Deklaratív Programozás

Szeredi Péter[1] Dóbé Péter[2]

[1] szeredi@cs.bme.hu
BME Számítástudományi és Információelméleti Tanszék

[2] dobe@iit.bme.hu
BME Irányítástechnika és Informatika Tanszék

2017 ősz

Az előadók köszönetüket fejezik ki Hanák Péternek, a tárgy alapítójának

I. rész

Bevezetés

1 Bevezetés
2 Cékla: deklaratív programozás C++-ban
3 Erlang alapok

A tárgy témája

- Deklaratív programozási nyelvek – gyakorlati megközelítésben
- Két fő irány:
  - funkcionális programozás **Erlang** nyelven
  - logikai programozás **Prolog** nyelven
- Bevezetésként foglalkozunk a C++ egy deklaratív résznyelvével, a Cékla nyelvvel – C(É) deKLAratív része
- A **két fő nyelv**ként az **Erlang** és **Prolog** nyelvekre hivatkozunk majd (lásd követelmények)

Tartalom

1 Bevezetés
- Követelmények, tudnivalók
- Egy kedvcsináló példa Prolog nyelven
- A példa Erlang változata

Honlap, ETS, levelezési lista

- Honlap: `http://dp.iit.bme.hu`
  a jelen félév honlapja: `http://dp.iit.bme.hu/dp-current`
- ETS, az Elektronikus TanárSegéd
  `http://dp.iit.bme.hu/ets`
- Levelezési lista:
  `http://www.iit.bme.hu/mailman/listinfo/dp-l`
- A listára automatikusan felvesszük a tárgy hallgatóit az ETS-beli címükkel. Címet módosítani csak az ETS-ben lehet.
- A listára levelet küldeni a `dp-l@iit.bme.hu` címre lehet.
- Csak a feliratkozási címről küldött levelek jutnak el moderátori jóváhagyás nélkül a listatagokhoz.

Prolog-jegyzet

- Szeredi Péter, Benkő Tamás: Deklaratív programozás. Bevezetés a logikai programozásba. Budapest, 2005
  - Elektronikus változata letölthető a honlapról (ps, pdf)
  - Nyomtatott változata kifogyott
  - Kellő számú további igény esetén megszervezzük az újranyomtatást
- A SICStus Prolog kézikönyve (angol):
  `http://www.sics.se/isl/sicstuswww/site/documentation.html`

Magyar nyelvű Prolog szakirodalom

- Farkas Zsuzsa, Futó Iván, Langer Tamás, Szeredi Péter:
  Az MProlog programozási nyelv.
  Műszaki Könyvkiadó, 1989
  jó bevezetés, sajnos az MProlog beépített eljárásai nem szabványosak.
- Márkusz Zsuzsa: Prologban programozni könnyű.
  Novotrade, 1988
  mint fent
- Futó Iván (szerk.): Mesterséges intelligencia. (9.2 fejezet, Szeredi Péter)
  Aula Kiadó, 1999
  csak egy rövid fejezet a Prologról
- Peter Flach: Logikai Programozás. Az intelligens következtetés példákon keresztül.
  Panem — John Wiley & Sons, 2001
  jó áttekintés, inkább elméleti érdeklődésű olvasók számára

Angol nyelvű Prolog szakirodalom

- Logic, Programming and Prolog, 2nd Ed., by Ulf Nilsson and Jan Maluszynski, Previously published by John Wiley & Sons Ltd. (1995)
  Letölthető a `http://www.ida.liu.se/~ulfni/lpp` címről.
- Prolog Programming for Artificial Intelligence, 3rd Ed., Ivan Bratko, Longman, Paperback - March 2000
- The Art of PROLOG: Advanced Programming Techniques, Leon Sterling, Ehud Shapiro, The MIT Press, Paperback - April 1994
- Programming in PROLOG: Using the ISO Standard, C.S. Mellish, W.F. Clocksin, Springer-Verlag Berlin, Paperback - July 2003

Erlang-szakirodalom (angolul)

- Joe Armstrong: Programming Erlang. Software for a Concurrent World. The Pragmatic Bookshelf, 2007. `http://www.pragprog.com/titles/jaerlang/programming-erlang`
- Joe Armstrong, Robert Virding, Claes Wikström, Mike Williams: Concurrent Programming in Erlang. Second Edition. Prentice Hall, 1996. Az első rész szabadon letölthető PDF-ben: `http://erlang.org/download/erlang-book-part1.pdf`

További irodalom:

- On-line Erlang documentation `http://erlang.org/doc.html` vagy `erl -man <module>`
- Learn You Some Erlang for great good! `http://learnyousomeerlang.com`
- ERLANG összefoglaló magyarul `http://nyelvek.inf.elte.hu/leirasok/Erlang/`
- Wikibooks on Erlang Programming `http://en.wikibooks.org/wiki/Erlang_Programming`
- Francesco Cesarini, Simon Thompson: Erlang Programming. O´Reilly, 2009. `http://oreilly.com/catalog/9780596518189/`

Fordító- és értelmezőprogramok

- SICStus Prolog – 4.3 verzió (licensz az ETS-en keresztül kérhető)
- Más Prolog rendszer is használható (pl. SWI Prolog `http://www.swi-prolog.org/`, Gnu Prolog `http://www.gprolog.org/`), de a házi feladatokat csak akkor fogadjuk el, ha azok a SICStus rendszerben (is) helyesen működnek.
- Erlang (szabad szoftver)
- Letöltési információ a honlapon (Linux, Windows)
- Webes Prolog gyakorló felület az ETS-ben (ld. honlap)
- Kézikönyvek HTML-, ill. PDF-változatban
- Emacs szövegszerkesztő Erlang-, ill. Prolog-módban (Linux, Win95/98/NT/XP/Vista/7)
- Eclipse fejlesztői környezet (SPIDER, erlIDE)

Deklaratív programozás: követelmények

Nagy házi feladat (NHF)

- Programozás mindkét fő nyelven (Prolog, Erlang)
- Mindenkinek önállóan kell kódolnia (programoznia)!
- Hatékony (időlimit!), jól dokumentált („kommentezett”) programok
- A két programhoz közös, 5–10 oldalas fejlesztői dokumentáció (TXT, HTML, PDF, PS; de nem DOC vagy RTF)
- Kiadás legkésőbb a 4. héten, a honlapon, letölthető keretprogrammal
- Beadás a 9. héten; elektronikus úton (ld. honlap)
- A beadáskor és a pontozáskor külön-külön tesztsorozatot használunk (nehézségben hasonlókat, de nem azonosakat)
- Azok a programok, amelyek megoldják a tesztesetek 80%-át *létraversenyen* vesznek részt (hatékonyság, gyorsaság plusz pontokért)

Deklaratív programozás: követelmények (folyt.)

Nagy házi feladat (folyt.)

- A beadási határidőig többször is beadható, csak az utolsót értékeljük
- Pontozása mindkét fő nyelvből:
  - helyes (azaz jó eredményt időkorláton belül adó) futás esetén a 10 teszteset mindegyikére 0,5-0,5 pont, összesen max. 5 pont
  - a dokumentációra, a kód olvashatóságára, kommentezettségére max. 2,5 pont
  - tehát nyelvenként összesen max. 7,5 pont szerezhető
- A NHF súlya az osztályzatban: 15% (a 100 pontból 15)
- A megajánlott jegy előfeltétele, hogy a hallgató nagy házi feladata mindkét fő nyelvből bejusson a létraversenybe (minimum 80%-os teljesítmény)
- A NHF beadása **nem kötelező, de ajánlott!**

Deklaratív programozás: követelmények (folyt.)

Kis házi feladatok (KHF)

- 3 feladat Prologból, 3 Erlangból, 1 Céklából
- Beadás elektronikus úton (ld. honlap)
- Egy KHF beadása érvényes, ha minden tesztesetre lefut
- **Kötelező** a KHF-ek legalább 50%-ának érvényes beadása, és legalább egy érvényes KHF beadása Prologból is és Erlangból is. Azaz kötelező 1 Prolog, 1 Erlang, és 1 bármilyen (összesen 3) KHF érvényes beadása.
- Minden feladat jó megoldásáért 1-1 jutalompont (azaz a 100 alappont feletti pont) jár
- Minden KHF-nak külön határideje van, pótlási lehetőség nincs
- A KHF-k egyre összetettebbek és egymásra épülnek– érdemes minél előbb elkezdeni a KHF-k beadását!
- A házi feladatot önállóan kell elkészíteni! Másolás esetén kötelesek vagyunk fegyelmi eljárást indítani: `http://www.kth.bme.hu/document/189/original/bme_rektori_utasitas_05.pdf` ("Beadandó feladat ... elkészíttetése mással")

Deklaratív programozás: követelmények (folyt.)

Gyakorlatok

- 2. héttől kezdődően 2 órás gyakorlatok, időpontok a Neptunban
- Laptop használata megengedett
- **Kötelező** részvétel a gyakorlatok 70 %-án (pontosabban n gyakorlat esetén legalább $\lfloor 0.7n \rfloor$ gyakorlaton)
- További Prolog gyakorlási lehetőség az ETS rendszerben (gyakorló feladatok, lásd honlap)

Deklaratív programozás: követelmények (folyt.)

