Az Erlang programozási nyelv

Deklaratív programozás, 2008. őszi félév

Hanák Péter

hanak@iit.bme.hu

Tartalom

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 fügvénynek.
- 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 = futam1(P,X,Xs),
    Ms = maradek1(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} = futam2(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

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

Öt művelet halmazokon

- Öt ismert halmazműveletet definiálunk a továbbiakban (rendezetlen listákkal ábrázolt halmazokon):
  - unió (`union`, $S \bigcup T$),
  - metszet (`intersect`, $S \bigcap T$),
  - 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.
```

Részhalmaza-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* részhalmazának 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 $T \bigcup \{x\} \subseteq S$, azaz mind T, mind $T \bigcup \{x\}$ eleme S hatványhalmazának.
- Miközben a fentieket rekurzív módon alkalmazzuk (azaz fölsoroltatjuk az $S - \{x\}$ stb. részhalmazait), a *már kiválasztott* elemeket gyűjtjük. Egy-egy rekurzív lépésben a gyűjtő vagy változatlan (T), vagy kiegészül az x elemmel ($T \bigcup \{x\}$).

Hatványhalmaz, segédfüggvénnyel

- `pws` kétfelé ágazó rekurziót tartalmaz. Ez gyorsan a verem túlcsordulásához 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.
- `pws(Xs,Zs)` = $\{S \bigcup \texttt{Zs} \mid S \subseteq \texttt{Xs}\}$, azaz az $\texttt{Xs} \bigcup \texttt{Zs}$ 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

Beszúró rendezés

Az `ins1` segédfüggvény az `X` elemet a megfelelő helyre rakja be az `Ys` listában

```
%% @spec ins1(X::any(),Ys::[any()]) -> Zs::[any()]
%% @pre: Ys az =< reláció szerint rendezve van
%%   Zs az =< reláció alapján beszúrt X-szel bővített Ys
ins1(X,[]) -> [X];
ins1(X,[Y|Ys]) when X=<Y ->
    [X,Y|Ys];
ins1(X,[Y|Ys]) ->
    [Y|ins1(X,Ys)].
```

`inssort11`-gyel rekurzívan rendezzük a lista maradékát; végrehajtási ideje $O(n^2)$

```
inssort11([]) -> [];
inssort11([X|Xs]) ->
    ins1(X,inssort11(Xs)).
```

Beszúró 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 beszúró függvénnyel az =<
%%   reláció szerint rendezett Ys
inssort12(_F,[]) ->
    [];
inssort12(F,[X|Xs]) ->
    F(X,inssort12(F,Xs)).
```

Beszúró 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 =< reláció szerint rendezve van
%%   Zs az F reláció alapján beszúrt 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.
```

Beszúró rendezés, generikus változat (folyt.)

```
inssort21(_F,[]) ->
    [];
inssort21(F,[X|Xs]) ->
    ins2(F,X,inssort21(F,Xs)).
```

Jobbrekurzív változat

```
inssort22(_F,[],Zs) ->
    Zs;
inssort22(F,[X|Xs],Zs) ->
    inssort22(F,Xs,ins2(F,X,Zs)).
```

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").
```

Beszúró 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) -> ins2(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* algoritmusa:

qs [] = []

qs(m::xs) = qs([...,x_i,...] | $x_i \leq$m) @ [m] @ qs([...,x_j,...] | $x_j >$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 qsort*(Xs::[any()],Zs::[any()]) -> Ws::[any()]
%% @type pred() = (any(), any()) -> bool()
%%   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 qsort*(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.

Fölülről lefelé haladó összefésülő rendezés

A fölülről 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 algoritmusa (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
%%   Zs 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 `LLLss` 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
  - `mergepairs` 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 `mergepairs`-t azokban a sorokban, amelyekben a `J` új értéket vesz föl, a többi helyen `mergepairs` hívása rekurzív.
- Ne feledjük, hogy `mergepairs`-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)* algoritmusa 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.)

