

# VI. rész

## Haladó Erlang

- 1 Bevezetés
- 2 Cékla: deklaratív programozás C++-ban
- 3 Prolog alapok
- 4 Erlang alapok
- 5 Haladó Prolog
- 6 Haladó Erlang**

# Tartalom

6

## Haladó Erlang

- Rekurzív adatstruktúrák
- Pontos megoldás funkcionális megközelítésben
- Listák használata: futamok
- Rekurzió fajtái
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Mohó és lusta kiértékelés

## Lineáris rekurzív adatstruktúrák – Verem (Stack)

- Lista: rekurzív adatstruktúra: `@type list() = [] | [any()|list()]`
- Verem: ennessel valósítjuk meg, listával triviális lenne
- Műveletek: üres verem létrehozása, verem üres voltának vizsgálata, egy elem berakása, utoljára berakott elem leválasztása

```
stack.erl
```

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

```
empty() -> empty.
```

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

```
push(X, empty) -> {X,empty};  
push(X, {_X,_S}=S) -> {X,S}. % {_X,_S}=S: réteges minta
```

```
pop(empty) -> error;  
pop({X,S}) -> {X,S}.
```

## Verem példák

```
2> S1 = stack:push(1, stack:empty()).  
{1,empty}  
3> S2 = stack:push(2, S1).  
{2,{1,empty}}  
4> S3 = stack:push(3, S2).  
{3,{2,{1,empty}}}
```

- Pl. megfordíthatunk egy listát; 1. lépés: verembe tesszük az elemeket

```
5> Stack = lists:foldl(fun stack:push/2, stack:empty(), "szoveg").  
{103,{101,{118,{111,{122,{115,empty}}}}}}
```

- 2. lépés: a verem elemeit sorban kivesszük és listába fűzzük

`stack.erl` – folytatás

*% to\_list(S) az S verem elemeit tartalmazó lista LIFO sorrendben.*

```
to_list(empty) -> [];
```

```
to_list({X,S}) -> [X|to_list(S)].
```

```
6> stack:to_list(Stack).  
"gevozs"
```

## Elágazó rekurzív adatstruktúrák (pl. bináris fa)

- Műveletek bináris fákon: létrehozása, mélysége, leveleinek száma

tree.erl – Műveletek bináris fákon: létrehozása, mélysége, leveleinek száma

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

```
empty() -> leaf.           % Üres fa.
```

```
node(V, Lt, Rt) -> {V,Lt,Rt}. % Lt és Rt fák összekapcsolása  
% egy új V értékű csomóponttal.
```

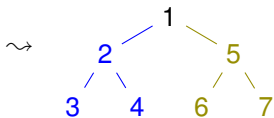
```
max(X, Y) when X>Y -> X;  
max(_X, Y)          -> Y.
```

```
depth(leaf)          -> 0;           % Fa legnagyobb mélysége.  
depth({_ ,Lt,Rt}) -> 1 + max(depth(Lt), depth(Rt)).
```

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

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

```
L=empty(), T=node(1, node(2, node(3,L,L),
                             node(4,L,L)),
                  node(5, node(6,L,L),
                             node(7,L,L)))
```



$T \mapsto \{1, \{2, \{3, \text{leaf}, \text{leaf}\}, \{4, \text{leaf}, \text{leaf}\}\}, \{5, \{6, \text{leaf}, \text{leaf}\}, \{7, \text{leaf}, \text{leaf}\}\}$

## tree.erl – folytatás

```
to_list_prefix(leaf) -> [];
```

```
to_list_prefix({V,Lt,Rt}) ->
```

```
[V] ++ to_list_prefix(Lt) ++ to_list_prefix(Rt).
```

```
to_list_infix(leaf) -> [];
```

```
to_list_infix({V,Lt,Rt}) ->
```

```
to_list_infix(Lt) ++ [V] ++ to_list_infix(Rt).
```

```
from_list([]) -> empty();
```

```
from_list(L) -> {L1,[X|L2]} = lists:split(length(L) div 2, L),
                node(X, from_list(L1), from_list(L2)).
```

## Elágazó rekurzív adatstruktúrák – könyvtárszerkezet

```

2> Home = {d,"home",                               %   home
           [{d,"kitti",                             %   home/kitti
             [{d,".firefox",[]},                    %   home/kitti/.firefox
              {f,"dir.erl"},                          %   home/kitti/dir.erl
              {f,"khf1.erl"},                         %   home/kitti/khf1.pl
              {f,"khf1.pl"}]}],                     %   home/kitti/khf1.erl
           {d,"ludvig",[]}}.                         %   home/ludvig

```

## dir.erl – Könyvtárszerkezet kezelése

```

% @type tree()      = file() | directory().
% @type file()     = {f, name()}.
% @type directory() = {d, name(), [tree()]}.
% @type name()     = string().

```

*% Fa mérete (könyvtárak és fájlok számának összege).*

```

count({f,_})      -> 1;
count({d,_ ,L})   -> 1 + lists:sum([count(I) || I <- L]).

```

## Könyvtárszerkezet – folytatás

dir.erl – Könyvtárszerkezet kezelése (folytatás)

```
% @spec subtree(Path::[name()], Tree::tree()) -> tree() | notfound.
% Tree fa Path útvonalon található részfája.
subtree([Name], {f,Name} = Tree)           -> Tree;
subtree([Name], {d,Name,_} = Tree)        -> Tree;
subtree([Name|[Sub|_]=SubPath], {d,Name,L}) ->
    case lists:keyfind(Sub, 2, L) of
        false -> notfound;
        Tree  -> subtree(SubPath, Tree)
    end;
subtree(_, _)                             -> notfound.
```

```
3> dir:subtree(string:tokens("home/kitti/.firefox", "/"), Home).
{d,".firefox",[]}
4> dir:subtree(string:tokens("home/kitti/firefox", "/"), Home).
notfound
```



# Tartalom

## 6 Haladó Erlang

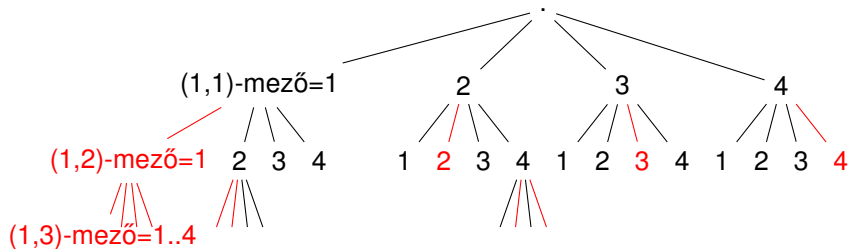
- Rekurzív adatstruktúrák
- **Pontos megoldás funkcionális megközelítésben**
- Listák használata: futamok
- Rekurzió fajtái
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Mohó és lusta kiértékelés

## Pontos megoldás (Exact solution)

- Kombinatorikában sokszor *optimális megoldás* (optimal solution)
- Egy probléma pontos (egzakt) megoldása
- Nem közelítő (approximáció), nem szuboptimális (bizonyos heurisztikák)
- Keresési feladat: valamilyen *értelmezési tartomány* azon elemeit keressük, melyek megfelelnek a kiírt *feltételeknek*
  - lehetséges megoldás = *jelölt*
  - értelmezési tartomány = *keresési tér (search space)*, jelöltek halmaza
  - feltételek = *korlátok vagy kényszerek (constraints)*
- Pl. egy 16 mezős Sudoku-feladvány helyes megoldásai, 8 királynő egy saktáblán, Hamilton-kör egy gráfban, Imre herceg nagyszülei . . .
- A Prolog végrehajtási algoritmus képes egy predikátumokkal és egy célsorozattal leírt probléma összes megoldását felsorolni (!)
- Funkcionális megközelítésben a megoldások felsorolását a programozónak meg kell írnia (logikaiban is megírható természetesen)

## Keresési tér bejárása

- Itt csak véges keresési térrel foglalkozunk
- A megoldás keresését esetekre bonthatjuk, azokat alesetekre stb.  $\rightsquigarrow$  Ilyenkor egy *keresési fát* járunk be
- Pl. 16 mezős Sudoku (1. sor, 1. oszlop) mezeje lehet 1,2,3,4  
Ezen belül (1. sor, 2. oszlop) mezeje lehet 1,2,3,4 stb.