Nagyzárthelyi, pótzárthelyi (NZH, PZH, PPZH)

- A zárthelyi **kötelező**, de van megajánlott jegy = zárthelyi alóli mentesség
  - Alapfeltételek: KHF, Gyakorlat teljesítése; NHF „megvédése”
  - Jó (4): a nagy házi feladat mindkét fő nyelvből bejut a létraversenybe
  - Jeles (5): legalább 40%-os eredmény a létraversenyen, mindkét fő nyelvből
- Semmilyen jegyzet, segédlet nem használható
- 40%-os szabály (nyelvenként a maximális részpontszám 40%-a kell az eredményességhez)
- NZH: 11. héten, később meghirdetendő időpontban
- A PPZH-ra a pótlási időszakban egyetlen alkalommal adunk lehetőséget
- Az NZH anyaga az addig előadott tananyag
- A PZH, ill. a PPZH anyaga azonos az NZH anyagával
- A zárthelyi súlya az osztályzatban: 85% (a 100 pontból 85)

Tartalom

1 Bevezetés
- Követelmények, tudnivalók
- Egy kedvcsináló példa Prolog nyelven
- A példa Erlang változata

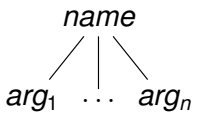
Bevezető példa: adott értékű kifejezés előállítása

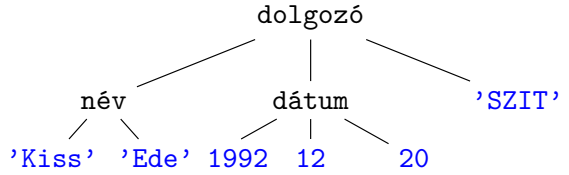
- A feladat: írjunk Prolog programot a következő feladvány megoldására:
  - Adott számokból a négy alapművelet (+, -, *, /) segítségével építsünk egy megadott értékű kifejezést!
  - A számok nem „tapaszthatók” össze hosszabb számokká
  - Mindegyik adott számot pontosan egyszer kell felhasználni, sorrendjük tetszőleges lehet
  - Nem minden alapműveletet kell felhasználni, egyfajta alapművelet többször is előfordulhat
  - Zárójelek tetszőlegesen használhatók
- Példák a fenti szabályoknak megfelelő, az 1, 3, 4, 6 számokból felépített kifejezésekre: $1 + 6 * (3 + 4)$, $(1 + 3)/4 + 6$
- Viszonylag nehéz megtalálni egy olyan kifejezést, amely az 1, 3, 4, 6 számokból áll, és értéke 24

A Prolog nyelv adatfogalma

A Prolog adatokat Prolog kifejezésnek hívjuk (angolul: term). Fajtái:

- egyszerű kifejezés: számkonstans (pl. 3), névkonstans (pl. `alma`, `'SZIT'`), vagy változó (pl. `X`)
- összetett kifejezés (rekord, struktúra): *name*$(arg_1, \ldots, arg_n)$
  - *name* egy névkonstans, az arg_i mezők tetsz. Prolog kifejezések
  - példa: `dolgozó(név('Kiss','Ede'),dátum(1992,12,20),'SZIT').`
  - Az összetett kifejezések valójában fastruktúrát alkotnak:

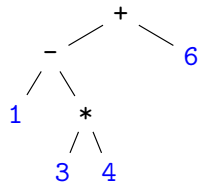




Szintaktikus „édesítőszerek” Prologban

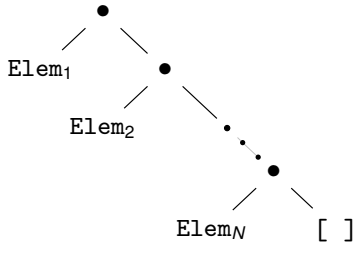
- Egy- és kétargumentumú struktúrák operátoros (infix. prefix stb.) írásmódja: `1+2`≡`+(1,2)`

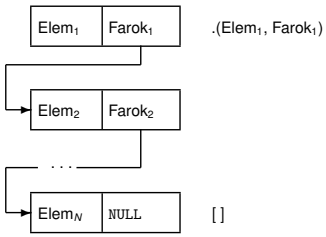
```
| ?- write_canonical(1-3*4+6).
+(-(1,*(3,4)),6)
```



- Listák, mint speciális struktúrák

```
| ?- write_canonical([a,b,c]).
'.'(a,'.'(b,'.'(c,[])))
```





Aritmetikai kifejezések kezelése Prologban – ellenőrzés

Írjunk egy `kif` nevű egyargumentumú Prolog eljárást!
A `kif(X)` hívás sikeresen fut le, ha `X` egy olyan kifejezés, amely számokból a négy alapművelet (+, -, *, /) segítségével épül fel (röviden, ha `X` **helyes**).

- Az alábbi sorokat helyezzük el pl. a `kif0.pl` file-ban:

```
% kif(K): K számokból a négy alapművelettel képzett helyes kifejezés.
kif(K) :-   number(K).      % K helyes, ha K szám. (number beépített elj.)
kif(X+Y) :- kif(X), kif(Y). % X+Y helyes, ha X helyes és Y helyes
kif(X-Y) :- kif(X), kif(Y).
kif(X*Y) :- kif(X), kif(Y).
kif(X/Y) :- kif(X), kif(Y).
```

- Betöltése: `| ?- compile(kif0).` vagy `| ?- consult(kif0).`
- Futtatás nyomkövetés nélkül és nyomkövetéssel (`consult`-ot követően):

```
| ?- kif(alma).
no
| ?- kif(1+2).
yes
| ?-
```

```
| ?- trace, kif(alma).
        1      1 Call: kif(alma) ?
        2      2 Call: number(alma) ?
        2      2 Fail: number(alma) ?
        1      1 Fail: kif(alma) ?
no
| ?-
```

Aritmetikai kifejezések ellenőrzése – továbbfejlesztett változat

- A `kif` Prolog eljárás segédeljárást használó változata:

```
% kif2(K): K számokból, a négy alapművelettel képzett kifejezés.
kif2(Kif) :-
        number(Kif).
kif2(Kif) :-
        alap4(X, Y, Kif),
        kif2(X), kif2(Y).
```

- Az `alap4` segédeljárás:

```
% alap4(X, Y, Kif): A Kif kifejezés az X és Y kifejezésekből
%                   a négy alapművelet egyikével áll elő.
alap4(X, Y, X+Y).                alap4(X, Y, X-Y).
alap4(X, Y, X*Y).                alap4(X, Y, X/Y).
```

- Ekvivalens, ún. diszjunkciót használó változat („;” ≡ „vagy”):

```
alap4(X, Y, Kif) :- (   Kif = X+Y   ;   Kif = X-Y
                    ;   Kif = X*Y   ;   Kif = X/Y
                    ).
```

`A=B` egy infix alakban írható beépített eljárás, jelentése:
`A` és `B` azonos alakra hozható, esetleges változóbehelyettesítésekkel.

Aritmetikai kifejezés levéllistájának előállítása

- A `kif_levelek` eljárás ellenőrzi a kifejezést és előállítja levéllistáját

```
% kif_levelek(Kif, L): A számokból alapműveletekkel felépülő Kif
%                       kifejezés leveleiben levő számok listája L.
kif_levelek(Kif, L) :-
        number(Kif), L = [Kif].  % L egyelemű, Kif-ből álló lista
kif_levelek(Kif, L) :-
        alap4(K1, K2, Kif),
        kif_levelek(K1, LX),
        kif_levelek(K2, LY),
        append(LX, LY, L).

| ?- kif_levelek(2/3-4*(5+6), L).     ⟶ L = [2,3,4,5,6]
```

- Az `append` egy beépített eljárás, fejkommentje és példafutása

```
% append(L1, L2, L3): Az L1 és L2 listák összefűzése az L3 lista.

| ?- append([1,2], [3,4], L).        ⟶ L = [1,2,3,4]
```


Az `append` eljárás többirányú használata

- Az `append` eljárás a fejkommentje által leírt *relációt* valósítja meg, sokféle módon használható, és több választ is adhat (új válasz kérése `;`-vel)

```
% append(L1, L2, L3): Az L1 és L2 listák összefűzése az L3 lista.

| ?- append(L, [3], [1,2,3]).   % [1,2,3] utolsó eleme-e 3,
L = [1,2] ? ;                   % és milyen L lista van előtte?
no                              % nincs TÖBB válasz
| ?- append([1,2], L, [1,2,3]). % [1,2,3,4] prefixuma-e [1,2]?
L = [3] ? ;  no
| ?- append(L1, L2, [1,2,3]).   % [1,2,3] hogyan bontható két részre?
L1 = [], L2 = [1,2,3] ? ;
L1 = [1], L2 = [2,3] ? ;
L1 = [1,2], L2 = [3] ? ;
L1 = [1,2,3], L2 = [] ? ; no
| ?- append(L, [2], L2).
L = [], L2 = [2] ? ;
L = [_A], L2 = [_A,2] ? ;
L = [_A,_B], L2 = [_A,_B,2] ? ; % végtelen sok válasz, problémás ...
...
```


Adott levéllistájú aritmetikai kifejezések előállítása

- A `kif_levelek` eljárás sajnos nem használható „visszafelé”, végtelen ciklusba esik, lásd pl. `| ?- kif_levelek(Kif, [1]).`
- Ez javítható a hívások átrendezésével és új feltételek beszúrásával:

```
% kif_levelek(+Kif, -L):
% Kif levéllistája L.
kif_levelek(Kif, L) :-
        number(Kif),
        L = [Kif].
kif_levelek(Kif, L) :-
        alap4(K1, K2, Kif),

        kif_levelek(K1, L1),
        kif_levelek(K2, L2),
        append(L1, L2, L).
```

```
% levelek_kif(+L, -Kif):
% Kif levéllistája L.
levelek_kif(L, Kif) :-
        L = [Kif],
        number(Kif).
levelek_kif(L, Kif) :-
        append(L1, L2, L),
        L1 \= [], L2 \= [],
        % L1, L2 nem-üres listák
        levelek_kif(L1, K1),
        levelek_kif(L2, K2),
        alap4(K1, K2, Kif).
```

```
| ?- levelek_kif([1,3,4], K).
K = 1+(3+4) ? ; K = 1-(3+4) ? ; K = 1*(3+4) ? ; K = 1/(3+4) ? ;
K = 1+(3-4) ? ; K = 1-(3-4) ? ; K = 1*(3-4) ? ; K = 1/(3-4) ? ; ...
```

Adott értékű kifejezés előállítása

- Bevezető példánk megoldásához szükséges további nyelvi elemek
  - A `lists` könyvtárban található `permutation` eljárás:

    ```
    % permutation(L, PL): PL az L lista permutációja.
    ```

  - Az `=:=` (`=\=`) beépített aritmetikai eljárás mindkét argumentumában aritmetikai kifejezést vár, azokat kiértékeli, és csakkor sikerül, ha az értékek aritmetikailag megegyeznek (különböznek), pl.