`smsorting` a futamokat ismételten előállítja és összefuttatja

```
%% @spec smsorting(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
smsorting([X|Xs],Lss,K) ->
    Run,Tail = nextrun([X],Xs),
    smsorting(Tail,mergepairs([Run|Lss],K+1),K+2);
smsorting([],Lss,_K) ->
    hd(mergepairs(Lss,0)).

%% @spec smsort(Xs::[any()]) -> Zs::[any()]
%%   Zs az =< reláció szerint rendezett Ys
smsort(Xs) ->
    smsorting(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: segédfüggények

```
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(inssort11,fun inssort11/1,R3000),
    runtime(inssort12,fun(Xs) ->
                          inssort12(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
gsmsort. Length: 3000, time: 463 ms
gsmsort. 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

Összetett kifejezés kiértékelése

Egy összetett kifejezést az Erlang két lépésben értékel 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:

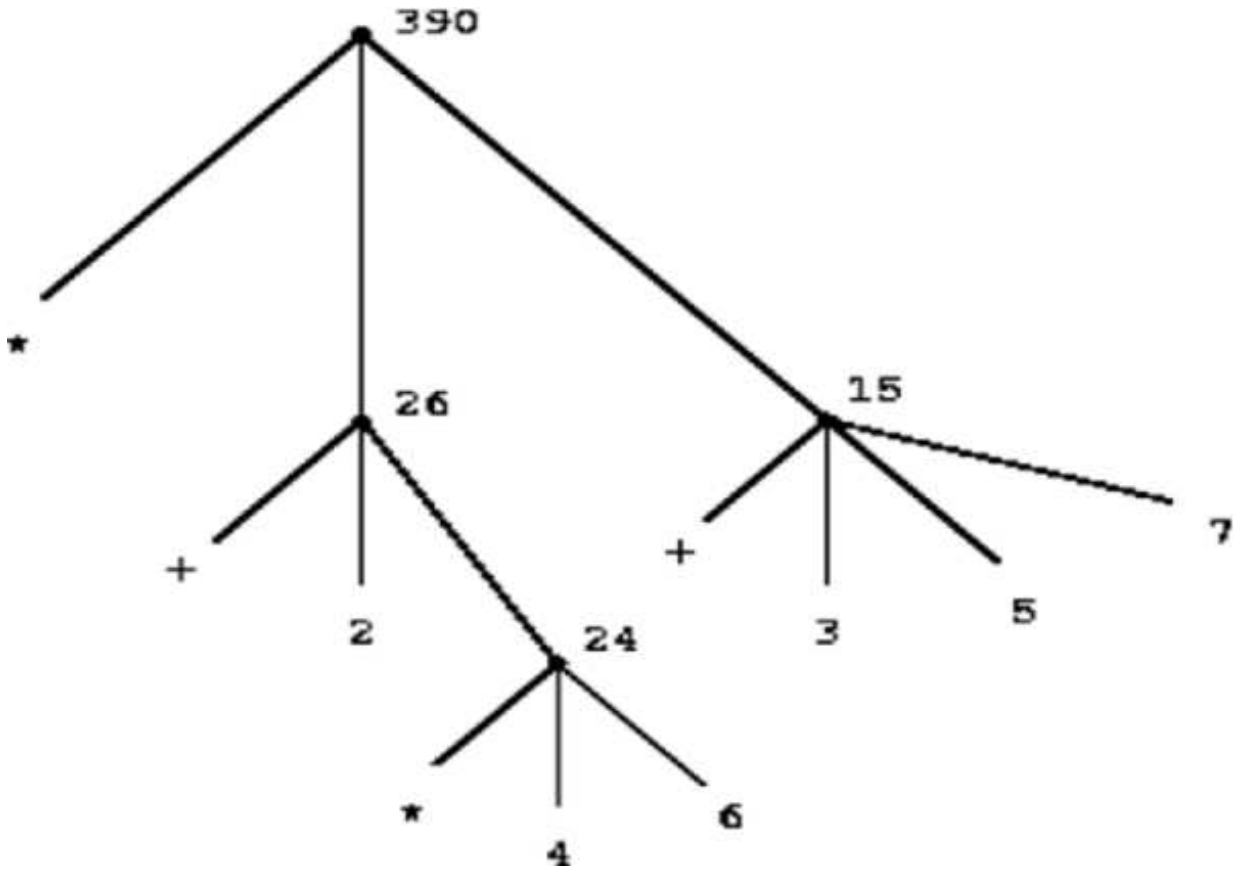
1. az állandó (jelölés) értéke az, amit jelöl,
2. a belső (beépített) művelet a megfelelő gépi utasításokat aktivizálja,
3. a név értéke az, amihez az adott *környezet* köti a nevet.