- Bizonyos eseteknél (piros) tudjuk, hogy nem lesz megoldás (ha egy sorban egy érték több mezőben is szerepel)
- Hatékony megoldás: a keresési fa részeit levágjuk (nem járjuk be)

## Példa: Send + More = Money

- Feladat: Keressük meg azon (S, E, N, D, M, O, R, Y) számnyolcasokat, melyekre  $0 \leq S, E, N, D, M, O, R, Y \leq 9$  és  $S, M > 0$ , ahol az eltérő betűk eltérő értéket jelölnek, és

```

  S E N D
+ M O R E
-----
M O N E Y

```

a papíron történő összeadás szabályai szerint, vagyis

$$\begin{aligned}
 (1000S + 100E + 10N + D) + (1000M + 1000 + 10R + E) &= \\
 &= 10000M + 10000 + 100N + 10E + Y.
 \end{aligned}$$

- Naív megoldásunk: járjuk be a teljes keresési teret, és szűrjük meg azon nyolcasokra, melyekre teljesülnek a feltételek
- Keresési tér  $\subseteq \{0, 1, \dots, 9\}^8$ , azaz egy 8-elemű Descartes-szorzat, mérete  $10^8$  (10 számjegy 8-adosztályú ismétléses variációi)
- Megoldás:

$$\{(S, E, N, D, M, O, R, Y) \mid S, E, N, D, M, O, R, Y \in \{0..9\}, \text{all\_different}, S, M > 0, \text{SEND} + \text{MORE} = \text{MONEY}\}$$

## Kimerítő keresés

Exhaustive search, Generate and test, Brute force

- Kimerítő keresés: teljes keresési tér bejárása, jelöltek szűrése

sendmory.erl – Send More Money megoldások, alapfogalmak

```
% @type d() = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.
```

```
% @type octet() = {d(),d(),d(),d(),d(),d(),d(),d()}.
```

```
% @spec num(Ns::[d()]) -> N::integer().
```

```
% Az Ns számjegylista decimális számként N.
```

```
num(Ns)-> lists:foldl(fun(X,E) -> E*10+X end, 0, Ns).
```

```
% @spec check_sum(octet()) -> bool().
```

```
% A jelölt teljesíti-e az összeadási feltételt.
```

```
check_sum({S,E,N,D,M,O,R,Y}) ->
```

```
    Send = num([S,E,N,D]),
```

```
    More = num([M,O,R,E]),
```

```
    Money = num([M,O,N,E,Y]),
```

```
    Send+More ::= Money.
```

## Kimerítő keresés – folytatás

sendmory.erl – folytatás

```
% @spec all_different(Xs::[any()]) -> B::bool()
```

```
all_different(L) -> length(L) == length(lists:usort(L)).
```

```
% @spec smm0() -> [octet()].
```

```
smm0() -> Ds = lists:seq(0, 9),
```

```
    [{S,E,N,D,M,O,R,Y} ||
```

```
        S <- Ds,
```

```
        E <- Ds,
```

```
        N <- Ds,
```

```
        D <- Ds,
```

```
        M <- Ds,
```

```
        O <- Ds,
```

```
        R <- Ds,
```

```
        Y <- Ds,
```

```
        all_different([S,E,N,D,M,O,R,Y]),
```

```
        S > 0, M > 0,
```

```
        check_sum({S,E,N,D,M,O,R,Y})].
```

## Keresési fa csökkentése (1)

- $10^8$  eset ellenőrzése túl sokáig tart
- Ötlet: korábban, már generálás közben is szűrhetjük az egyezéseket

sendmory.erl – folytatás

```
% @spec smm1() -> [octet()].
```

```
smm1() ->
```

```
  Ds = lists:seq(0, 9),
```

```
  [{S,E,N,D,M,O,R,Y} ||
```

```
    S <- Ds,
```

```
    E <- Ds, E /= S,
```

```
    N <- Ds, not lists:member(N, [S,E]),
```

```
    D <- Ds, not lists:member(D, [S,E,N]),
```

```
    M <- Ds, not lists:member(M, [S,E,N,D]),
```

```
    O <- Ds, not lists:member(O, [S,E,N,D,M]),
```

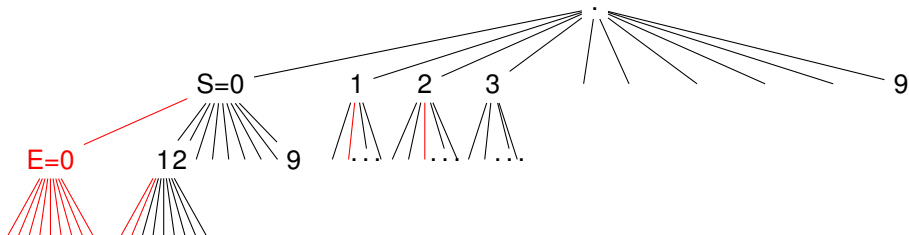
```
    R <- Ds, not lists:member(R, [S,E,N,D,M,O]),
```

```
    Y <- Ds, not lists:member(Y, [S,E,N,D,M,O,R]),
```

```
    S > 0, M > 0,
```

```
    check_sum({S,E,N,D,M,O,R,Y})].
```

## Keresési fa csökkentése (2)



- A keresési fában **azon részfákat, amelyekben egyezés van (pirosak)**, már generálás közben elhagyhatjuk
- Ez már nem kimerítő keresés (nem járjuk be az összes jelöltet)
- A javulást annak köszönhetjük, hogy jelöltek tesztelését előrébb hoztuk
- Vegyük észre, hogy a keresési tér csökkentésével is ide juthatunk: új keresési tér  $\subseteq$  {10 elem 8-adosztályú **ismétlés nélküli** variációi}
- Mérete  $10! / (10 - 8)! = 1\,814\,400 \ll 100\,000\,000$



## Variációk felsorolása listanézettel

```
1> Domain = [a,b,c,d].           % A halmaz.
[a,b,c,d]
2> IVar = [ {X,Y,Z} ||          % Ismétléses variációk.
            X <- Domain,
            Y <- Domain,
            Z <- Domain ].

[{a,a,a}, {a,a,b}, {a,a,c}, {a,a,d}, {a,b,a}, {a,b,b}, {...}|...]
3> length(IVar).
64                               % 4*4*4 = 64.
4> INVar = [ {X,Y,Z} ||        % Ismétlés nélküli variációk.
            X <- Domain,
            Y <- Domain -- [X],
            Z <- Domain -- [X,Y] ].

[{a,b,c}, {a,b,d}, {a,c,b}, {a,c,d}, {a,d,b}, {a,d,c},
 {b,a,c},
 {...}|...]
5> length(INVar).
24                               % 4!/1! = 24.
```

## Keresési tér csökkentése

- Újból kimerítő keresés, de kisebb a keresési tér

sendmory.erl – folytatás

```
% @spec smm2() -> [octet()].
```

```
% Minden ellenőrzés a generálás után történik.
```

```
smm2() ->
```

```
  Ds = lists:seq(0, 9),  
  [{S,E,N,D,M,O,R,Y} ||  
   S <- Ds -- [],  
   E <- Ds -- [S],  
   N <- Ds -- [S,E],  
   D <- Ds -- [S,E,N],  
   M <- Ds -- [S,E,N,D],  
   O <- Ds -- [S,E,N,D,M],  
   R <- Ds -- [S,E,N,D,M,O],  
   Y <- Ds -- [S,E,N,D,M,O,R],  
   S > 0, M > 0,  
   check_sum({S,E,N,D,M,O,R,Y})].
```

## Kimerítő keresés újból: keresési tér explicit felsorolása

- Vajon érdemes-e a jelöltek generálását elválasztani a teszteléstől? **Nem!**

sendmory.erl – folytatás

```
% @spec perms() -> [octet()].
```

```
% Számjegyek ismétlés nélküli 8-adosztályú variációi
```

```
perms8() -> Ds = lists:seq(0,9),  
           [{S,E,N,D,M,O,R,Y} ||  
            S <- Ds -- [],  
            E <- Ds -- [S],  
            N <- Ds -- [S,E],  
            D <- Ds -- [S,E,N],  
            M <- Ds -- [S,E,N,D],  
            O <- Ds -- [S,E,N,D,M],  
            R <- Ds -- [S,E,N,D,M,O],  
            Y <- Ds -- [S,E,N,D,M,O,R] ].
```

```
% @spec smm3() -> [octet()].
```

```
smm3() -> [Sol || {S,_E,_N,_D,M,_O,_R,_Y} = Sol <- perms8(),  
                S > 0, M > 0, check_sum(Sol)].
```