    ```
    | ?- 4+2 =\= 3*2. ⟶ no          | ?-  2.0 =:= 2. ⟶ yes
    | ?- 8/3 =:= 2.666666666666666. ⟶ no
    ```

- A példa „generál és ellenőriz” (generate-and-test) stílusú megoldása:

```
% levelek_ertek_kif(L, Ertek, Kif): Kif az L listabeli számokból
% a négy alapművelet segítségével felépített olyan kifejezés,
% amelynek értéke Ertek.
levelek_ertek_kif(L, Ertek, Kif) :-
        permutation(L, PL), levelek_kif(PL, Kif), Kif =:= Ertek.

| ?- levelek_ertek_kif([1,3,4], 11, Kif).
Kif = 3*4-1 ? ; Kif = 4*3-1 ? ; no
```


Adott értékű kifejezés előállítása – a teljes kód

```
:- use_module(library(lists), [permutation/2]). % importálás

% levelek_ertek_kif(L, Ertek, Kif): Kif az L listabeli számokból
% a négy alapművelettel felépített, Ertek értékű kifejezés.
levelek_ertek_kif(L, Ertek, Kif) :-
        permutation(L, PL), levelek_kif(PL, Kif), Kif =:= Ertek.

% levelek_kif(L, Kif): Az alapműveletekkel felépített Kif levéllistája L.
levelek_kif(L, Kif) :-
        L = [Kif], number(Kif).
levelek_kif(L, Kif) :-
        append(L1, L2, L),
        L1 \= [], L2 \= [], levelek_kif(L1, K1), levelek_kif(L2, K2),
        alap4_0(K1, K2, Kif).

% alap4_0(X, Y, K): K X-ből és Y-ból értelmes alapművelettel áll elő.
alap4_0(X, Y, X+Y).
alap4_0(X, Y, X-Y).
alap4_0(X, Y, X*Y).
alap4_0(X, Y, X/Y) :- Y =\= 0.      % a 0-val való osztás kiküszöbölése
```


Tartalom

1 Bevezetés
- Követelmények, tudnivalók
- Egy kedvcsináló példa Prolog nyelven
- A példa Erlang változata

Erlang-kifejezések

- Erlang: nem logikai, hanem *funkcionális* programnyelv
- Összetett Erlang-kifejezéseket, függvényhívásokat értékelünk ki:

```
1> [1-3*4+6, 1-3/4+6].
[-5,6.25]
2> lists:seq(1,3).
[1,2,3]
3> {1/2, '+', 1+1}.
{0.5,'+',2}
```

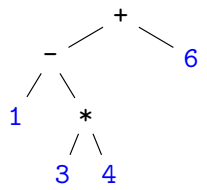
- *Hármas:* $\{K_1, K_2, K_3\}$, ahol K_i tetszőleges Erlang-kifejezés. *Pár:* $\{K_1,K_2\}$.
- A *listanézet* Erlang-kifejezés a matematikai halmaznézet imitációja:

```
4> [X || X <- [1,2,3] ].          %  {x|x ∈ {1,2,3}}
[1,2,3]
5> [X || X <- [1,2,3], X*X > 5].  %  {x|x ∈ {1,2,3}, x^2 > 5}
[3]
6> [{X,Y} || X <- [1,2,3], Y <- lists:seq(1,X)].
                                  % {(x,y)|x ∈ {1,2,3}, y ∈ {1 .. x}}
[{1,1},{2,1},{2,2},{3,1},{3,2},{3,3}]
```

Aritmetikai kifejezések ábrázolása

- Primitívebb a Prolognál: nem tudja automatikusan sem ábrázolni, se felsorolni az aritmetikai kifejezéseket
- Prolog egy aritmetikai kifejezést fában ábrázol:

```
| ?- write_canonical(1-3*4+6).
+(-(1,*(3,4)),6)
yes
```



- Erlangban explicit módon fel kell sorolnunk az összes ilyen fát, és explicit módon ki kell őket értékelni
- A példaprogramunkban a fenti aritmetikai kifejezést (önkényesen) egymásba ágyazott hármasokkal ábrázoljuk:

```
{{1, '-', {3, '*', 4}}, '+', 6}
```


Adott méretű fák felsorolása

- Fa-elrendezések felsorolása például csupa 1-esekből és '+' műveletekből
- Összesen 5 db. 4 levelű fa van:

```
{1,'+',{1,'+',{1,'+',1}}}
{1,'+',{{1,'+',1},'+',1}}
{{1,'+',1},'+',{1,'+',1}}
{{1,'+',{1,'+',1}},'+',1}
{{{1,'+',1},'+',1},'+',1}
```

Erlang-kód:

```
% @type fa() = 1 | {fa(),'+',fa()}.
% fak(N) = az összes N levelű fa listája.
fak(1) ->
    [1];
fak(N) ->
    [ {BalFa,'+',JobbFa}
      || I <- lists:seq(1, N-1),
         BalFa <- fak(I),
         JobbFa <- fak(N-I)    ].
```

Matematikai nézet

Fa definíciója.

- 1 levelet tartalmazó fák halmaza: $\{1\}$
- n levelet tartalmazók: $\{ (b, \text{'+'}, j) \mid i \in [1 \,..\, n-1], b \in \text{fak}(i), j \in \text{fak}(n-i) \}$

Adott levéllistájú aritmetikai kifejezések felsorolása

- Segédfv: egy lista összes lehetséges kettévágása nem üres listákra

```
1> kif:kettevagasok([1,3,4,6]).
[ {[1],[3,4,6]}, {[1,3],[4,6]}, {[1,3,4],[6]} ]
```

- Kifejezések adott számokból *adott sorrendben*, 4 alapműveletből:

Erlang-kód:

```
% @type kif() = {kif(),muvelet(),kif()}
%              | integer().
% @type muvelet() = '+' | '-' | '*' | '/'.
% kif(L) = L levéllistájú kifek listája.
kifek([H]) ->
  [H];
kifek(L) ->
  [ {B,M,J}
    || {LB,LJ} <- kettevagasok(L),
       B <- kifek(LB),
       J <- kifek(LJ),
       M <- ['+', '-', '*', '/']
  ].
```

Matematikai nézet:

Kifejezés (kif) definíciója. (Az előző általánosítása.)

- Egyetlen h levelet tartalmazó kifek: $\{h\}$
- L levéllistájú kifek: $\{(b, m, j) \mid L_B \oplus L_J = L, b \in \text{kifek}(L_B), j \in \text{kifek}(L_J), m \in \{+, -, *, /\}\}$

Utolsó lépés: a kifejezések explicit kiértékelése

```
% ertek(K) = a K kifejezés számértéke.
ertek({B,Muvelet,J}) ->
    erlang:Muvelet(ertek(B), ertek(J));
ertek(I) ->
    I.
```

- Példák:

```
1> erlang:'+'(1,3).
4
2> kif:ertek(3).
3
3> kif:ertek({{1,'-',{3,'*',4}},'+',6}).
-5
4> kif:ertek({1,'/',0}).
** exception error: ...
```

```
% permutaciok(L) = az L lista elemeinek minden permutációja.
5> kif:permutaciok([1,3,4]).
[[1,3,4], [1,4,3], [3,1,4], [3,4,1], [4,1,3], [4,3,1]]
```

Adott értékű kifejezések felsorolása – teljes kód

`kif:megoldasok([1,3,4,6], 24).`

```
-module(kif).
-compile([export_all]).

megoldasok(Szamok, Eredmeny) ->
    [Kif || L <- permutaciok(Szamok),
            Kif <- kifek(L),
            (catch ertek(Kif)) == Eredmeny].
```

- `catch`: 0-val való osztásnál keletkező kivétel miatt

```
kifek([H]) -> [H];
kifek(L) -> [ {B,M,J} || {LB,LJ} <- kettevagasok(L),
                         B <- kifek(LB),
                         J <- kifek(LJ),
                         M <- ['+', '-', '*', '/']  ].
ertek({B,M,J}) -> erlang:M(ertek(B), ertek(J));
ertek(I) -> I.
kettevagasok(L) -> [ {LB,LJ} || I <- lists:seq(1, length(L)-1),
                                {LB,LJ} <- [lists:split(I, L)]  ].
permutaciok([]) -> [[]];
permutaciok(L) -> [[H|T] || H <- L, T <- permutaciok(L--[H])].
```


II. rész

Cékla: deklaratív programozás C++-ban

1 Bevezetés
2 Cékla: deklaratív programozás C++-ban
3 Erlang alapok

Tartalom

2 Cékla: deklaratív programozás C++-ban
- Néhány deklaratív paradigma C nyelven
- Jobbrekurzió
- A Cékla programozási nyelv
- Listakezelés Céklában
- Magasabb rendű függvények

A deklaratív programozás jelmondata

- inkább MIT, kevésbé HOGYAN (WHAT rather than HOW):
  a *megoldás módja* helyett inkább
  a megoldandó *feladat leírását* kell megadni
- A gyakorlatban mindkét szemponttal foglalkozni kell

  Kettős szemantika:
  - deklaratív szemantika
    MIT (milyen feladatot) old meg a program;
  - procedurális szemantika
    HOGYAN oldja meg a program a feladatot.
- Új gondolkodási stílus, dekomponálható, verifikálható programok
- Új, magas szintű programozási elemek
  - rekurzió (algoritmus, adatstruktúra)
  - mintaillesztés
  - visszalépéses keresés
  - magasabb rendű függvények

Imperatív és deklaratív programozási nyelvek

- Imperatív program
  - felszólító módú, utasításokból áll
  - változó: változtatható értékű memóriahely
  - C nyelvű példa:

```
int pow(int A, int N) {  // pow(A,N) = A^N
  int P = 1;             // Legyen P értéke 1!
  while (N > 0) {        // Amíg N>0 ismételd ezt:
    N = N-1;             //    Csökkentsd N-et 1-gyel!
    P = P*A;    }        //    Szorozd P-t A-val!
  return P;           }  // Add vissza P végértékét
```

- Deklaratív program
  - kijelentő módú, egyenletekből, állításokból áll
  - változó: egyetlen, fix, a programírás idején ismeretlen értékkel bír
  - Erlang példa:

```
pow(A,N) -> if                      % Elágazás
            N==0 -> 1;              % Ha N == 0, akkor 1
            N>0  -> A * pow(A, N-1) % Ha N>0, akkor A*A^(N-1)
            end.                    % Elágazás vége
```


Deklaratív programozás imperatív nyelven

Lehet pl. C-ben is deklaratívan programozni

ha nem használunk: értékadó utasítást, ciklust, ugrást stb.,
amit használhatunk: csak konstans változók, (rekurzív) függvények, if-then-else

- Példa (a `pow` függvény deklaratív változata a `powd`):

```
// powd(A,N) = A^N
int powd(const int A, const int N) {
  if (N > 0)                    // Ha N > 0
    return A * powd(A,N-1);     // akkor A^N = A*A^(N-1)
  else
    return 1;                   // egyébként A^N = 1
}
```

- A (fenti típusú) rekurzió költséges, nem valósítható meg konstans tárigénnyel :-(

`powd(10,3) : 10*powd(10,2) : 10*(10*powd(10,1)) :` $\underbrace{10*(10*(10*1))}_{\text{tárolni kell}}$

Tartalom

2 Cékla: deklaratív programozás C++-ban
- Néhány deklaratív paradigma C nyelven
- Jobbrekurzió
- A Cékla programozási nyelv
- Listakezelés Céklában
- Magasabb rendű függvények

Hatékony deklaratív programozás

- A rekurziónak van egy hatékonyan megvalósítható változata
- Példa: döntsük el, hogy egy `A` szám előáll-e egy `B` szám hatványaként:

```
/* ispow(A,B) = létezik i, melyre B^i = A.
 * Előfeltétel: A > 0, B > 1 */
```

```
int ispow(int A, int B) {

  if (A == 1) return true;
  if (A%B==0) return ispow(A/B, B);
              return false;
}
```

```
int ispow(int A, int B) {
again:
  if (A == 1) return true;
  if (A%B==0) {A=A/B; goto again;}
              return false;
}
```

- Itt a színezett rekurzív hívás átírható iteratív kódra: értékadással és ugrással helyettesíthető!
- Ez azért tehető meg, mert a rekurzióból való visszatérés után *azonnal* kilépünk az adott függvényhívásból.
- Az ilyen függvényhívást **jobbrekurzió**nak vagy **terminális rekurzió**nak vagy **farok rekurzió**nak nevezzük („*tail recursion*”)
- A Gnu C fordító (GCC) megfelelő optimalizálási szint mellett a rekurzív definícióból is a nem-rekurzív (jobboldali) kóddal azonos kódot generál!

