzek menetét, *lokális* viselkedését.
- Egy számítási folyamat *globális* viselkedését (pl. a szükséges lépések számát vagy a végrehajtási időt) általában nehéz megbecsülni, de törekednünk kell rá.
- Ne tévesszük össze egymással a rekurzív számítási folyamatot és a rekurzív függvényt, eljárást!
 - ▶ Egy rekurzív függvény esetén csupán a szintaxisról van szó, arról, hogy hivatkozik-e a függvény, eljárás *önmagára*.
 - ▶ A folyamat esetében viszont a folyamat menetéről, lefolyásáról beszélünk.
- Ha egy függvény *jobbrekurzív (tail-recursive)*, a megfelelő folyamat – az értelmező/fordító jószágától függően – lehet iteratív.

Hivatkozás: Structure and Interpretation of Computer Programs, 2nd ed., by H. Abelson, G. J. Sussman, J. Sussman, The MIT Press, 1996



Programhelyesség informális igazolása

- Egy rekurzív programról is be kell látnunk, hogy
 - ▶ funkcionálisan helyes (azaz azt kapjuk eredményül, amit várunk),
 - ▶ a kiértékelése biztosan befejeződik (nem „végtelen” a rekurzió).
- Bizonyítása lineáris rekurzió esetén egyszerű, hossz szerinti *strukturális indukcióval* lehetséges (azaz visszavezethető a teljes indukcióra).
- A `map` példáján mutatjuk be:

```
%% @spec map(F::func(), Xs::[atype()]) ->
%%                                     Ys::[btype()].
%% @type func() = atype() -> btype().
%% @type atype() = any().
%% @type btype() = any().
map(_F, []) -> [];
map(F, [X|Xs]) -> [F(X) | map(F, Xs)].
```



Programhelyesség informális igazolása (folyt.)

```
map (_F, []) -> [];  
map (F, [X | Xs]) -> [F (X) | map (F, Xs)] .
```

1 A függvény funkcionálisan helyes, ui.

- ▶ belátjuk, hogy az F jól transzformálja a lista első elemét (a fejét);
- ▶ feltesszük (feltehetjük), hogy a függvény jól transzformálja az eggyel rövidebb listát (a lista farkát);
- ▶ belátható, hogy a fej transzformálásával kapott elem és a fark transzformálásával kapott lista összefűzése a várt listát adja.

2 A kiértékelés véges számú lépésben befejeződik, mert

- ▶ a lista (mohó kiértékelés mellett!) véges,
- ▶ a függvényt a *rekurzív ágba*n minden lépésben egyre rövidülő listára alkalmazzuk, és
- ▶ a rekurziót előbb-utóbb leállítjuk (ui. kezeljük az *alapesetet*, van rekurziót nem tartalmazó klóz).



Elágazó rekurzió – Fibonacci-számok

- Az előbb lineáris-rekurzív, ill. lineáris-iteratív folyamatokra láttunk példát (ld. faktoriális kiszámítása kétféleképpen).
- Most az *elágazó rekurzióra* nézünk példát: Fibonacci-számok sorozatát állítjuk elő kétfelé ágazó rekurzióval.
- Bármely Fibonacci-szám az öt megelőző két Fibonacci-szám összege: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...
- A matematikai definíció könnyen átírható Erlang-függvénnyé.