## Kimerítő keresés újból: keresési tér explicit felsorolása (2)

- Tovább csökkenthető a keresési tér, ha előrébb mozgatunk feltételeket

sendmory.erl – folytatás

```
% @spec smm4() -> [octet()].
```

```
% További ellenőrzések generálás közben.
```

```
smm4() ->
```

```
  Ds = lists:seq(0,9),
```

```
  [{S,E,N,D,M,O,R,Y} ||
```

```
    S <- Ds -- [0],                % 0 kizárva
```

```
    E <- Ds -- [S],
```

```
    N <- Ds -- [S,E],
```

```
    D <- Ds -- [S,E,N],
```

```
    M <- Ds -- [0,S,E,N,D],        % 0 kizárva
```

```
    O <- Ds -- [S,E,N,D,M],
```

```
    R <- Ds -- [S,E,N,D,M,O],
```

```
    Y <- Ds -- [S,E,N,D,M,O,R],
```

```
    check_sum({S,E,N,D,M,O,R,Y})].
```

## Vágások a keresési fában generálás közben

- Ötlet: építsük hátulról a számokat, és ellenőrizzük a részösszegeket még generálás közben

`sendmory.erl` – folytatás

```

smm5() ->                                %%      S E N D
Ds = lists:seq(0, 9),                      %%      + M O R E
[ {S,E,N,D,M,O,R,Y} | |                   %%      = M O N E Y
  D <- Ds -- [],
  E <- Ds -- [D],
  Y <- Ds -- [D,E],                        % <- lehetne javítani még...
  (D+E) rem 10 == Y,
  N <- Ds -- [D,E,Y],
  R <- Ds -- [D,E,Y,N],
  (num([N,D])+num([R,E])) rem 100 == num([E,Y]),
  O <- Ds -- [D,E,Y,N,R],
  (num([E,N,D])+num([O,R,E])) rem 1000 == num([N,E,Y]),
  S <- Ds -- [D,E,Y,N,R,O,O],
  M <- Ds -- [D,E,Y,N,R,O,S,O],
  check_sum({S,E,N,D,M,O,R,Y})] .

```

## Vágások a keresési fában generálás közben (2)

- A vágások eredményeképpen nagyságrendileg gyorsabb megoldást kapunk
- A generálás minél korábbi fázisában vágunk, annál jobb: a keresési fában nem a legalsó szintről kell *visszalépni*, hogy új megoldást keressünk
- Előzőből ötlet: építsünk rész megoldásokat, és minden építő lépésnél ellenőrizzük, hogy lehet-e értelme a rész megoldást bővíteni megoldássá

`sendmory.erl` – folytatás

```
% @type partial_solution() = {[d()], [d()], [d()]}
```

```
% @spec smm6() -> [octet()].
```

```
smm6() ->
```

```
    smm6({[], [], []}, 5, lists:seq(0,9)).
```

- `{[], [], []}` a kiindulási rész megoldásunk
- 5 méretű megoldásokat kell építeni
- `lists:seq(0,9)` a változók tartománya

## Vágások a keresési fában generálás közben (3)

- Egy `PartialSolution` = `{Sendlista, Morelista, Moneylista}` részmegoldás csak akkor bővíthető megoldássá, ha
  - A listák számjegyei jó pozícióban helyezkednek el: azonos betűk egyeznek, többi számjegy különbözik
  - A részletösszeg is helyes, csak az átvitelben térhet el

`sendmory.erl` – folytatás

```
% @spec check_equals(partial_solution()) -> bool().
```

```
check_equals(PartialSolution) ->
```

```
case PartialSolution of
```

```
{[D], [E], [Y]} -> all_different([D,E,Y]);
```

```
{[N,D], [R,E], [E,Y]} -> all_different([N,D,R,E,Y]);
```

```
{[E,N,D], [O,R,E], [N,E,Y]} -> all_different([O,N,D,R,E,Y]);
```

```
{[S,E,N,D], [M,O,R,E], [O,N,E,Y]} -> all_different([S,M,O,N,D,R,E,Y]);
```

```
{[O,S,E,N,D], [O,M,O,R,E], [M,O,N,E,Y]} ->
```

```
    all_different([S,M,O,N,D,R,E,Y]) andalso all_different([O,S,M]);
```

```
- -> false
```

```
end.
```

## Vágások a keresési fában generálás közben (4)

- Egy `PartialSolution = {Sendlista, Morelista, Moneylista}` részmegoldás csak akkor bővíthető megoldássá, ha
  - A listák számjegyei jó pozícióban helyezkednek el: azonos betűk egyeznek, többi számjegy különbözik
  - A részletösszeg is helyes, csak az átvitelben térhet el

`sendmory.erl` – folytatás

```
% @spec check_sum(partial_solution()) -> bool().  
% Ellenőrzi, hogy aritmetikai szempontból helyes-e a részmegoldás.  
% Az átvittel (carry) nem foglalkozik, mert mindkettő helyes:  
% {[1,2],[3,4],[4,6]} és {[9],[2],[1]},  
% mert építhető belőlük teljes megoldás.  
check_partialsom({Send, More, Money}) ->  
  S = num(Send), M = num(More), My = num(Money),  
  (S+M) rem round(math:pow(10,length(Send))) := My.
```



## Vágások a keresési fában generálás közben (5)

sendmory.erl – folytatás

```
% @spec smm6(PS::partial_solution(), Num::integer(),
%           Domain::[integer()]) -> Sols::[octet()].
% Sols az összes megoldás, mely a PS részmegoldásból építhető,
% mérete (Send hossza) =< Num, a számjegyek tartománya Domain.
smm6({Send,_,_} = PS, Num, _Domain) when length(Send) == Num ->
  {[O,S,E,N,D], [O,M,O,R,E], [M,O,N,E,Y]} = PS,
  [{S,E,N,D,M,O,R,Y}];
smm6({Send,More,Money}, Num, Domain) ->
  [Solution ||
    Dsend <- Domain,
    Dmore <- Domain,
    Dmoney <- Domain,
    PSol1 <- [ {[Dsend|Send], [Dmore|More], [Dmoney|Money]} ],
    % pl. így tudunk lekötni változót: PSol1 <- [ Érték ],
    check_equals(PSol1),
    check_partialsun(PSol1),
    Solution <- smm6(PSol1, Num, Domain) ].
```

## Korlát-Kielégítési Probléma (Constraint Satisfaction Problem)

- Eddig előre „könnyen” átlátható keresési fát terveztünk meg, vágunk meg és jártunk be; de a végső cél nem az átlátható keresési fa
- CSP-megközelítés:
  - amíg lehet, szűkítsük a választási lehetőségeket a *korlátok* alapján
  - ha már nem lehet, bontsuk esetekre a választási lehetőségeket

### SMM mint CSP probléma = (Változók, Tartományok, Korlátok)

- Változók: S, E, N, D, M, O, R, Y, segédváltozók: 0, C<sub>1</sub>, C<sub>2</sub>, C<sub>3</sub>, C<sub>4</sub>

Tartományok:	0	c <sub>1</sub>	c <sub>2</sub>	c <sub>3</sub>	c <sub>4</sub>	s	e	n	d	m	o	r	y
Alsó határ:	0	0	0	0	0	1	0	0	0	1	0	0	0
Felső határ:	0	1	1	1	1	9	9	9	9	9	9	9	9

- Korlátok:

$$\begin{array}{r}
 \text{S E N D} \\
 + \text{M O R E} \\
 \hline
 \text{M O N E Y}
 \end{array}
 \qquad
 \begin{array}{l}
 d + e + 0 = y + 10 \cdot c_1 \\
 n + r + c_1 = e + 10 \cdot c_2 \\
 e + o + c_2 = n + 10 \cdot c_3 \\
 s + m + c_3 = o + 10 \cdot c_4 \\
 0 + 0 + c_4 = m + 10 \cdot 0
 \end{array}$$