Jobbrekurzív függvények

- Lehet-e jobbrekurzív kódot írni a hatványozási (`pow(A,N)`) feladatra?
  - A gond az, hogy a rekurzióból „kifelé jövet” már nem csinálhatunk semmit
  - Tehát a végeredménynek az utolsó hívás belsejében elő kell állnia!
  - A megoldás: segédfüggvény definiálása, amelyben egy vagy több ún. gyűjtőargumentumot (*akkumulátort*) helyezünk el.
- A `pow(A,N)` jobbrekurzív (iteratív) megvalósítása:

```
// Segédfüggvény: powi(A, N, P) = P*A^N
int powi(const int A, const int N, const int P) {
  if (N > 0)
    return powi(A, N-1, P*A);
  else
    return P;
}

int powi(const int A, const int N){
  return powi(A, N, 1);
}
```


Imperatív program átírása jobbrekurzív, deklaratív programmá

- Minden ciklusnak egy segédfüggvényt feleltetünk meg (Az alábbi példában: while ciklus $\Longrightarrow$ `powi(A,N,P)` függvény)
- A segédfüggvény argumentumai a ciklus által tartalmazott változóknak felelnek meg (`powi` argumentumai az `A`, `N`, `P` értékek)
- A segédfüggvény eredménye a ciklus által az őt követő kódnak továbbadott változó(k) értéke (`powi` eredménye a `P` végértéke)
- Példa: a hatványszámító program

```
int pow(int A, int N) {
  int P = 1;


  while (N > 0) {
    N = N-1;
    P = P*A;    }

  return P;
}
```

```
int powi(int A, int N)  {
  return powi(A, N, 1); }

int powi(int A, int N, int P) {
  if (N > 0) return powi(A,
                         N-1,
                         P*A);
  else
    return P;
}
```


Példák: jobbrekurzióra átírt rekurziók

Általános rekurzió	Jobbrekurzió

`// fact(N) = N!`
`int fact(const int N) {`
`  if (N==0) return 1;`
`  return N * fact(N-1);`
`}`
	`// facti(N, I) = N! * I`
	`int facti(const int N, const int I) {`
	`  if (N==0) return I;`
	`  return facti(N-1, I*N);`
	`}`
	`int facti(const int N) {`
	`  return facti(N, 1); }`

`// fib(N) =`
`// N. Fibonacci szám`
`int fib(const int N) {`
`  if (N<2) return N;`
`  return fib(N-1) +`
`  fib(N-2);`
`}`
	`int fibi(const int N,  // Segédfv`
	`  const int Prev, const int Cur)`
	`{`
	`  if (N==0) return 0;`
	`  if (N==1) return Cur;`
	`  return fibi(N-1, Cur, Prev + Cur);`
	`}`
	`int fibi(const int N) {`
	`  return fibi(N, 0, 1); }`

Tartalom

2 Cékla: deklaratív programozás C++-ban
- Néhány deklaratív paradigma C nyelven
- Jobbrekurzió
- A Cékla programozási nyelv
- Listakezelés Céklában
- Magasabb rendű függvények

Cékla 2: A „CÉ++” nyelv egy deKLAratív része

- Megszorítások:
  - Típusok: csak `int`, lista vagy függvény (lásd később)
  - Utasítások: `if-then-else`, `return`, blokk, kifejezés
  - Változók: csak egyszer, deklarálásukkor kaphatnak értéket (`const`)
  - Kifejezések: változókból és konstansokból kétargumentumú operátorokkal, függvényhívásokkal és feltételes szerkezetekkel épülnek fel
    - ⟨ aritmetikai-op ⟩: `+ | - | * | / | % |`
    - ⟨ hasonlító-op ⟩: `< | > | == | != | >= | <=`
- C++ fordítóval is fordítható a `cekla.h` fájl birtokában: láncolt lista kezelése, függvénytípusok és kiírás
- Kiíró függvények: főleg nyomkövetéshez, ugyanis *mellékhatásuk* van!
  - `write(X);` Az `X` kifejezés kiírása a standard kimenetre
  - `writeln(X);` Az `X` kifejezés kiírása és soremelés
- A (Prologban írt) Cékla fordító és a `cekla.h` letölthető a tárgy honlapjáról

Cékla Hello world!

`hello.cpp`

```
#include "cekla.h"          // így C++ fordítóval is fordítható
int main() {                // bárhogy nevezhetnénk a függvényt
  writeln("Hello World!");  // nem-deklaratív utasítás
}                           // C++ komment megengedett
```

- Fordítás és futtatás a `cekla` programmal:

```
$ cekla hello.cpp                    Cékla parancssori indítása
Welcome to Cekla 2.238: a compiler for a declarative C++ sublanguage
* Function ‘main' compiled
* Code produced
To get help, type:      |* help;
|* main()                           Kiértékelendő kifejezés
Hello World!                        a mellékhatás
|* ^D                               end-of-file (Ctrl+D v Ctrl+Z)
Bye
$ g++ hello.cpp && ./a.out          Szabályos C++ program is
Hello World!
```


A Cékla nyelv szintaxisa

- A szintaxist BNF jelöléssel adjuk meg, kiterjesztés:
  - ismétlés (0, 1, vagy többszöri): `«ismétlendő»...`
  - zárójelezés: `[ ... ]`
  - `< >` jelentése: semmi
- A program szintaxisa

```
<program> ::=              <preprocessor_directive>...
                           <function_definition>...
<function_definition> ::=  <head> <block>
<head> ::=                 <type> <identifier>(<formal_args>)
<type> ::=                 [const | < >] [int | list | fun1 | fun2]
<formal_args> ::=          <formal_arg>[, <formal_arg>]... | < >
<formal_arg> ::=           <type> <identifier>
<block> ::=                { [<declaration> | <statement>]... }
<declaration> ::=          <type> <declaration_elem>
                           [, <declaration_elem>]... ;
<declaration_elem> ::=     <identifier> = <expression>
```


Cékla szintaxis folytatás: utasítások, kifejezések

```
<statement> ::=      if (<expression>) <statement> <else_part>
                   | <block>
                   | <expression> ;
                   | return <expression> ;
                   | ;
<else_part> ::=      else <statement> | < >

<expression> ::=     <expression_3> [? <expression> : <expression> | < >]
<expression_3> ::=   <expression_2> [<comp_op> <expression_2>]...
<expression_2> ::=   <expression_1> [<add_op> <expression_1>]...
<expression_1> ::=   <expression_0> [<mul_op> <expression_0>]...
<expression_0> ::=   <identifier>
                   | <constant>
                   | <identifier>(<actual_args>)
                   | (<expression>)
<constant> ::=       <integer> | <string> | '<char>'
<actual_args> ::=    <expression> [, <expression>]... | < >
<comp_op> ::=        < | > | == | != | >= | <=
<add_op> ::=         + | -
<mul_op> ::=         * | / | %
```

Tartalom

2 Cékla: deklaratív programozás C++-ban
- Néhány deklaratív paradigma C nyelven
- Jobbrekurzió
- A Cékla programozási nyelv
- Listakezelés Céklában
- Magasabb rendű függvények

Lista építése

- Egészeket tároló láncolt lista
- Üres lista: `nil` (globális konstans)
- Lista építése:

```
// Visszaad egy új listát: első eleme Head, farka a Tail lista.
list cons(int Head, list Tail);
```

`pelda.cpp` – példaprogram

```
#include "cekla.h"            // így szabályos C++ program is
int main() {                  // szabályos függvénydeklaráció
    const list L1 = nil;                    // üres lista
    const list L2 = cons(30, nil);          // [30]
    const list L3 = cons(10, cons(20, L2)); // [10,20,30]
    writeln(L1);              // kimenet: []
    writeln(L2);              // kimenet: [30]
    writeln(L3);              // kimenet: [10,20,30]
}
```


Futtatás Céklával

```
$ cekla
Welcome to Cekla 2.xxx: a compiler for a declarative C++ sublanguage
To get help, type:      |* help;
|* load "pelda.cpp";
* Function 'main' compiled
* Code produced
|* main();
[]
[30]
[10,20,30]
|* cons(10,cons(20,cons(30,nil)));
[10,20,30]
|* ^D
Bye
$
```


Lista szétbontása

- Első elem lekérdezése:

```
int  hd(list L)     // Visszaadja a nem üres L lista fejét.
```

- Többi elem lekérdezése:

```
list tl(list L)     // Visszaadja a nem üres L lista farkát.
```

- Egyéb operátorok: `=` (inicializálás), `==`, `!=` (összehasonlítás)
- Példa:

```
int sum(const list L) {         // az L lista elemeinek összege
    if (L == nil) return 0;     // ha L üres, akkor 0,
    else {                      // különben hd(L) + sum(tl(L))
        const int X = hd(L);    // segédváltozókat használhatunk,
        const list T = tl(L);   // de csak konstansokat
        return X + sum(T);      // rekurzió (ez nem jobbrekurzió!)
    }       //  Fejtörő: csak akkor lehet jobbrekurzióvá alakítani, ha
}           //  a T objektumot nem kell felszabadítani (destruktor)
int main() {
    const int X = sum(cons(10,cons(20,nil))); // sum([10,20]) == 30
    writeln(X);                 // mellékhatás: kiírjuk a 30-at
}
```