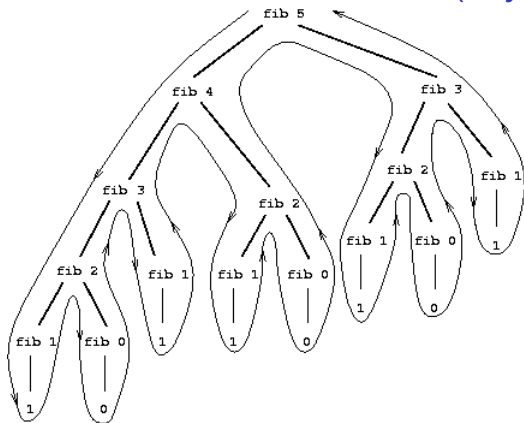
$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n-1) + \\ &\quad F(n-2), \\ &\quad \text{ha } n > 1 \end{aligned}$$

```
%% @spec fib1(N::integer()) ->
%%                               F::integer().
%% @pre 0 <= N.
%% F az N-edik Fibonacci-szám
fib1(0) -> 0;
fib1(1) -> 1;
fib1(N) when N>1 ->
    fib1(N-1)+fib1(N-2).
```

Emlékeztető: a fenti definícióban a `fib(N)` klóznak kell az utolsónak lennie, mert az `N` minta minden argumentumra illeszkedik.



Elágazó rekurzió – Fibonacci-számok (folyt.)



- Az ábra illusztrálja az elágazóan rekurzív folyamatot `fib 5` kiszámításakor.
- `fib 5`-öt `fib 4` és `fib 3`, `fib 4`-et `fib 3` és `fib 2` kiszámításával stb. kapjuk.



Elágazó rekurzió – Fibonacci-számok (folyt.)

- Az előző program alkalmas az elágazó rekurzió lényegének bemutatására, de alkalmatlan a Fibonacci-számok előállítására.
- Vegyük észre, hogy pl. `fib 3`-at kétszer is kiszámítjuk, azaz a munkának ezt a részét (kb. a harmadát) feleslegesen végezzük el.
- Belátható, hogy az $F(n)$ meghatározásához pontosan $F(n+1)$ levélből álló fát kell bejárni, azaz ennyiszer kell meghatározni $F(0)$ -at vagy $F(1)$ -et.
- $F(n)$ exponenciálisan nő n -nel.

Pontosabban, $F(n)$ a $\Phi^n/\sqrt{5}$ -höz közel eső egész, ahol $\Phi = (1 + \sqrt{5})/2 \approx 1.61803$, az ún. *arany metszés* arányszáma.

Φ kielégíti a $\Phi^2 - \Phi - 1 = 0$ egyenletet.



Elágazó rekurzió – Fibonacci-számok (folyt.)

- A megteendő lépések száma tehát $F(n)$ -nel együtt exponenciálisan nő n -nel.
- A tárigény ugyanakkor csak lineárisan nő n -nel, mert csak azt kell nyilvántartani, hogy hányadik szinten járunk a fában.
- Általában is igaz, hogy elágazó rekurzió esetén *a lépések száma* a fa csomópontjainak a számával, *a tárigény* viszont a fa maximális mélységével arányos.
- A Fibonacci-számok azonban lineáris-iteratív folyamattal is előállíthatók.
Ha az a és b változók kezdőértéke rendre $F(1) = 1$ és $F(0) = 0$, és ismétlődően alkalmazzuk az $a \leftarrow a + b$, $b \leftarrow a$ transzformációkat, akkor n lépés után $a = F(n+1)$ és $b = F(n)$ lesz.
- Az iteratív folyamatot létrehozó Erlang-függvény egy változatát a következő dián mutatjuk be.



Elágazó rekurzió – Fibonacci-számok (folyt.)

- `fib2(N) when N>=0 -> fib2(N,0,0,1).`

```
fib2(N,N,A,_B) -> A;
```

```
fib2(N,I,A,B) -> fib2(N,I+1,A+B,A).
```

Figyelem: a klózik sorrendje, mivel nem egymást kizáróak a minták, lényeges!

- Egy argumentummal kevesebb is elég, ha I -t nem növeljük, hanem N -től 0-ig csökkentjük.

```
fib3(N) when N>=0 -> fib3(N,0,1).
```

```
fib3(0,A,_B) -> A;
```

```
fib3(I,A,B) -> fib3(I-1,A+B,A).
```

- A Fibonacci-példában a lépések száma elágazó rekurziónál tehát n -nel exponenciálisan, lineáris rekurziónál n -nel arányosan nőtt, kis n -ekre is hatalmas a nyereség!