## CSP tevékenységek – szűkítés

- **Szűkítés egy korlát szerint:** egy korlát egy változójának  $d_i$  értéke *felesleges*, ha nincs a korlát többi változójának olyan értékrendszere, amely  $d_i$ -vel együtt kielégíti a korlátot  
Pl. az utolsó korlát:  $0 + 0 + c_4 = m + 10 \cdot 0$ , a változók tartománya:  
 $0 \in [0]$ ,  $c_4 \in [0, 1]$ ,  $m \in [1, 2, 3, 4, 5, 6, 7, 8, 9]$   
 $m \in [2, 3, 4, 5, 6, 7, 8, 9]$  értékek feleslegesek!
- Felesleges érték elhagyásával (szűkítéssel) ekvivalens CSP-t kapunk
- SMM kezdeti tartománya; és megszűkítve, tovább már nem szűkíthető:

c1: 01

c2: 01

c3: 01

c4: 01

s: 123456789

e: 0123456789

n: 0123456789

d: 0123456789

m: 123456789

o: 0123456789

r: 0123456789

y: 0123456789

szűkítés az összes  
lehetséges  
korláttal, ameddig  
sikerül:

c1: 01

c2: 01

c3: 01

c4: 1

s: 89

e: 0123456789

n: 0123456789

d: 0123456789

m: 1

o: 01

r: 0123456789

y: 0123456789

## CSP tevékenységek – címkézés (labeling)

- Tovább már nem szűkíthető CSP esetén vizsgáljuk a többértelműséget:
- Többértelműség: van legalább két elemet tartalmazó tartomány, és egyik tartomány sem üres
- **Címkézés (elágazás):**
  - 1 kiválasztunk egy többértelmű változót (pl. a legkisebb tartományút),
  - 2 a tartományt két vagy több részre osztjuk (választási pont),

c1: 01	Két új	c1: 0		c1: 1
c2: 01	CSP-t	c2: 01		c2: 01
c3: 01	készítünk:	c3: 01		c3: 01
c4: 1	c1=0 és	c4: 1	és	c4: 1
s: 89	c1>0	s: 89		s: 89
e: 0123456789	esetek:	e: 0123456789		e: 0123456789
...		...		...

- 3 az egyes választásokat mind megoldjuk, mint új CSP-eket.

## CSP tevékenységek – visszalépés

- Ha nincs többértelműség, és a tartományok nem szűkíthetők tovább, két eset lehet:
  - Ha valamely változó tartománya üres, nincs megoldás ezen az ágon
  - Ha minden változó tartománya egy elemű, előállt egy megoldás

Az SMM CSP megoldás folyamata összefoglalva:

- 1 Felvesszük a változók és segédváltozók tartományait, ez az első *állapotunk*, ezt betesszük az *S* listába
- 2 Ha az *S* lista üres, megállunk, nincs több megoldás
- 3 Az *S* listából kivesszünk egy állapotot, és szűkítjük, ameddig csak lehet
- 4 Ha van üres tartományú változó, akkor az állapotból nem jutunk megoldáshoz, folytatjuk a 2. lépéssel
- 5 Ha nincs többértelmű változó az állapotban, az állapot egy megoldás, eltesszük, folytatjuk a 2. lépéssel
- 6 Valamelyik többértelmű változó tartományát részekre osztjuk, az így keletkező állapotokat visszatesszük a listába, folytatjuk a 2. lépéssel

## SMM CSP megoldással – részlet

smm99.erl – SMM CSP megoldásának alapjai

```
% @type state() = {varname(), domain()}.
```

```
% @type varname() = any().
```

```
% @type domain() = [d()].
```

```
% @spec initial_state() -> St::state().
```

```
% St describes the variables of the SEND MORE MONEY problem.
```

```
initial_state() ->
```

```
    VarNames = [0,c1,c2,c3,c4,s,e,n,d,m,o,r,y],
```

```
    From      = [0, 0, 0, 0, 0,1,0,0,0,1,0,0,0],
```

```
    To        = [0, 1, 1, 1, 1,9,9,9,9,9,9,9,9],
```

```
    [ {V,lists:seq(F,T)} ||
```

```
        {V,{F,T}} <- lists:zip(VarNames, lists:zip(From, To))] .
```

```
% @spec smm() -> [octet()].
```

```
smm() ->
```

```
    St = initial_state(),
```

```
    process(St, [], []).
```

## SMM CSP megoldással – részlet (2)

smm99.erl – SMM CSP megoldásának fő függvénye

```
% process(St0::state(),Sts::[state()],Sols0::[octet()])->Sols::[octet()].
% Sols = Sols1++Sols0 s.t. Sols1 are the sols obtained from [St0|Sts].
process(...) -> ...;
process(St0, Sts, Sols0) ->
    St = narrow_domains(St0),
    DomSizes = [ length(Dom) || {_,Dom} <- St ],
    Max = lists:max(DomSizes),
    Min = lists:min(DomSizes),
    if Min == 0 ->                % there are empty domains
        process(final, Sts, Sols0);
    (St /= St0) ->                % state changed
        process(St, Sts, Sols0);
    Max == 1 ->                  % all domains singletons, solution found
        Sol = [Val || {_,[Val]} <- problem_vars(St)],
        process(final, Sts, [Sol|Sols0]);
    true ->
        {CSt1, CSt2} = make_choice(St),    % labeling
        process(CSt1, [CSt2|Sts], Sols0)
end.
```

# Tartalom

6

## Haladó Erlang

- Rekurzív adatstruktúrák
- Pontos megoldás funkcionális megközelítésben
- **Listák használata: futamok**
- Rekurzió fajtái
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Mohó és lusta kiértékelés



## Futam definíció

- *Futam*: olyan nem üres lista, amelynek szomszédos elemei adott feltételnek megfelelnek
- A feltételt az előző és az aktuális elemre alkalmazandó *predikátumként* adjuk át a futamot előállító függvénynek
  - *Predikátum*: logikai függvény (igaz/hamis), példa:

```
1> P = fun erlang:'<'>/2.  
#Fun<erlang.<.>2>  
2> P(1, 2).  
true  
3> P(2, 2).  
false
```
- Feladat: írjunk olyan Erlang-függvényt, amely egy lista egymás utáni elemeiből képzett (diszjunkt, tovább nem bővíthető) futamok listáját adja eredményül – az elemek eredeti sorrendjének megőrzésével
- Az első, naív változatban egy-egy segédfüggvényt írunk egy lista *első* (prefix) *futamának*, valamint a *maradéklistának* az előállítására (vö. `dp11a_gy3:rampa, lists:splitwith/2`)

## Futamok előállítására – naív változat

- Példa:

```
4> futam:also_futam(P, [1,3,9,5,7,2,5,9,1,6,0,0,3,5,6,2]).  
[1,3,9]
```

### futam.erl – Futamok felsorolása

```
% @type pred() = fun(elem(), elem()) -> bool().
```

```
% @type elem() = any().
```

```
% @spec also_futam(P::pred(), List::[elem()]) -> Fs::[elem()].
```

```
% Fs a List P-t kielégítő első (prefix) futama.
```

```
also_futam(_P, [X]) ->
```

```
  [X];
```

```
also_futam(P, [X|Ys=[Y|_]]) ->
```

```
  case P(X, Y) of
```

```
    false -> [X];
```

```
    true  -> [X|also_futam(P, Ys)]
```

```
end.
```

## Futamok előállítás – naív változat (2)

- Példa:

```
4> futam:elso_futam(P, [1,3,9,5,7,2,5,9,1,6,0,0,3,5,6,2]).  
[1,3,9]  
5> futam:maradek(P, [1,3,9,5,7,2,5,9,1,6,0,0,3,5,6,2]).  
[5,7,2,5,9,1,6,0,0,3,5,6,2]
```

### futam.erl – folytatás

```
% @spec maradek(P::pred(), List::[elem()]) -> Ms::[elem()].  
% Ms a List P-t kielégítő első futam utáni maradéka.  
maradek(_P, [_X]) ->  
  [];  
maradek(P, [X|Ys=[Y|_]]) ->  
  case P(X, Y) of  
    false -> Ys;  
    true  -> maradek(P, Ys)  
  end.
```

## Futamok előállítás – naív változat (3)

- Példa:

```
6> futam:naiv_futamok(P, [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]]  
7> futam:naiv_futamok(P, []).  
[]  
8> futam:naiv_futamok(P, [1]).  
[[1]]
```