Sztringek Céklában

- Sztring nem önálló típus: karakterkódok listája, „szintaktikus édesítőszer”
- A lista a C nyelvből ismert „lezáró nullát” (`'\0'`) nem tárolja!
- `write` heurisztikája: ha a lista csak nyomtatható karakterek kódját tartalmazza (32..126), sztring formában íródik ki:

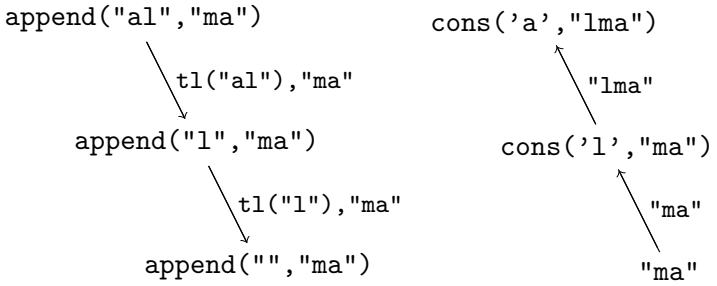
```
int main() {
  const list L4 = "abc";                    // abc
  const list L5 = cons(97,cons(98,cons(99,nil)));  // abc
  writeln(L4 == L5);                        // 1
  writeln(nil == "");                       // 1, true int-értéke
  writeln(nil);                             // []
  writeln(L5);                              // abc
  writeln(cons(10, L5));                    // [10,97,98,99]
  writeln(tl(L4));                          // bc
}
```


Listák összefűzése: append

- `append(L1, L2)` visszaadja L1 és L2 elemeit egymás után fűzve

```
// append(L1, L2) = L1 ⊕ L2 (L1 és L2 összefűzése)
list append(const list L1, const list L2)  {
  if (L1 == nil) return L2;
  return cons(hd(L1), append(tl(L1), L2)); }
```

- Például `append("al", "ma") == "alma"` (vagyis `[97,108,109,97]`).



- $O(n)$ lépésszámú (`L1` hossza), ha a lista átadása, `cons`, `hd`, `tl` $O(1)$
- Megjegyzés: a fenti megvalósítás nem jobbrekurzív

Lista megfordítása: nrev, reverse

- Naív (négyzetes lépésszámú) megoldás

```
// nrev(L) = az L lista megfordítva
list nrev(const list L) {
  if (L == nil) return nil;
  return append(nrev(tl(L)), cons(hd(L), nil));
}
```

- Lineáris lépésszámú megoldás

```
// reverse(L) = az L lista megfordítva
list reverse(const list L) {
  return revapp(L, nil);
}
// revapp(L, L0) = az L lista megfordítása L0 elé fűzve
list revapp(const list L, const list L0) {
  if (L == nil) return L0;
  return revapp(tl(L), cons(hd(L), L0));
}
```

- Egy jobbrekurzív `appendi(L1, L2)`: `revapp(revapp(L1,nil), L2)`

További listakezelő függvények

- Elem keresése listában

```
// ismember(X, L) = 1, ha az X érték eleme az L listának
int ismember(const int X, const list L) {
  if (L == nil) return false;
  if (hd(L) == X) return true;
  return ismember(X, tl(L));            }
```

- Döntsük el, hogy egy számlista csupa negatív számból áll-e!

```
// allneg(L) = 1, ha az L lista minden eleme negatív.
int allneg(const list L)        {
  if (L == nil) return true;
  if (hd(L) >= 0) return false;
  return allneg(tl(L));         }
```

- Állítsuk elő egy számlista negatív elemeiből álló listát!

```
// filterneg(L) = Az L lista negatív elemeinek listája.
list filterneg(const list L)                    {
  if (L == nil) return nil;
  const int X = hd(L); const list TL = tl(L);
  if (X >= 0) return filterneg(TL);
  else return cons(X, filterneg(TL));           }
```

Tartalom

2 Cékla: deklaratív programozás C++-ban
- Néhány deklaratív paradigma C nyelven
- Jobbrekurzió
- A Cékla programozási nyelv
- Listakezelés Céklában
- Magasabb rendű függvények

Magasabb rendű függvények Céklában

- Magasabb rendű függvény: paramétere vagy eredménye függvény
- A Cékla két függvénytípust támogat:

```
typedef int(* fun1 )(int)       // Egy paraméteres egész fv
typedef int(* fun2 )(int, int)  // Két paraméteres egész fv
```

- Példa: ellenőrizzük, hogy egy lista számjegykarakterek listája-e

```
// Igaz, ha L minden X elemére teljesül a P(X) predikátum
int for_all(const fun1 P, const list L) {
  if (L == nil) return true;          // triviális
  else {
    if (P(hd(L)) == false) return false;   // ellenpélda?
    return for_all(P, tl(L));         // többire is teljesül?
  }
}
int digit(const int X)   {  // Igaz, ha X egy számjegy kódja
  if (X < '0') return false;           // 48 == '0'
  if (X > '9') return false;           // 57 == '9'
  return true;           }
int szamjegyek(const list L) { return for_all(digit, L);  }
```


Gyakori magasabb rendű függvények: `map`, `filter`

- `map(F,L)`: az `F(X)` elemekből álló lista, ahol `X` végigfutja az `L` lista elemeit

```
list map(const fun1 F, const list L) {
    if (L == nil) return nil;
    return cons(F(hd(L)), map(F, tl(L)));
}
```

- Például az `L=[10,20,30]` lista elemeit eggyel növelve: `[11,21,31]`

```
int incr(const int X) { return X+1; }
```

Így a `map(incr, L)` kifejezés értéke `[11,21,31]`.

- `filter(P,L)`: az `L` lista azon `X` elemei, amelyekre `P(X)` teljesül

```
list filter(const fun1 P, const list L) {
    if (L == nil) return nil;
    if (P(hd(L))) return cons(hd(L), filter(P, tl(L)));
    else return filter(P, tl(L));
}
```

- Például keressük meg a "X=100;" sztringben a számjegyeket:
A `filter(digit, "X=100;")` kifejezés értéke `"100"` (azaz `[49,48,48]`)

Gyakori magasabb rendű függvények: `foldl`

- Hajtogatás balról

```
// foldl(F, a, [x1,...,xn]) = F(xn, ..., F(x2, F(x1, a))...)
int foldl(const fun2 F, const int Acc, const list L) {
  if (L == nil) return Acc;
  else
    return foldl(F, F(hd(L),Acc), tl(L));           }
```

- Futási példák, `L = [1,5,3,8]`

```
int xmy(int X, int Y) { return X-Y; }
int ymx(int X, int Y) { return Y-X; }

foldl(xmy, 0, L) = (8-(3-(5-(1-0)))) = 9
foldl(ymx, 0, L) = ((((0-1)-5)-3)-8) = -17
```

Gyakori magasabb rendű függvények: `foldr`

- Hajtogatás jobbról

```
// foldr(F, a, [x1, ..., xn]) = F(x1, F(x2, ..., F(xn, a)...))
int foldr(const fun2 F, const int Acc, const list L) {
  if (L == nil) return Acc;
  else
    return F(hd(L), foldr(F, Acc, tl(L)));          }
```

- Futási példák, `L = [1,5,3,8]`

```
int xmy(int X, int Y) { return X-Y; }
int ymx(int X, int Y) { return Y-X; }

foldr(xmy, 0, L) = (1-(5-(3-(8-0)))) = -9
foldr(ymx, 0, L) = ((((0-8)-3)-5)-1) = -17
```


Deklaratív programozási nyelvek — a Cékla tanulságai

- Mit vesztettünk?
  - a megváltoztatható változókat,
  - az értékadást, ciklus-utasítást stb.,
  - általánosan: a megváltoztatható állapotot
- Hogyan tudunk mégis állapotot kezelni deklaratív módon?
  - az állapotot a (segéd)függvény paraméterei tárolják,
  - az állapot változása (vagy helybenmaradása) explicit!
- Mit nyertünk?
  - Állapotmentes szemantika: egy nyelvi elem értelme nem függ a programállapottól
    - Hivatkozási átlátszóság (referential transparency) — pl. ha $f(x) = x^2$, akkor $f(a)$ **helyettesíthető** a^2-tel.
    - Egyszeres értékadás (single assignment) — párhuzamos végrehajthatóság.
  - A deklaratív programok **dekomponálhatók**:
    - A program részei egymástól **függetlenül** megírhatók, tesztelhetők, verifikálhatók.
    - A programon könnyű következtetéseket végezni, pl. helyességét bizonyítani

Cékla kis házi feladat – előzetes

Egy S pozitív szám A alapú *átrendezettje*, $A > 1$:

1. Képezzük az S szám A alapú számrendszerben vett jegyeinek listáját (balról jobbra), legyen ez $S_1, S_2, S_3, S_4, \ldots, S_k$
2. Rendezzük át a számjegyek listáját: elöl a páratlan indexű számjegyek jönnek, majd a páros indexűek.
   - Ha k páros, az eredmény: $S_1, \ldots, S_{k-1}, S_2, \ldots, S_k$
   - Ha k ptlan, az eredmény: $S_1, \ldots, S_k, S_2, \ldots, S_{k-1}$
3. Az előállt számjegysorozatot A számrendszerbeli számnak tekintjük, és képezzük ennek a számnak az értékét

```
|* atrendezett(123456, 10);
135246
|* atrendezett(12345, 10);
13524
|* atrendezett(12, 2);
10
```

1. A 12 (decimális) szám 2-es alapú számjegyei 1, 1, 0, 0,
2. hátratéve a páros indexűeket: 1, 0, 1, 0,
3. ezek 2-es számrendszerben a 10 (decimális) szám számjegyei.

III. rész

Erlang alapok

1 Bevezetés
2 Cékla: deklaratív programozás C++-ban
3 Erlang alapok

Tartalom

3 Erlang alapok
- Bevezetés
- Típusok
- Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabbrendű függvények, függvényérték
- Műveletek, beépített függvények
- Őr

Funkcionális programozás: mi az?