Elágazó rekurzió (folyt.)

- Téves lenne azonban azt a következtetést levonni az előző példából, hogy az elágazó rekurzió használhatatlan. Amikor hierarchikusan strukturált adatokon kell műveleteket végezni, pl. egy fát kell bejárni, akkor az elágazó rekurzió (angolul: *tree recursion*) nagyon is természetes és hasznos eszköz.
- Az elágazó rekurzió numerikus számításoknál az algoritmus első megfogalmazásakor is hasznos lehet: gondoljunk csak arra, hogy milyen könnyű volt átírni a Fibonacci-számok matematikai definícióját programmá.
- Ha már értjük a feladatot, az első, rossz hatékonyságú változatot könnyebb átírni jó, hatékony programmá. Az elágazó rekurzió segíthet a feladat megértésében.

Az iteratív Fibonacci-algoritmushoz csak egy aprócska ötlet kellett. A következő feladatra azonban nem könnyű iteratív algoritmust írni.



Elágazó rekurzió – pénzváltás

- Hányféleképpen lehet felváltani egy dollárt 50, 25, 10, 5 és 1 centesekre?
- Általánosabban: adott összeget adott érmékkel hányféleképpen lehet felváltani?

Tegyük föl, hogy n darab érme áll a rendelkezésünkre valamilyen (pl. nagyság szerint csökkenő) sorrendben. Ekkor az a összeg lehetséges felváltásainak számát úgy kapjuk meg, hogy

- kiszámoljuk, hogy az a összeg hányféleképpen váltható fel az első (d értékű) érmét kivéve a többi érmével, és ehhez
- hozzáadjuk, hogy az $a - d$ összeg hányféleképpen váltható fel az összes érmével, az elsőt is beleértve – más szóval azt, hogy az a összeget hányféleképpen tudjuk úgy felváltani, hogy a d érmét legalább egyszer felhasználjuk.

A feladat tehát rekurzióval megoldható, hiszen redukálható úgy, hogy kisebb összegeket kevesebb érmével kell felváltanunk.



Elágazó rekurzió – pénzváltás (folyt.)

A következő alapeseteket különböztessük meg:

- Ha $a = 0$, a felváltások száma 1.
(Ha az összeg 0, csak egyféleképpen, 0 db érmével lehet „felváltani”.)
- Ha $a < 0$, a felváltások száma 0.
- Ha $n = 0$, a felváltások száma 0.

A példában a `firstDenomination` (magyarul *első címlet*) függvényt felsorolással valósítottuk meg.

Tömörebb és rugalmasabb lenne a megvalósítása lista alkalmazásával – ezt meghagyjuk otthoni gyakorló feladatnak.



Elágazó rekurzió – pénzváltás (folyt.)

```
valtas(Osszeg) -> váltas(Osszeg, 5) .
```

```
váltas(Osszeg, Erme fajta) ->  
    if Osszeg < 0 orelse Erme fajta == 0 -> 0;  
        Osszeg == 0 -> 1;  
        true -> váltas(Osszeg, Erme fajta-1) +  
                váltas(Osszeg-cimlet (Erme fajta) , Erme fajta)  
    end.
```

```
cimlet(1) -> 1;  
cimlet(2) -> 5;  
cimlet(3) -> 10;  
cimlet(4) -> 25;  
cimlet(5) -> 50.
```

Gyakorló feladatok

- Írja át a `cimlet` függvényt úgy, hogy a címleteket lista tárolja.
- Írja át a `cC` függvényt több klózból álló függvénnnyé!



Hatványozás

- A ma eddig látott folyamatokban a kiértékelési (végrehajtási) lépések száma az adatok n számával lineárisan, ill. exponenciálisan nőtt. Most olyan példa következik, amelyben a lépések száma az n logaritmusával arányos.
- A b szám n -edik hatványának definícióját ugyancsak könnyű átírni Erlangba.