### futam.erl – folytatás

```
% @spec naiv_futamok(Pred::pred(), List::[elem()])  
%           -> Lists::[[elem()]].  
% Lists a List szomszédos elemeiből álló, Pred-et kielégítő  
% futamok listája.  
naiv_futamok(_P, []) -> [];  
naiv_futamok(P, List) -> Fs = elso_futam(P, List),  
                          Ms = maradek(P, List),  
                          [Fs|naiv_futamok(P, Ms)].
```

## Futamok előállítás – hatékonyabb változat

- Pazarlás kétszer megkeresni az első futamot, a korábbi példa:  
`4> futam:elso_futam(P, [1,3,9,5,7,2,5,9,1,6,0,0,3,5,6,2]).`  
`[1,3,9]`  
`5> futam:maradek(P, [1,3,9,5,7,2,5,9,1,6,0,0,3,5,6,2]).`  
`[5,7,2,5,9,1,6,0,0,3,5,6,2]`
- Kezeljük az első futamot és a maradékot *egyetlen párként*:  
`9> futam:futam_maradek(P, [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]}`

```
% @spec futam_maradek(P::pred(), L::[elem()]) ->
%   {Fs::[elem()],Ms::[elem()]}.
% Fs ::= elso_futam(P, L) és Ms ::= maradek(P, L).
futam_maradek(_P, [X])          -> {[X], []};
futam_maradek(P, [X|Ys=[Y|_]]) ->
  case P(X, Y) of
    true  -> {Fs, Ms} = futam_maradek(P, Ys),
              {[X|Fs], Ms};
    false -> {[X], Ys}
  end.
```

## Futamok előállítás – hatékonyabb változat (2)

futam.erl – folytatás

```
% @spec *futamok(Pred::pred(), L::[elem()]) -> Lists:[[elem()]].
naiv_futamok(_P, []) ->
  [];
naiv_futamok(P, L) ->
  Fs = elso_futam(P, L),
  Ms = maradek(P, L),
  [Fs|naiv_futamok(P, Ms)].

futamok(_P, []) ->
  [];
futamok(P, L) ->
  {Fs, Ms} = futam_maradek(P, L),
  [Fs|futamok(P, Ms)].
```

- Példa futam\_maradek felhasználására: számtani sorozatok gyűjtése

```
10> futam:difek([1,3,5,7,7,5,3,1,1,1,1,2]).
[[1,3,5,7],[7,5,3,1],[1,1,1],[2]]
```

```
% @spec difek(Xs::[number()]) -> Dss::[[number()]].
% Dss az Xs számtani sorozatot alkotó részlistáinak listája.
difek([X1,X2|_]=L) ->
  {Fs,Ms} = futam_maradek(fun(A, B) -> B-A=:X2-X1 end, L),
  [Fs|difek(Ms)];
difek(_]=L) -> [L]; difek([]) -> [].
```

# Tartalom

## 6 Haladó Erlang

- Rekurzív adatstruktúrák
- Pontos megoldás funkcionális megközelítésben
- Listák használata: futamok
- **Rekurzió fajtái**
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Mohó és lusta kiértékelés

## Rekurzió alapesetek

- Lineáris rekurzió

Példa: lista összegének meghatározása

`rek.erl` – Rekurzió példák

```
sum([])      -> 0;
sum([H|T])  -> H + sum(T).
```

- Elágazó rekurzió (Tree recursion)

Példa: bináris fa leveleinek száma

```
% @type btree() = leaf | {any(), btree(), btree()}.
leaves(leaf)      -> 1;
leaves({_, Lt, Rt}) -> leaves(Lt) + leaves(Rt).
```

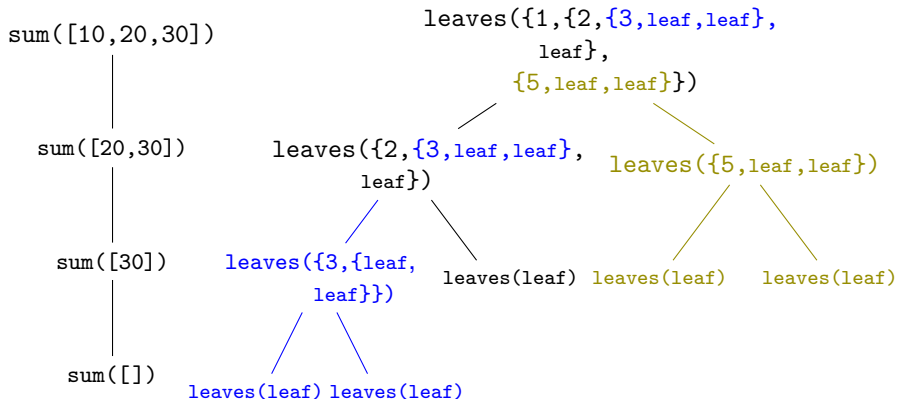
- Mindkettőből *rekurzív folyamat* jön létre, ha alkalmazzuk: minden egyes rekurzív hívás mélyíti a vermet
- Például `sum/1` az egész listát kiteríti a vermen: `sum([1,2,3])` →

`1 + sum([2,3])` → `1 + (2 + sum([3]))` → `1 + (2 + (3 + sum([])))`



## Rekurzív folyamat erőforrásigénye

- Hívási fa (call graph, CG fa): futás során meghívott függvények



- A lépések száma főként a *CG méretének* függvénye
- A tárigény (veremigény) főként a *CG mélységének* függvénye<sup>7</sup>

<sup>7</sup>itt lineáris függvénye

## Jobbrekurzió, iteráció

Gyakran érdemes akkumulátorok bevezetésével jobbrekurzióvá alakítani

- Példa: lista összegének meghatározása

$\text{sumi}(L) \rightarrow \text{sumi}(L, 0)$ .

$\text{sumi}([], N) \rightarrow N$ ;

$\text{sumi}([H|T], N) \rightarrow \text{sumi}(T, N+H)$ .

- A segédfüggvényt jobb nem exportálni, hogy elrejtjük az akkumulátort
- A jobbrekurzióból *iteratív folyamat* hozható létre, amely nem mélyíti a vermet:  $\text{sumi}/2$  tárigénye konstans:  $\text{sumi}([1,2,3], 0) \rightarrow \text{sumi}([2,3], 1) \rightarrow \text{sum}([3], 3) \rightarrow \text{sum}([], 6)$
- Ne tévesszük össze egymással a rekurzív számítási folyamatot és a rekurzív függvényt, eljárást!
  - Rekurzív függvény esetén csupán a szintaxisról van szó, arról, hogy hivatkozik-e a függvény, eljárás *önmagára*
  - Folyamat esetében viszont a folyamat menetéről, lefolyásáról beszélünk
- Ha egy függvény *jobbrekurzív (tail-recursive)*, a megfelelő folyamat – az értelmező/fordító jóságától függően – lehet iteratív

## Rekurzív folyamat erőforrásigénye – Példák

Példa <sup>8</sup>	Lépések (CG méret)	CG mélység	Tárigény $\approx$ mélység* állapot
<code>sum(lists:seq(1, n))</code>	$\Theta(n)$	$\Theta(n)$	$\Theta(n) * \Theta(1)$
<code>sumi(lists:seq(1, n))</code>	$\Theta(n)$	$\Theta(1)$ (konst.)	$\Theta(1) * \Theta(1)$
SEND+MORE=MONEY kimerítő keresés, itt $n = 8$	$\Theta(10^n)$	$\Theta(n)$	$\Theta(n) * \Theta(n) = \Theta(n * n)$
$(n * n)$ -es Sudoku kimerítő keresés, tipikusan $n = 9$	$\Theta(n^{n*n})$	$\Theta(n * n)$	$\Theta(n^2) * \Theta(n^2) = \Theta(n^4)$

- A rekurzióból fakadó tárigény lehet jelentős is (vö `sum/1`, `sumi/1`), és lehet elhanyagolható is a lépésekhez képest (SMM, Sudoku)
- Az eljárások, függvények olyan *minták*, amelyek megszabják a számítási folyamatok, processzek menetét, *lokális* viselkedését
- Egy számítási folyamat *globális* viselkedését (pl. idő- és tárigény) általában nehéz megbecsülni, de törekednünk kell rá