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

Példák funkcionális programozási nyelvekre, nyelvcsaládokra:

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

Az Erlang nyelv

- 1985: megszületik „Ellemtelben” (Ericsson–Televerket labor)
  - Agner Krarup Erlang dán matematikus, ERicsson LANGuage
- 1991: első megvalósítás, első projektek
- 1997: első OTP (Open Telecom Platform)
- 1998-tól: nyílt forráskódú, szabadon használható
  `http://www.erlang.org/`
- Funkcionális alapú (Functionally based)
- Párhuzamos programozást segítő (Concurrency oriented)
- Hatékony hibakezelés, hibatűrő (Fault tolerance)
- Gyakorlatban használt
  `http://en.wikipedia.org/wiki/Erlang_(programming_language)#Distribution`

„Programming is fun!”

Erlang emulátor

- Erlang shell (interaktív értelmező) indítása

```
$ erl
Erlang R13B03 (erts-5.7.4) [source] [smp:...

Eshell V5.7.4  (abort with ^G)
1>
1> 3.2 + 2.1 * 2.    % Lezárás és indítás ,,pont-bevitel''-lel!
7.4
2> atom.
atom
3> 'Atom'.
'Atom'
4> "string".
"string"
5> {ennes, 'A', a, 9.8}.
{ennes,'A',a,9.8}
6> [lista, 'A', a, 9.8].
[lista,'A',a,9.8]
7> q().
ok
```

Erlang shell: parancsok

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


Erlang shell: ^G és ^C

- ^G hatása

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

- ^C hatása

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


Saját program lefordítása

bevezeto.erl – Faktoriális számítása

```
-module(bevezeto).   % A modul neve (kötelező; modulnév = fájlnév)
-export([fac/1]).    % Látható függvények (praktikusan kötelező)

% @spec fac(N::integer()) -> F::integer().
% F = N! (F az N faktoriálisa).
fac(0) -> 1;                 % ha az N=0 mintaillesztés sikeres
fac(N) -> N * fac(N-1).      % ha az N=0 mintaillesztés nem volt sikeres
```

- Fordítás, futtatás

```
1> c(bevezeto).
{ok,bevezeto}
2> bevezeto:fac(5).
120
3> fac(5).
** exception error: undefined shell command fac/1
4> bevezeto:fac(5)
4>
4> .
120
```


Listakezelés – rövid példák (1)

```
1> L1 = [10,20,30].      % új változó kötése, '=' a mintaillesztés
[10,20,30]
2> H = hd(L1).           % hd: Built-in function (BIF)
10
3> T = tl(L1).           % tl: Built-in function
[20,30]
4> b().                  % kötött változók kiírása, lásd help().
H = 10
L1 = [10,20,30]
T = [20,30]
ok
5> T =:= [20|[30|[]]].   % egyenlőségvizsgálat
true
6> tl([]).
** exception error: bad argument
     in function  tl/1
        called as tl([])
7> c(bevezeto).
{ok,bevezeto}
```

Listakezelés – rövid példák (2)

`bevezeto.erl` – folytatás

```
% sum(L) az L lista összege.
sum([]) -> 0;
sum(L)  -> H = hd(L), T = tl(L), H + sum(T).

% append(L1, L2) az L1 lista L2 elé fűzve.
append([], L2) -> L2;
append(L1, L2) -> [hd(L1)|append(tl(L1), L2)].

% revapp(L1, L2) az L1 megfordítása L2 elé fűzve.
revapp([], L2) -> L2;
revapp(L1, L2) -> revapp(tl(L1), [hd(L1)|L2]).
```

```
8> bevezeto:sum(L1).
60
9> bevezeto:append(L1, [a,b,c,d]).
[10,20,30,a,b,c,d]
10> bevezeto:revapp(L1, [a,b,c,d]).
[30,20,10,a,b,c,d]
```


Tartalom

3 Erlang alapok
- Bevezetés
- Típusok
- Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabbrendű függvények, függvényérték
- Műveletek, beépített függvények
- Őr

Típusok

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

magyarul	*angolul*
Szám (egész, lebegőpontos)	Number (integer, float)
Atom vagy Névkonstans	Atom
Függvény	Function
Ennes (rekord)	Tuple (record)
Lista	List

  - További típusok

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

Szám (number)

- Egész
  - Pl. `2008`, `-9`, `0`
  - Tetszőleges számrendszerben `radix#szám` alakban, pl. `2#101001`, `16#fe`
  - Az egész korlátlan pontosságú, pl. `12345678901234567890123456789012345678901234`
  - Karakterkód
    - Ha nyomtatható: `$z`
    - Ha vezérlő: `$\n`
    - Oktális számmal: `$\012`
- Lebegőpontos
  - Pl. `3.14159`, `0.2e-22`
  - IEEE 754 64-bit

Atom

- Névkonstans (nem füzér!)
- Kisbetűvel kezdődő, bővített alfanumerikus[1] karaktersorozat, pl. `sicstus`, `erlang_OTP`
- Bármilyen[2] karaktersorozat is az, ha egyszeres idézőjelbe tesszük, pl. `'SICStus'`, `'erlang OTP'`, `'35 May'`
- Hossza tetszőleges, vezérlőkaraktereket is tartalmazhat, pl.
  `'ez egy hosszú atom, ékezetes betűkkel spékelve'`
  `'formázókarakterekkel \n\c\f\r'`
- Saját magát jelöli
- Hasonló a Prolog névkonstanshoz (atom)
- C++, Java nyelvekben a legközelebbi rokon: enum

[1] Bővített alfanumerikus: kis- vagy nagybetű, számjegy, aláhúzás (_), kukac (@)
[2] bármilyen latin-1 kódolású karaktert tartalmazó (R14B)

Függvény

- A függvény is érték: változóhoz köthető, adatszerkezet eleme lehet, ...
- Példák:

```
1> F = fun bevezeto:fac/1.
#Fun<bevezeto.fac.1>
2> F(6).
720
3> L = [fun erlang:'+'/2, fun erlang:'-'/2].
[#Fun<erlang.+.2>,#Fun<erlang.-.2>]
4> (hd(L))(4,3).
7
```

- Részletesen később, a „Magasabbrendű függvények” szakaszban

Ennes (tuple)

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

Lista (list)

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

```
11> ['egy'|['két']] ++ [elem,123|[3.14,'elem']]
[egy,két,elem,123,3.14,elem]
```

Füzér (string)

- Csak rövidítés, tkp. karakterkódok listája, pl.
  `"erl"` ≡ `[$e,$r,$l]` ≡ `[101,114,108]`
- Az Eshell a nyomtatható karakterkódok listáját füzérként írja ki:
```
12> [101,114,108]
"erl"
```
- Ha más érték is van a listában, listaként írja ki:
```
13> [31,101,114,108]
[31,101,114,108]
14> [a,101,114,108]
[a,101,114,108]
```
- Egymás mellé írással helyettesíthető, pl.
```
15> "erl" "ang".
"erlang"
```


Tartalom

3 Erlang alapok
- Bevezetés
- Típusok
- Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabbrendű függvények, függvényérték
- Műveletek, beépített függvények
- Őr

Term, változó

Term

- *Közelítő rekurzív definíció:* szám-, atom-, vagy függvénytípusú értékekből vagy *termekből* ennes- és listakonstruktorokkal felépített kifejezés.
- Néhány típussal (ref., port, pid, binary) most nem foglalkozunk
- Tovább nem egyszerűsíthető kifejezés, érték a programban
- Minden termnek van típusa
- Pl. `10` vagy `{'Diák Detti', [{khf, [cekla, prolog, erlang, prolog]}]}`
  Pl. nem term: `5 + 6`, mert műveletet (függvényhívást) tartalmaz
- Termek összehasonlítási sorrendje (v.ö. típusok)
  `number < atom < ref. < fun < port < pid < tuple < list < binary`

Változó

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

Kifejezés

- Lehet
  - term
  - változó
  - minta
    - Minta: term alakú kifejezés, amelyben szabad változó is lehet
    - termek ⊂ minták [3]
    - változók ⊂ minták

  továbbá
  - összetett kifejezés, függvényalkalmazás
  - szekvenciális kifejezés
  - egyéb: if, case, try/catch, catch stb.
- Kifejezés kiértékelése alapvetően: **mohó** (eager vagy strict evaluation).

```
16> Nevezo = 0.
0
17> (Nevezo > 0) and (1 / Nevezo > 1).
** exception error: bad argument in an arithmetic expression
```

[3] néhány nem túl gyakorlatias ellenpéldától eltekintve, például hibás:
`[X,fun erlang:'+'/2] = [5,fun erlang:'+'/2].`

Kifejezés: összetett és függvényalkalmazás

Függvényalkalmazás

- Szintaxisa
  - `fnév(arg1, arg2, ..., argn)`
    vagy
  - `modul:fnév(arg1, arg2, ..., argn)`
- Például

```
18> length([a,b,c]).
3
19> erlang:tuple_size({1,a,'A',"1aA"}).
4
20> erlang:'+'(1,2).
3
```

Összetett kifejezés

- Kiértékelhető műveleteket, függvényeket is tartalmazó, kifejezés, pl. 5+6, vagy: `[{5+6, math:sqrt(2+fib(X))}, alma]`
- Különbözik a termtől, a termben a művelet/függvényhívás tiltott

Kifejezés: szekvenciális

Szekvenciális kifejezés

- Kifejezések sorozata, szintaxisa:
  - **begin** `exp1, exp2, ..., expn` **end**
  - `exp1, exp2, ..., expn`
- A `begin` és `end` párt akkor kell kiírni, ha az adott helyen egyetlen kifejezésnek kell állnia
- Értéke az utolsó kifejezés értéke: `expn`
-

```
21> L2 = [10,20,30], H2 = hd(L2), T2 = tl(L2),
21> H2 + bevezeto:sum(T2).
60
22> [begin a, "a", 5, [1,2] end, a].
[[1,2],a]
```

- Eltérés Prologtól (gyakori hiba): a vessző itt nem jelent logikai ÉS kapcsolatot, csak egymásutániságot!
  - exp_i-ben ($i < n$) vagy változót kötünk,
  - vagy mellékhatást keltünk (pl. kiírás)

Függvénydeklaráció

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

```
fnév(A11, ..., A1m) [when ŐrSz1] -> SzekvenciálisKif1;
...
fnév(An1, ..., Anm) [when ŐrSzn] -> SzekvenciálisKifn.
```