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 értékel 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!

```
sq(X) -> X * X.
sumsq(X,Y) -> sq(X) + sq(Y).
f(A) -> sumsq(A + 1, A * 2).
```

Mohó és lusta kiértékelés

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

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

- 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!

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

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

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 deklaratívnak, 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 *fölülről 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:

$$0! = 1$$
$$n! = n(n-1)!$$

```
%% @spec factorial1(N::integer()) ->
%%                                 F::integer()
%% @pre N >= 0
%%   F == N!
factorial1(0) -> 1;
factorial1(N) when N>0 ->
    N * factorial1(N-1).
```

- 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 őr 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 jóságától függően – lehet iteratív.

Hivatkozás: Structure and Interpretation of Computer Programs, 2nd ed., by H. Abelsson, 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().
%% @tpye 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 farok transzformálásával kapott lista összefűzése a várt listát adja.
2. A kiértekelés véges számú lépésben befejeződik, mert
   - a lista (mohó kiértékelés mellett!) véges,
   - a függvényt a *rekurzív ágban* 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

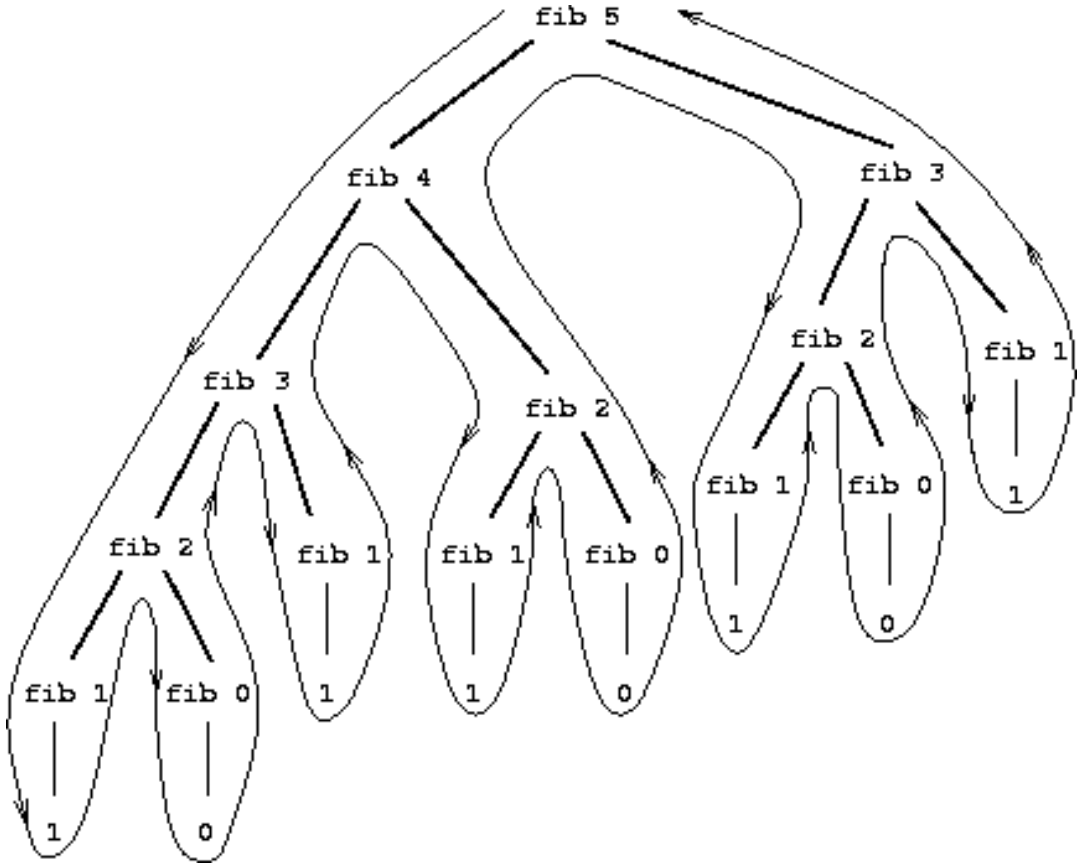
- Az előbb lineáris-rekurzív, ill. lineáris-iteratív folyamatokra láttunk példát (ld. faktoriális kiszámítása kétféleképpen).
- Most az*elágazó rekurzióra* nézünk példát: Fibonacci-számok sorozatát állítjuk elő kétfelé ágazó rekurzióval.
- Bármely Fibonacci-szám az őt megelőző két Fibonacci-szám összege: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