<sup>8</sup> $f(n) = \Theta(g(n))$  jelentése:  $g(n) \cdot k_1 \leq f(n) \leq g(n) \cdot k_2$  valamilyen  $k_1, k_2 > 0$ -ra

## A jobbrekurzió mindig előnyösebb? Nem!

- A jobbrekurzív `sum(L1)` össz-tárigénye konstans, a lineáris-rekurzív `sum(L1)` össz-tárigénye  $\Theta(\text{length}(L1))$
- Melyiknek alacsonyabb az össz-tárigénye?
  - `bevezeto:append(L1,L2)`
  - `R1=lists:reverse(L1),bevezeto:revapp(R1,L2)` % *jobbrek.*
- `append` kiteríti `L1` elemeit a vermen, ennek tárigénye  $\Theta(\text{length}(L1))$ , majd ezeket `L2` elé fűzi, így tárigénye  $\Theta(\text{length}(L1)+\text{length}(L2))$
- `revapp(R1,L2)` iteratív számítási folyamat, nem mélyíti a vermet, **de** `revapp` felépíti az `L1++L2` akkumulátort, ennek tárigénye szintén  $\Theta(\text{length}(L1)+\text{length}(L2))$
- **A jobbrekurzív `revapp` össz-tárigénye hasonló, mint a lineáris-rekurzív `append` függvényé!**
- Ha a jobbrekurzív esetben a `reverse` után nem törli *azonnal* a szemétgyűjtő (garbage collector) `L1`-et, akkor egyszerre kell tárolni az `L1`, `R1`, `L2`, `L1++L2` listákat, így tárigénye nagyobb is lehet (pl. Erlang R14B)
- Ha az akku. mérete nem konstans, meggondolandó a jobbrekurzió. . .

## Elágazó rekurzió példa: Fibonacci-sorozat

- Amikor hierarchikusan strukturált adatokon kell műveleteket végezni, pl. egy fát kell bejárni, akkor az elágazó rekurzió nagyon is természetes és hasznos eszköz
- Az elágazó rekurzió numerikus számításoknál az algoritmus első megfogalmazásakor is hasznos lehet; példa: írjuk át a Fibonacci-számok matematikai definícióját programmá – 0,0,1,1,2,3,5,8,13,...

$$F(n) = \begin{cases} 0 & \text{ha } n = 0, \\ 1 & \text{ha } n = 1, \\ F(n-1) + F(n-2) & \text{különb.} \end{cases}$$

Naív Fibonacci, előfelt.:  $N \in \mathcal{N}$

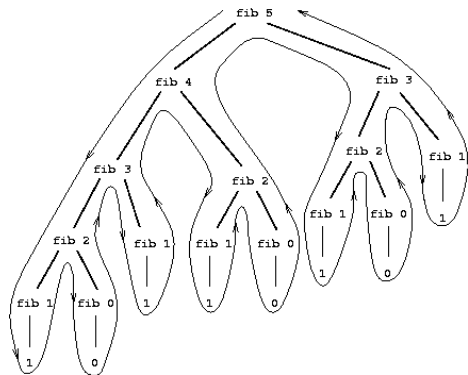
fib(0) -> 0;

fib(1) -> 1;

fib(N) -> fib(N-2) + fib(N-1).

- Ha már értjük a feladatot, az első, rossz hatékonyságú változatot könnyebb átírni jó, hatékony programmá. Az elágazó rekurzió segíthet a feladat megértésében.
- *Hivatkozás:* Structure and Interpretation of Computer Programs, 2nd ed., by H. Abelson, G. J. Sussman, J. Sussman, The MIT Press, 1996

## Elágazó rekurzió példa: Fibonacci-sorozat (2)



- Elágazóan rekurzív folyamat hívási fája  $\text{fib}(5)$  kiszámításakor
- Alkalmatlan a Fibonacci-számok előállítására
- A  $F(n)$  meghatározásához pontosan  $F(n+1)$  levélből álló fát kell bejárni, azaz ennyiszor kell meghatározni  $F(0)$ -at vagy  $F(1)$ -et

- $F(n)$  exponenciálisan nő  $n$ -nel:  $\lim_{n \rightarrow \infty} \frac{F(n+1)}{F(n)} = \varphi$ , ahol  $\varphi = (1 + \sqrt{5})/2 \approx 1.61803$ , az *arany metszés* arányzáma

## Elágazó rekurzió példa: Fibonacci-sorozat (3)

- A lépések száma tehát  $F(n)$ -nel együtt exponenciálisan nő  $n$ -nel
- A tárigeny ugyanakkor csak lineárisan nő  $n$ -nel, mert csak azt kell nyilvántartani, hogy hányadik szinten járunk a fában
- 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

```
fibi(0) -> 0;                                % fibi(N) az N. Fibonacci-szám.  
fibi(N) -> fibi(N-1, 1, 0).
```

```
% fibi(N,A,B) az  $A \leftarrow A+B$ ,  $B \leftarrow A$  trafó  $N$ -szeri ismétlése utáni  $A$ .  
fibi(0, A, _B) -> A;  
fibi(I, A, B) -> fibi(I-1, B+A, A).
```

- A Fibonacci-példában a lépések száma elágazó rekurziónál  $n$ -nel exponenciálisan, lineáris rekurziónál  $n$ -nel arányosan nőtt!
- Pl. a `tree:leaves/1` függvény is lineáris rekurzióvá alakítható, **de** ezzel már nem javítható a hatékonysága: valamilyen LIFO tárolót kellene használni a mélységi bejáráshoz a rendszer stackje helyett

## 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).
- Ellenpélda: `fib(-1)` „végtelen” ciklus; bár a paraméter csökken, az **abszolút értéke** nem csökken 0-ba, csak ha a paraméter nemnegatív
- Bizonyítása rekurzió esetén egyszerű, *strukturális indukcióval* lehetséges, azaz visszavezethető a teljes indukcióra valamilyen *strukturális tulajdonság* szerint
- Csak meg kell választanunk a strukturális tulajdonságot, amire vonatkoztatjuk az indukciót; pl. a `fib/1` az  $N = 0$  paraméterre leáll, de a 0 nem a legkisebb szám, de *abszolút értéke* a legkisebb  $\rightarrow$  az indukciót  $\text{abs}(N)$ -re vonatkoztatjuk
- A `map` példáján mutatjuk be



## Programhelyesség informális igazolása (folyt.)

```
% @spec map(fun(A) -> B, [A]) -> [B].  
map(_F, []) -> [];  
map(F, [X|Xs]) -> [F(X)|map(F, Xs)].
```

- 1 A strukturális tulajdonság itt a lista hossza
- 2 A függvény funkcionálisan helyes, mert
  - belátjuk, hogy a függvény jól transzformálja az üres listát;
  - belátjuk, hogy az  $F$  jól transzformálja a lista első elemét (a fejét);
  - indukciós feltevés: a függvény jól transzformálja az eggyel rövidebb listát (a lista farkát);
  - belátjuk, hogy a fej transzformálásával kapott elem és a fark transzformálásával kapott lista összefűzése a várt listát adja.
- 3 A kiértékelés véges számú lépésben befejeződik, mert
  - a lista (mohó kiértékelés mellett!) véges,
  - a függvényt a *rekurzív ágban* minden lépésben egyre rövidülő listára alkalmazzuk (strukturális tulajdonság csökken), és
  - a rekurziót előbb-utóbb leállítjuk (ui. kezeljük az *alapesetet*, ahol a strukturális tulajdonság zérus, van rekurziót nem tartalmazó klóz).