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

```
fac(N) -> fac(N, 1).

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

- (Őrök bemutatása kicsit később)

Tartalom

3 Erlang alapok
- Bevezetés
- Típusok
- Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabbrendű függvények, függvényérték
- Műveletek, beépített függvények
- Őr

Minta, mintaillesztés (egyesítés)

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

`Pi = 3.14159`	$\leadsto$[4]	`Pi` $\mapsto$[5] `3.14159`
`3 = P`	$\leadsto$	hiba (jobboldal nem tömör)
`[H1|T1] = [1,2,3]`	$\leadsto$	`H1` $\mapsto$ `1`, `T1` $\mapsto$ `[2,3]`
`[1,2|T2] = [1,2,3]`	$\leadsto$	`T2` $\mapsto$ `[3]`
`[H2|[3]] = [1,2,3]`	$\leadsto$	meghiúsulás, hiba
`{A1,B1} = {{a},'Beta'}`	$\leadsto$	`A1` $\mapsto$ `{a}`, `B1` $\mapsto$ `'Beta'`
`{{a},B2} = {{a},'Beta'}`	$\leadsto$	`B2` $\mapsto$ `'Beta'`

[4] `Kif` $\leadsto$ jelentése: „`Kif` kiértékelése után”.
[5] `X` $\mapsto$ `V` jelentése: „`X` a `V` értékhez van kötve”.

Mintaillesztés függvény klózaira – 1. példa

- Függvény alkalmazásakor a klóz kiválasztása is mintaillesztéssel történik
- Máshol is, pl. a `case` vezérlési szerkezetnél is történik illesztés

`khf.erl` – DP kisházik ellenőrzése

```
-module(khf).
-compile(export_all).        % mindent exportál, csak teszteléshez!
%-export([kiadott/1, ...]). % tesztelés után erre kell cserélni

% kiadott(Ny) az Ny nyelven kiadott kisházik száma.
kiadott(cekla)  -> 1;        % 1. klóz
kiadott(prolog) -> 3;        % 2. klóz
kiadott(erlang) -> 3.        % 3. klóz
```

```
2> khf:kiadott(cekla).  % sikeres illesztés az 1. klózra
1
3> khf:kiadott(erlang). % sikertelen: 1-2. klóz, sikeres: 3. klóz
3
4> khf:kiadott(java).   % 3 sikertelen illesztés után hiba
** exception error: no function clause matching ...
```


Mintaillesztés függvény klózaira – 2. példa

- Hányszor szerepel egy elem egy listában? Első megoldásunk:

`khf.erl` – folytatás

```
% @spec elofordul0(E::term(), L::[term()]) -> N::integer().
% E elem az L listában N-szer fordul elő.
elofordul0(E, [])          -> 0;                    % 1.
elofordul0(E, [E|Farok])   -> 1 + elofordul0(E, Farok); % 2.
elofordul0(E, [Fej|Farok]) -> elofordul0(E, Farok).     % 3.
```

```
5> khf:elofordul0(a, [a,b,a,1]).  % 2. klóz, majd 3., 2., 3., 1.
2
6> khf:elofordul0(java, [cekla,prolog,prolog]).  % 3., 3., 3., 1.
0
```

- A minták összekapcsolhatóak, az `E` változó több argumentumban is szerepel: `elofordul0(E, [E|Farok]) -> ...`
- Számít a klózok sorrendje, itt pl. a 3. általánosabb, mint a 2.!

Kitérő: változók elnevezése

- Az előző függvényre figyelmeztetést kapunk:

```
Warning: variable 'E' is unused
Warning: variable 'Fej' is unused
```

- A figyelmeztetés kikapcsolható alulvonással (_) kezdődő változóval

`khf.erl` – folytatás

```
elofordul1(_E, [])          -> 0;
elofordul1(E, [E|Farok])    -> 1 + elofordul1(E, Farok);
elofordul1(E, [_Fej|Farok]) -> elofordul1(E, Farok).
```

- A változó neve akár el is hagyható, de az _ elnevezésű változót tömör kifejezésben nem lehet használni (vagyis nem lehet kiértékelni)
- Több _ változónk is lehet, például:
  `[H,_,_] = [1,2,3]` $\leadsto$ `H` $\mapsto$ `1`
- Találós kérdés: miben különböznek az alábbi mintaillesztések?

```
A=hd(L).
[A|_]=L.
[A,_|_]=L.
```

Mintaillesztés függvény klózaira – 3. példa

- Teljesítette-e egy hallgató a khf követelményeket?

```
7> Hallgato1 = {'Diák Detti',
                [{khf, [cekla,prolog,erlang,prolog]},
                 {zh, 59}]}.
```

khf.erl – folytatás

```
% @spec megfelelt(K::kovetelmeny(), H::hallgato()) -> true | false.
megfelelt(khf, {_Nev, [{khf, L}|_]}) ->
    C = elofordul1(cekla, L),
    P = elofordul1(prolog, L),
    E = elofordul1(erlang, L),
    (P >= 1) and (E >= 1) and (C + P + E >= 3);
megfelelt(zh, {_Nev, [{zh, Pont}|_]}) ->
    Pont >= 24;
megfelelt(K, {Nev, [_|F]}) ->
    megfelelt(K, {Nev, F});
megfelelt(_, {_, []}) ->
    false.
```


„Biztonságos” illesztés: ha egyik mindig sikerül

- Mit kezdjünk a `kiadott(java)` kiértékelésekor keletkező hibával?
- Erlangban gyakori: jelezzük a sikert vagy a hibát az eredményben

khf.erl – folytatás

```
% @spec safe_kiadott(Ny::atom()) -> {ok, Db::integer()} | error.
% Az Ny nyelven Db darab kisházit adtak ki.
safe_kiadott(cekla)  -> {ok, 1};
safe_kiadott(prolog) -> {ok, 3};
safe_kiadott(erlang) -> {ok, 3};
safe_kiadott(_Ny)    -> error.   % e klóz mindig illeszthető
```

- Az `ok` és az `error` atomokat konvenció szerint választottuk
- Kötés: ha a minta egyetlen szabad változó (`_Ny`), az illesztés sikeres
- De hogy férjünk hozzá az eredményhez?

```
8> khf:safe_kiadott(cekla).
{ok,1}
9> khf:safe_kiadott(java).
error
```


Feltételes kifejezés mintaillesztéssel (case)

- **case** Kif **of**
  Minta$_1$ [**when** ŐrSz$_1$] -> SzekvenciálisKif$_1$;
  ...
  Minta$_n$ [**when** ŐrSz$_n$] -> SzekvenciálisKif$_n$
  end.
- Kiértékelés: balról jobbra
- Értéke: az első illeszkedő minta utáni szekvenciális kifejezés
- Ha nincs ilyen minta, hibát jelez

```
1> X=2, case X of 1 -> "1"; 3 -> "3" end.
** exception error: no case clause matching 2
2> X=2, case X of 1 -> "1"; 2 -> "2" end.
"2"
3> Y=fagylalt, 3 * case Y of fagylalt -> 100; tolcser -> 15 end.
300
4> Z=kisauto, case Z of fagylalt -> 100;
4>                      tolcser  -> 15;
4>                      Barmi    -> 99999 end.
99999
```


case példa

- Az adott nyelvből hány százalékot adtunk be?

khf.erl – folytatás

```
% @spec safe_teljesitmeny(Nyelv::atom(), Beadott_db::integer()) ->
%     {ok, Teljesitmeny::float()} | error.
safe_teljesitmeny(Nyelv, Beadott_db) ->
    case safe_kiadott(Nyelv) of
        {ok, Kiadott_db} -> {ok, Beadott_db / Kiadott_db};
        error            -> error
    end.
```

- Függvény klózai összevonhatóak a `case` segítségével:

```
kiadott(cekla)  -> 1;
kiadott(prolog) -> 3;
kiadott(erlang) -> 3.
```

helyett írható:

```
kiadott(Ny)  ->
    case Ny of
        cekla  -> 1;
        prolog -> 3;
        erlang -> 3
    end.
```

Tartalom

3 Erlang alapok
- Bevezetés
- Típusok
- Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabbrendű függvények, függvényérték
- Műveletek, beépített függvények
- Őr

Listanézet (List Comprehensions)

- Listanézet (List Comprehensions): `[Kif || Minta <- Lista, Feltétel]`
  Közelítő definíció: `Kif` kifejezések *listája*, ahol a `Minta` a `Lista` olyan eleme, melyre `Feltétel` igaz.
- `Feltétel` tetszőleges logikai (`true` v. `false` atom értékű) kifejezés lehet. A `Mintá`ban előforduló változónevek elfedik a listakifejezésen kívüli azonos nevű változókat.
- Kis példák

```
1> [2*X || X <- [1,2,3]].   %  { 2·x | x ∈ {1,2,3} }
[2,4,6]
2> [2*X || X <- [1,2,3], X rem 2 =/= 0, X > 2].
[6]
3> lists:seq(1,3).          %  egészek 1-től 3-ig
[1,2,3]
4> [{X,Y} || X <- [1,2,3,4], Y <- lists:seq(1,X)].
[{1,1},
 {2,1},{2,2},
 {3,1},{3,2},{3,3},
 {4,1},{4,2},{4,3},{4,4}]
```

- Pontos szintaxis: `[X || Q1, Q2, ...]`, ahol `X` tetszőleges kifejezés, Q_i lehet generátor (`Minta <- Lista`) vagy szűrőfeltétel (predikátum)

Listanézet: példák

- Pitagoraszi számhármasok, melyek összege legfeljebb *N*

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

- Hányszor fordul elő egy elem egy listában?