$b^0 = 1$		<code>%% @spec expt1(B::integer(),</code>
$b^n = b \cdot b^{n-1}$		<code>%% N::integer()) -></code>
		<code>%% R::integer() .</code>
		<code>%% @pre N >= 0.</code>
		<code>%% B N-edik hatványa R.</code>
		<code>expt1(B,0) -> 1;</code>
		<code>expt1(B,N) when N>0 -></code>
		<code> B * expt1(B,N-1) .</code>

- A létrejövő folyamat lineáris-rekurzív. $O(n)$ lépés és $O(n)$ méretű tár kell a végrehajtásához.



Hatványozás – lineáris-iteratív

- A faktoriálisszámításhoz hasonlóan könnyű felírni $\text{expt1}/2$ lineáris-iteratív változatát.

$\text{expt2}(B, N) \text{ when } N \geq 0 \rightarrow \text{expt2}(B, N, 1) .$

$\text{expt2}(B, 0, P) \rightarrow P ;$

$\text{expt2}(B, N, P) \rightarrow$
 $\text{expt2}(B, N-1, B * P) .$

- $O(n)$ lépés és $O(1)$ – azaz konstans – méretű tár kell a végrehajtásához.



Hatványozás - logaritmikus

- Kevesebb lépés is elég, ha kihasználjuk az alábbi egyenlőségeket:

$$b^0 = 1$$

$$b^n = (b^{n/2})^2, \text{ ha } n \text{ páros}$$

$$b^n = b \cdot b^{n-1}, \text{ ha } n \text{ páratlan}$$

```
expt3(_B,0) -> 1;
expt3(B,N) when N>0 ->
    case even(N) of
        true ->
            sq(expt3(B,N div 2));
        false ->
            B * expt3(B,N-1)
    end.

even(N) -> N rem 2 == 0.
```

- A lépések száma és a tár mérete $O(\lg n)$ -nel arányos.



Hatványozás – logaritmikus, konstans tárigényű

- Konstans tárigényű, iteratív változata:

```
expt4(B,N) when N>=0 -> expt4(B,N,1).
```

```
expt4(_B,0,R) -> R;  
expt4(B,N,R) ->  
    case even(N) of  
        true ->  
            expt4(B,N div 2,R*R);  
        false ->  
            expt4(B,N-1,R*B)  
    end.
```

```
> f(F), F = fun(B,N) -> misc:expt4(B,N) end,  
    {F(2,0), F(2,1), F(2,2), F(2,3), F(2,4),  
     F(2,5), F(2,6), F(2,7), F(2,8)}.  
??
```



Hatványozás – logaritmikus, konstans tárigényű (folyt.)

- Tesztelési eredmények

```
> f(F), F = fun(B,N) -> misc:expt4(B,N) end,  
  {F(2,0), F(2,1), F(2,2), F(2,3), F(2,4),  
   F(2,5), F(2,6), F(2,7), F(2,8)}.  
{1, 2, 2, 8, 2, 32, 8, 128, 2}
```

- Sajnos, ez a változat ROSSZ 0-nál nagyobb páros kitevőkre.

```
expt4(B,N) when N>=0 ->  
  expt4(B,N,1).
```

```
expt4(_B,0,R) -> R;  
expt4(B,N,R) ->  
  case N rem 2 == 0 of  
    true ->  
      expt4(B,N div 2,R*B);  
    false ->  
      expt4(B,N-1,R*B)  
  end.
```

```
expt3(_B,0) -> 1;  
expt3(B,N) when N>0 ->  
  case N rem 2 == 0 of  
    true ->  
      sq(expt3(B,N div 2));  
    false ->  
      B * expt3(B,N-1)  
  end.
```

- Házi feladat: javítsuk!