# Tartalom

## 6 Haladó Erlang

- Rekurzív adatstruktúrák
- Pontos megoldás funkcionális megközelítésben
- Listák használata: futamok
- Rekurzió fajtái
- **Halmazműveletek (rendezetlen listával)**
- Generikus keresőfák
- Mohó és lusta kiértékelés

## Tagsági vizsgálat

- A halmazt itt egy rendezetlen listával ábrázoljuk
- A műveletek sokkal hatékonyabbak volnának rendezett adatszerkezettel (pl. rendezett lista, keresőfa, hash)
- STDLIB: sets, ordsets, gb\_sets modulok

set.erl – Halmazkezelő függvények

```
% @type set() = list().
```

```
% empty() az üres halmaz.
```

```
empty() ->  
[].
```

```
% Az absztrakció miatt szükséges:  
% ábrázolástól független interfész.
```

```
% isMember(X, Ys) igaz, ha az X elem benne van az Ys halmazban.
```

```
isMember(_, []) ->
```

```
    false;
```

```
isMember(X, [Y|Ys]) ->
```

```
    X:=Y orelse isMember(X, Ys).
```

- **Megjegyzés:** orelse lusta kiértékelésű

## Új elem berakása egy halmazba, listából halmaz

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

`set.erl` – folytatás

```
% @spec newMember(X::any(), Xs::set()) -> Xs2::set().
```

```
% Xs2 halmaz az Xs halmaz és az [X] halmaz uniója.
```

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

- `listToSet` listát halmazzá alakít a duplikátumok törlésével; naív (lassú)

```
% @spec listToSet(list()) -> set().
```

```
% listToSet(Xs) az Xs lista elemeinek halmaza.
```

```
listToSet([]) ->  
  [];  
listToSet([X|Xs]) ->  
  newMember(X, listToSet(Xs)).
```

# Halmazműveletek

- 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`,  $2^S$ ).
- Otthoni gyakorlásra: halmazműveletek megvalósítása rendezett listákkal, illetve fákkal.  
A vizsgán lehetnek ilyen feladatok. . .

## Unió, metszet

### set.erl – folytatás

```
% @spec union(Xs::set(), Ys::set()) -> Zs::set().
```

```
% Zs az Xs és Ys halmazok uniója.
```

```
union([], Ys)      -> Ys;
union([X|Xs], Ys) ->
  newMember(X, union(Xs, Ys)).
```

```
union2(Xs, Ys) ->
  foldr(fun newMember/2, Ys, Xs).
```

```
% @spec intersect(Xs::set(), Ys::set()) -> Zs::set().
```

```
% Zs az Xs és Ys halmazok metszete.
```

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

```
intersect3(Xs, Ys) ->
  [ X || X <- Xs,
    isMember(X, Ys)
  ].
```

## Részhalmaza-e, egyenlők-e

set.erl – folytatás

```
% @spec isSubset(Xs::set(), Ys::set()) -> B::bool().
```

```
% B igaz, ha Xs részhalmaza Ys-nek.
```

```
isSubset([], _) ->
```

```
    true;
```

```
isSubset([X|Xs], Ys) ->
```

```
    isMember(X, Ys) andalso isSubset(Xs, Ys).
```

```
% @spec isEqual(Xs::set(), Ys::set()) -> B::bool().
```

```
% B igaz, ha Xs és Ys elemei azonosak.
```

```
isEqual(Xs, Ys) ->
```

```
    isSubset(Xs, Ys) andalso isSubset(Ys, Xs).
```

- isSubset lassú a rendezetlenség miatt
- andalso lusta kiértékelésű
- 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

- Az  $S$  halmaz hatványhalmazának nevezzük az  $S$  összes részhalmazának a halmazát, jelölés itt:  $2^S$
- $S$  hatványhalmazát *rekurzívan* például úgy állíthatjuk elő, hogy kivesszünk  $S$ -ből egy  $x$  elemet, majd előállítjuk az  $S \setminus \{x\}$  hatványhalmazát
- Például  $S = \{10, 20, 30\}$ ,  $x \leftarrow 10$ ,  $2^{S \setminus \{x\}} = \{\{\}, \{20\}, \{30\}, \{20, 30\}\}$
- Ha tetszőleges  $T$  halmazra  $T \subseteq S \setminus \{x\}$ , akkor  $T \subseteq S$  és  $T \cup \{x\} \subseteq S$ , azaz mind  $T$ , mind  $T \cup \{x\}$  eleme  $S$  hatványhalmazának
- Vagyis  $2^{\{10, 20, 30\}} =$

$$\{\{\}, \{20\}, \{30\}, \{20, 30\}\} \cup \{\{\} \cup \{10\}, \{20\} \cup \{10\}, \{30\} \cup \{10\}, \{20, 30\} \cup \{10\}\}$$

*% powerSet\*(S) az S halmaz hatványhalmaza.*

```
powerSet1([]) ->
  [[]];
powerSet1([X|Xs]) ->
  P = powerSet1(Xs),
  P ++ [ [X|Ys] || Ys <- P ].
```

```
powerSet2(Xs) -> % jobbrekurzívan
  foldl(fun(X, P) ->
    P ++ [ [X|Ys] || Ys <- P ]
  end,
  [[]],
  Xs).
```



## Halmaz hatványhalmaza – hatékonyabb változat

- `A P ++ [ [X|Ys] || Ys <- P ]` művelet hatékonyabbá tehető

`set.erl` – folytatás

```
% @spec insAll(X::any(), Yss::[[any()]], Zss::[[any()]]) -> Xss::[[any()]]
```

```
% Xss az Yss lista Ys elemeinek Zss elé fűzött
```

```
% listája, amelyben minden Ys elem elé X van beszúrva.
```

```
insAll(_X, [], Zss) ->
```

```
  Zss;
```

```
insAll(X, [Ys|Yss], Zss) ->
```

```
  insAll(X, Yss, [[X|Ys]|Zss]).
```

```
powerSet3([]) ->
```

```
  [[]];
```

```
powerSet3([X|Xs]) ->
```

```
  P = powerSet3(Xs),
```

```
  insAll(X, P, P).  % [ [X|Ys] || Ys <- P ] ++ P kiváltására
```

# Tartalom

## 6 Haladó Erlang

- Rekurzív adatstruktúrák
- Pontos megoldás funkcionális megközelítésben
- Listák használata: futamok
- Rekurzió fajtái
- Halmazműveletek (rendezetlen listával)
- **Generikus keresőfák**
- Mohó és lusta kiértékelés

## 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.

### Megjegyzések:

- `gtree:set_to_list/1` funkcióban azonos `tree:to_list_infix/1` függvénnyel, de hatékonyabb: nincs benne összefűzés ( $L1++L2$ ), csak építés (`[H|T]`)
- ```
1> gtree:list_to_set([3,1,5,4,2,1]).
{3,{1,leaf,{2,leaf,leaf}},{5,{4,leaf,leaf},leaf}}
```

```
2> io:format("~10p~n",
             [gtree:list_to_map([3,a],[1,a],[5,a],[4,b],[2,b],[1,x]])]).
{{3,a},
 {1,x},
 leaf,
 {{2,b},
 leaf,
 leaf}},
 ...
```

# Tartalom

6

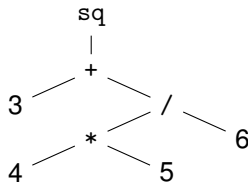
## Haladó Erlang

- Rekurzív adatstruktúrák
- Pontos megoldás funkcionális megközelítésben
- Listák használata: futamok
- Rekurzió fajtái
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Mohó és lusta kiértékelés

# Összetett kifejezés kiértékelése

- Egy összetett kifejezést az Erlang két lépésben értékeli ki, **mohó** kiértékeléssel; az alábbi rekurzív kiértékelési szabállyal:
  - Először** kiértékeli az operátort (műveleti jelet, függvényjelet) és az argumentumait,
  - majd **ezután** alkalmazza az operátort az argumentumokra.
- A kifejezéseket *kifejezésfával* ábrázolhatjuk
  - Hasonló a Prolog-kifejezés ábrázolásához:
 

```
| ?- write_canonical(sq(3+4*5/6)).  
sq(+ (3, / (* (4, 5), 6)))
```
  - A mohó kiértékelés során az operandusok alulról fölfelé „terjednek”
- Felhasználói függvény mohó alkalmazása (fent 2. pont):
  - a függvény törzsében a formális paraméterek összes előfordulását lecseréli a megfelelő aktuális paraméterre,
  - majd kiértékeli a függvény törzsét.