```
elofordul2(Elem, L) ->
    length([X || X <- L, X=:=Elem]).
```

- A khf követelményeket teljesítő hallgatók

```
L = [{'Diák Detti', [{khf, [...]}]}, {'Lusta Ludvig', []}],
[Nev || {Nev, M} <- L, khf:megfelelt(khf, {Nev, M})].
```


Listanézet: érdekes példák

- Quicksort

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

- Permutáció

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

  - Listák különbsége: `As--Bs` vagy `lists:subtract(As,Bs)`
    `As--Bs` az `As` olyan másolata, amelyből ki van hagyva a `Bs`-ben előforduló összes elem balról számított első előfordulása, feltéve, hogy volt ilyen elem `As`-ben

Tartalom

3 Erlang alapok
- Bevezetés
- Típusok
- Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabbrendű függvények, függvényérték
- Műveletek, beépített függvények
- Őr

Függvényérték

- A funkcionális nyelvekben a függvény is *érték:*
  - leírható (jelölhető)
  - van típusa
  - névhez (változóhoz) köthető
  - adatszerkezet eleme lehet
  - paraméterként átadható
  - függvényalkalmazás eredménye lehet (zárójelezni kell!)
- Névtelen függvény (függvényjelölés) mint érték

```
fun (A11, ..., A1m) [when ŐrSz1] -> SzekvenciálisKif1;
    ...;
    (An1, ..., Anm) [when ŐrSzn] -> SzekvenciálisKifn
end.
```

- Már deklarált függvény mint érték

```
fun Modul:Fnev/Aritas  % például fun bevezeto:sum/1
fun Fnev/Aritas        % ha az Fnev „látható", pl. modulból
```


Függvényérték: példák

```
2> Area1 = fun ({circle,R})   -> R*R*3.14159;
               ({rectan,A,B}) -> A*B;
               ({square,A})   -> A*A
           end.
#Fun<erl_eval.6.13229925>
3> Area1({circle,2}).
12.56636
4> Osszeg = fun bevezeto:sum/1.
#Fun<bevezeto.sum.1>
5> Osszeg([1,2]).
3
6> fun bevezeto:sum/1([1,2]).
3
7> F1 = [Area1, Osszeg, fun bevezeto:sum/1, 12, area].
[#Fun<erl_eval.6.13229925>,#Fun<bevezeto.sum.1>,...]
8> (hd(F1))({circle, 2}). % külön zárójelezni kell!
12.56636
% hd/1 itt magasabbrendű függvény, zárójelezni kell értékét
```


Magasabb rendű függvények alkalmazása – `map`, `filter`

- Magasabb rendű függvény: paramétere vagy eredménye függvény
- Leképzés: `lists:map(Fun, List)` A `List` lista `Fun`-nal transzformált elemeiből álló lista

```
9> lists:map(fun erlang:length/1, ["alma", "korte"]).
[4,5]       % erlang:length/1: Built-In Function, lista hossza
10> lists:map(Osszeg, [[10,20], [10,20,30]]).
[30,60]
11> L = [{'Diák Detti', [{khf, [...]}]}, {'Lusta Ludvig', []}].
[{'Diák Detti',[{khf,[...]}]},{'Lusta Ludvig',[]}]
12> lists:map(fun(Hallg) -> khf:megfelelt(khf, Hallg) end, L).
[true,false]
```

- Szűrés: `lists:filter(Pred, List)`
  A `List` lista `Pred`-et kielégítő elemeinek listája

```
13> lists:filter(fun erlang:is_number/1, [x, 10, L, 20, {}]).
[10,20]
14> lists:filter(fun(Hallg) -> khf:megfelelt(khf, Hallg) end, L).
[{'Diák Detti',[{khf,[...]}]}]
```

Magasabb rendű függvények alkalmazása – `filter` példa

- Hányszor szerepel egy elem egy listában? Új megoldásunk:

`khf.erl` – folytatás

```
% @spec elofordul3(E::term(), L::[term()]) -> N::integer().
% E elem az L listában N-szer fordul elő.
elofordul3(Elem, L) ->
    length(lists:filter(fun(X) -> X =:= Elem end, L)).
```

```
15> khf:elofordul3(prolog, [cekla,prolog,prolog]).
2
```

- A névtelen függvényben felhasználhatjuk az `Elem` lekötött változót!
- A `filter/2` egy lehetséges megvalósítása:

```
filter(_, [])          -> [];
filter(P, [Fej|Farok]) -> case P(Fej) of
                              true  -> [Fej|filter(P,Farok)];
                              false -> filter(P,Farok)
                          end.
```

- *Fejtörő:* miért lehet érdemes leírni kétszer a `filter(P,Farok)` hívást?

Redukálás a `fold` függvényekkel

- Jobbról balra haladva: `lists:foldr(Fun,Acc,List)`
- Balról jobbra haladva: `lists:foldl(Fun,Acc,List)`
- A `List` lista elemeiből és az `Acc` elemből a kétoperandusú `Fun`-nal képzett érték

```
lists:foldr(fun(X, Acc) -> X - Acc end, 0, [1,2,3,4]) ≡ -2
lists:foldl(fun(X, Acc) -> X - Acc end, 0, [1,2,3,4]) ≡ 2
```

- Példa `foldr` kiértékelési sorrendjére: `1-(2-(3-(4-0)))` = -2
  Példa `foldl` kiértékelési sorrendjére: `4-(3-(2-(1-0)))` = 2

```
                                % plus(X, Sum) -> X + Sum.
    sum(Acc, []) ->             foldr(Fun, Acc, []) ->
R       Acc;                        Acc;
    sum(Acc, [H|T]) ->          foldr(Fun, Acc, [H|T]) ->
        plus(H, sum(Acc, T)).       Fun(H, foldr(Fun, Acc, T)).

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


Tartalom

3 Erlang alapok
- Bevezetés
- Típusok
- Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabbrendű függvények, függvényérték
- Műveletek, beépített függvények
- Őr

Listaműveletek

- Alapműveletek: `hd(L)`, `tl(L)`, `length(L)` (utóbbi lassú: $O(n)$!)
- Listák összefűzése (`As` $\oplus$ `Bs`): `As++Bs` vagy `lists:append(As,Bs)`
  `Cs = As++Bs` $\rightsquigarrow$ `Cs` $\mapsto$ az `As` összes eleme a `Bs` elé fűzve az eredeti sorrendben
- Példa

```
1> [a,'A',[65]]++[1+2,2/1,'A'].
[a,'A',"A",3,2.0,'A']
```

- Listák különbsége: `As--Bs` vagy `lists:subtract(As,Bs)`
  `Cs = As--Bs` $\rightsquigarrow$ `Cs` $\mapsto$ az `As` olyan másolata, amelyből ki van hagyva a `Bs`-ben előforduló összes elem balról számított első előfordulása, feltéve, hogy volt ilyen elem `As`-ben
- Példa

```
1> [a,'A',[65],'A']--["A",2/1,'A'].
[a,'A']
2> [a,'A',[65],'A']--["A",2/1,'A',a,a,a].
['A']
3> [1,2,3]--[1.0,2].  % erős típusosság:  1 ≠ 1.0
[1,3]
```

Aritmetikai műveletek

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

Relációk

- Termek összehasonlítási sorrendje (v.ö. típusok):
  `number < atom < ref. < fun < port < pid < tuple < list < binary`
- Kisebb-nagyobb reláció
  `<`, `=<`, `>=`, `>`
- Egyenlőségi reláció (aritmetikai egyenlőségre is):
  `==`, `/=` **ajánlás: helyette azonosan egyenlőt használjunk!**
- Azonosan egyenlő (különbséget tesz `integer` és `float` közt):
  `=:=`, `=/=` Példa:
  ```
  1> 5.0 =:= 5.
  false
  ```
- Az összehasonlítás eredménye a `true` vagy a `false` atom
- 
  Lebegőpontos értékre kerülendő:
  `== =< >= =:=`
  Elrettentő példák:
  ```
  1> 10.1 - 9.9 == 0.2.
  false
  2> 0.0000000000000001 + 1 == 1.
  true
  ```
- Kerekítés (`float` ↦ `integer`), explicit típuskonverzió (`integer` ↦ `float`):
  `erlang:trunc/1`, `erlang:round/1`, `erlang:float/1`

Logikai műveletek

- Logikai művelet:
  `not`, `and`, `or`, `xor`
- Csak a `true` és `false` atomokra alkalmazhatóak
- Lusta kiértékelésű („short-circuit”) logikai művelet:
  `andalso`, `orelse`
- Példák:
  ```
  1> false and (3 div 0 =:= 2).
  ** exception error:  bad argument in an arithmetic expression

  2> false andalso (3 div 0 =:= 2).
  false
  ```


Beépített függvények (BIF)

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

További BIF-ek

- Rendszer:
  `date()`, `time()`, `erlang:localtime()`, `halt()`
- Típusvizsgálat
  - `is_integer(Term)`, `is_float(Term)`,
  - `is_number(Term)`, `is_atom(Term)`,
  - `is_boolean(Term)`,
  - `is_tuple(Term)`, `is_list(Term)`,
  - `is_function(Term)`, `is_function(Term,Arity)`
- Típuskonverzió
  - `atom_to_list(Atom)`, `list_to_atom(String)`,
  - `integer_to_list(Int)`, `list_to_integer(String)`, `erlang:list_to_integer(String, Base)`,
  - `float_to_list(Float)`, `list_to_float(String)`,
  - `tuple_to_list(Tuple)`, `list_to_tuple(List)`
- Érdekesség: a BIF-ek mellett megtalálhatóak az operátorok az `erlang` modulban, lásd az `m(erlang).` kimenetét, pl. `fun erlang:'*'/2(3,4).`

Tartalom

3 Erlang alapok
- Bevezetés
- Típusok
- Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabbrendű függvények, függvényérték
- Műveletek, beépített függvények
- Őr

Őr: strukturális mintaillesztés finomítása

- Nézzük újra a következő definíciót:

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

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

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

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


Ismétlés: függvénydeklaráció, `case`

- Függvénydeklaráció:

```
fnév(A11, ..., A1m) [when ŐrSz1] -> SzekvenciálisKif1;
...
fnév(An1, ..., Anm) [when ŐrSzn] -> SzekvenciálisKifn.
```

- Feltételes mintaillesztés (`case`):

```
case Kif of
    Minta1 [when ŐrSz1] -> SzekvenciálisKif1;
    ...
    Mintan [when ŐrSz1n] -> SzekvenciálisKifn
end.
```

Őrkifejezés

Őrkifejezés lehet:

- Term (vagyis konstans érték)
- Kötött változó
- Őrkifejezésekből aritmetikai, összehasonlító és logikai műveletekkel felépített kifejezés
- Őrkifejezéseket tartalmazó ennes vagy lista
- Bizonyos BIF-ek őrkifejezéssel paraméterezve:
  - Típust vizsgáló predikátumok (`is_TÍPUS`)
  - `abs(Number) round(Number) trunc(Number) float(Term)`
    `element(N, Tuple) tuple_size(Tuple)`
    `hd(List) length(List) tl(List)`
    `bit_size(Bitstring) byte_size(Bitstring) size(Tuple|Bitstring)`
    `node() node(Pid|Ref|Port) self()`

Őrkifejezés **nem** lehet:

- Függvényalkalmazás, mert esetleg mellékhatása lehet vagy lassú
- `++` (`lists:append/2`), `--` (`lists:subtract/2`)