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

```
%% @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. *aranymetszés* 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 klózok 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) -> valtas(Osszeg,5).

valtas(Osszeg,Ermefajta) ->
    if Osszeg < 0 orelse Ermefajta == 0 -> 0;
       Osszeg == 0 -> 1;
       true -> valtas(Osszeg,Ermefajta-1) +
               valtas(Osszeg-cimlet(Ermefajta),Ermefajta)
    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 függvénnyé!

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.

$$b^0 = 1$$
$$b^n = b \cdot b^{n-1}$$

```
%% @spec expt1(B::integer(),
%%             N::integer()) ->
%%                R::integer().
%% @pre N >= 0.
%% B N-edik hatványa R.
expt1(B,0) -> 1;
expt1(B,N) when N>0 ->
              B * expt1(B,N-1).
```

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

Hatványozás – lineáris-iteratív

- A faktoriálisszámításhoz hasonlóan könnyű felírni `expt1/2` lineáris-iteratív változatát.

```
expt2(B,N) when N>=0 -> expt2(B,N,1).

expt2(B,0,P) -> P;
expt2(B,N,P) ->
    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:

$b^0 = 1$

$b^n = (b^{n/2})^2$, ha n páros

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

X0 = expt4(2,0), X1 = expt4(2,1),
X2 = expt4(2,2), X4 = expt4(2,4),
X0,X1,X2,X4. ??
```

Hatványozás – logaritmikus, konstans tárigényű (folyt.)

- Tesztelési eredmények

```
{X0, X1, X2, X4} ==
{ 1,  2,  2,  2}.
```

- Sajnos, ez a változat ROSSZ, az `R=1` kezdőértékezés miatt.

```
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.
```

- Javítsuk!

Hatványozás – logaritmikus, konstans tárigényű (folyt.)

- Konstans tárigényű, iteratív változat, javított

```
expt5(_B,0) -> 1;                    %% Kell ez a klóz!
expt5(B,N) when N>0 -> expt5(B,N,B).

expt5(_B,1,R) -> R;
expt5(B,N,R) ->
    case even(N) of
        true ->
            expt5(B,N div 2,R*R);
        false ->
            expt5(B,N-1,R*B)
    end.
```

Hatványozás – logaritmikus, konstans tárigényű (folyt.)

- Nagy a csábítás, hogy azt higgyük, ez a változat már jó.

```
expt5(_B,0) -> 1;                    %% Kell ez a klóz!
expt5(B,N) when N>0 -> expt5(B,N,B).

expt5(_B,1,R) -> R;
expt5(B,N,R) ->
    case even(N) of
        true  -> expt5(B,N div 2,R*R);
        false -> expt5(B,N-1,R*B)
    end.
```

- Sajnos, ROSSZ ez is:

```
X0 = expt5(2,0), X1 = expt5(2,1), X2 = expt5(2,2),
X3 = expt5(2,3), X4 = expt5(2,4), X5 = expt5(2,5),
X6 = expt5(2,6), X7 = expt5(2,7), X8 = expt5(2,8),
X0,X1,X2,X3,X4,X5,X6,X7,X8. ??

{X0, X1, X2, X3, X4,  X5, X6,   X7,  X8} ==
{ 1,  2,  4, 16, 16, 256, 64, 1024, 256}.
```