## Függvényalkalmazás mohó kiértékelése

Tekintsük 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 kiértékelés esetén minden lépésben egy részkifejezést egy vele egyenértékű kifejezéssel helyettesítünk. Pl. az  $f(5)$  mohó kiértékelése:

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

- A függvényalkalmazás itt bemutatott *helyettesítési modellje*, az „egyenlők helyettesítése egyenlőkkel” (*equals replaced by equals*) segíti a függvényalkalmazás *jelentésének* megértését
- Olyan esetekben alkalmazható, amikor egy függvény *jelentése független* a környezetétől (pl. ha minden mellékhatás kizárva)
- Az fordítók rendszerint bonyolultabb modell szerint működnek

## Lusta kiértékelés

- 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
- Példa: az  $f(5)$  lusta kiértékelése:

```
f(5) → sumsq(5+1, 5*2) → sq(5+1) + sq(5*2) → (5+1)*(5+1) +  
(5*2)*(5*2) → 6*(5+1) + (5*2)*(5*2) → 6*6 + (5*2)*(5*2) → 36 +  
(5*2)*(5*2) → 36 + 10*(5*2) → 36 + 10*10 → 36 + 100 → 136
```

- Példa: a `false andalso f(5) > 100` lusta kiértékelése:

```
false andalso f(5) > 100 → false
```

## Mohó és lusta kiértékelés

- 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.



## Lusta kiértékelés Erlangban: lusta farkú lista

- Ismétlés: `% @type erlang:list() = [] | [any()|list()]`.
- A `[H|T]` egy speciális szintaxisú kételemű ennes, nemcsak listákra használhatjuk:
 

```
1> [1|[2]].
[1,2]    % Lista, mert a | utáni rész lista.
2> [1|[2|[]]].
[1,2]    % Lista, mint az előző.
3> [1|2].
[1|2]    % Egy kételemű ennes, mert a | utáni rész nem lista.
```
- A következő fóliákon az átláthatóság kedvéért a listaszintaxist használjuk egy kételemű ennesre, a lusta listára

### lazy.erl – Lusta farkú lista

```
% @type lazy:list() = [] | [any()|fun() -> lazy:list()]
```

- A fenti szerkezetben a második tag (fark) *késleltett kiértékelésű* (delayed evaluation)
- Teljesen lusta lista: `verylazy.erl` (nem tananyag)
 

```
% @type verylazy:list() = fun() -> ([] | [any()|verylazy:list()])
```

## Lusta farkú lista építése

- Végtelen számsorozat:

lazy.erl – folytatás

```
% @spec infseq(N::integer())
%       -> lazy:list().
infseq(N) ->
  [N|fun() -> infseq(N+1) end].
```

- Példák:

```
1> lazy:infseq(0).
[0|#Fun<lazy.1.65678590>]
2> T1 = tl(lazy:infseq(0)).
#Fun<lazy.1.65678590>
3> T1().
[1|#Fun<lazy.1.65678590>]
```

- Véges számsorozat:

lazy.erl – folytatás

```
% @spec seq(M::integer(),
%         N::integer())
%         -> lazy:list().
seq(M, N) when M =< N ->
  [M|fun() -> seq(M+1, N) end];
seq(_, _) ->
  [].
```

- Példák:

```
1> lazy:seq(1,1).
[1|#Fun<lazy.0.35745118>]
2> tl(lazy:seq(1,1)).
#Fun<lazy.0.35745118>
3> (tl(lazy:seq(1,1)))().
[]
```

## Erlang-lista konvertálása

Erlang-listából lusta lista:

- Nagyon gyors: egyetlen függvényhívás

```
% @spec cons(erlang:list())
%         -> lazy:list().
cons([]) ->
    [];
cons([H|T]) ->
    [H|fun() -> cons(T) end].
```

```
1> lazy:cons([1,2]).
[1|#Fun<lazy.10.66878903>]
2> T2 = tl(lazy:cons([1,2])).
#Fun<lazy.10.66878903>
3> T2().
[2|#Fun<lazy.10.66878903>]
4> (tl(T2()))().
[]
```

Lusta listából Erlang-lista:

- Csak az első  $N$  elemét értékeljük ki: lehet, hogy végtelen

```
% @spec take(lazy:list(),
%           N::integer())
%         -> erlang:list().
take(_, 0) -> [];
take([], _) -> [];
take([H|_], 1) -> [H]; % optim.
take([H|T], N) ->
    [H|take(T(), N-1)].
```

```
1> lazy:take(lazy:infseq(0), 5).
[0,1,2,3,4]
2> lazy:take(lazy:cons([1,2]), 5).
[1,2]
```

- 3. klózban  $T()$  felesleges lenne

## Eddig bemutatott függvények lusta listára – iteratív sum

- Lista összegzése (csak véges listára)

lazy.erl – folytatás

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

```
% @spec sum(lazy:list(), number()) -> number().
```

```
sum([], X) -> X;
```

```
sum([H|F], X) -> sum(F(), H+X).    % jobbrekurzív!
```

- Összehasonlítás:
  - `lists:sum(lists:seq(1,N=10000000))`  
mohó, gyors, tárigénye lineáris N-ben
  - `lazy:sum(lazy:seq(1,N=10000000))`  
lusta, kicsit lassabb, tárigénye kb. konstans (korlátos számok esetén)
- Általánosabban a lusta lista és a mohó Erlang-lista összehasonlítása:
  - Tárigénye csak a kiértékelt résznek van
  - Lusta lista *teljes* kiértékelése sokkal lassabb is lehet (késleltetés)
  - De időigénye alacsonyabb *lehet*, ha nem kell teljesen kiértékelni

## Eddig bemutatott függvények lusta listára – filter, append

- Motiváció: listanézet, ++ nem alkalmazható; a lusta szintaxis elrejtése

lazy.erl – folytatás

```
% filter(fun(any()) -> bool(), lazy:list()) -> lazy:list().  
% Kicsit mohó, az eredménylista fejéig kiértékeli a listát  
filter(_, []) ->  
    [];  
filter(P, [H|T]) ->  
    case P(H) of  
        true -> [H|fun() -> filter(P, T()) end];  
        false -> filter(P, T()) % megkeressük az eredmény fejét  
    end.  
  
% @spec append(lazy:list(), lazy:list()) -> lazy:list().  
append([], L2) -> L2;  
append([H|T], L2) -> [H|fun() -> append(T(), L2) end].
```

## Nevezetes számsorozatok

- Fibonacci-sorozat

```
% @spec fibs(integer(), integer()) -> lazy:list.  
fibs(Cur, Next) -> [Cur|fun() -> fibs(Next, Cur+Next) end].
```

```
1> lazy:take(lazy:fibs(0,1),10).  
[0,1,1,2,3,5,8,13,21,34]
```

- Eratoszteni szita

```
% @spec sift(Prime::integer(), L::lazy:list()) -> L2::lazy:list().  
% L2 lista L lista azon elemei, melyek nem oszthatóak Prime-mal.  
sift(Prime, L) -> filter(fun(N) -> N rem Prime /= 0 end, L).
```

```
% @spec sieve(L1::lazy:list()) -> L2::lazy:list().  
% L2 lista az L1 végtelen lista szitálása (üres listára hibát ad).  
sieve([H|T]) -> [H|fun() -> sieve(sift(H, T())) end].
```

```
1> lazy:take(lazy:sieve(lazy:infseq(2)),10).  
[2,3,5,7,11,13,17,19,23,29]
```

## Lusta append alkalmazása: lusta qsort

```
% Csak emlékeztetőül: @spec eqsort(erlang:list()) -> erlang:list().
eqsort([])          -> [];
eqsort([Pivot|Xs]) -> eqsort([X || X <- Xs, X < Pivot])
                    ++ [Pivot|eqsort([X || X <- Xs, X >= Pivot])].
```

```
% @spec qsort(lazy:list()) -> lazy:list().
```

```
qsort([])          -> [];
qsort([Pivot|Xs]) ->
  io:format("hivas: qsort(~w)~n", [take([Pivot|Xs], 100)]),
  Low = fun(X) -> X < Pivot end, High = fun(X) -> X >= Pivot end,
  append(qsort(filter(Low, Xs())),
         [Pivot|fun() -> qsort(filter(High, Xs())) end]).
```

```
1> L=cons([5,3,6,8,1,7]).
[5|#Fun<lazy.10.7...>]
2> take(qsort(L), 1).
hivas: qsort([5,3,6,8,1,7])
hivas: qsort([3,1])
hivas: qsort([1])
[1]
```

```
3> take(qsort(L), 2).
hivas: qsort([5,3,6,8,1,7])
hivas: qsort([3,1])
hivas: qsort([1])
[1,3]
```

```
4> take(qsort(L), 6).
hivas: qsort([5,3,...])
hivas: qsort([3,1])
hivas: qsort([1])
hivas: qsort([6,8,7])
hivas: qsort([8,7])
hivas: qsort([7])
[1,3,5,6,7,8]
```