II. rész

Cékla: deklaratív programozás C++-ban

Tartalom

A deklaratív programozás jelmondata

- MIT és nem 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

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) = 1 <=> létezik i, melyre B^i = A.
 * Előfeltétel: A > 0, B > 1 */
int ispow(const int A, const int B) {
                                        // again:
 if (A == 1)       return 1;
 else if (A%B==0) return ispow(A/B, B); // {A=A/B; goto again;}
 else              return 0;
}
```

- Itt a színezett rekurzív hívás átírható iteratív kódra: a kommentben jelzett 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 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` 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 átírható 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

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

- Megszorítások:
  - Típusok: csak `int`, lista vagy függvény (lásd később)
  - Utasítások: `if-then-else`, `return`, blokk
  - Változók: csak egyszer, deklarálásukkor kaphatnak értéket
  - Kifejezések: változókból és konstansokból kétargumentumú operátorokkal és függvényhívásokkal 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
- Korábban készült egy csak az `int` típust és a C résznyelvét támogató Cékla 1 fordító (Prologban íródott és Prologra is fordít)
- A (szintén Prologos) Cékla 2.x fordító letölthető a tárgy honlapjáról

Cékla Hello world!

- hello.cc:

```
#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.cc                  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.cc && ./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

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);
```

- 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 2.238: a compiler for a declarative C++ sublanguage
To get help, type:      |* help;
|* load "pelda.c";
* 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 nemüres L lista fejét.
```

- Többi elem lekérdezése:

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

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

```
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
}           //  tl(L) 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: 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:

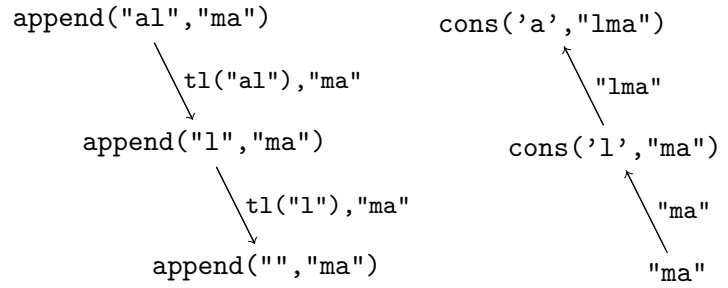
```
int main() {
  const list L4 = "abc";                    // abc
  const list L5 =
    cons(97,cons(98,cons(99,nil)));         // abc
  writeln(L4 == L5);                        // 1
  writeln(97 == 'a');                       // 1
  writeln(nil == "");                       // 1
  writeln(nil);                             // []
  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 általános 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 0 ;
  if (hd(L) == X) return 1;
  return ismember(X, tl(L));              }
```

- Két, ismétlődésmentes listaként ábrázolt halmaz metszete

```
// intersection(L1, L2) = L1 és L2 közös elemeinek listája.
list intersection(const list L1, const list L2) {
  if (L1 == nil) return nil;
  const list L3 = intersection(tl(L1), L2);
  const int X = hd(L1);
  if (ismember(X, L2))
    return cons(X,L3);
  else return L3;                                  }
```


Műveletek számlistákkal

- 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 1 ;
  if (hd(L) >= 0) return 0;
  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));           }
```

- Állítsuk elő egy számlista elemeinek négyzeteiből álló listát!

```
// sqlist(L) = az L lista elemeinek négyzeteit tartalmazó lista.
list sqlist(const list L)          {
  if (L == nil) return nil ;
  const int HD = hd(L);
  const list TL = tl(L);
  return cons(HD*HD, sqlist(TL));   }
```

Imperatív C programok átírása Céklába gyűjtőargumentumokkal

- Példafeladat: Hatékony hatványozási algoritmus
  - Alaplépés: a kitevő felezése, az alap négyzetre emelése.
  - A kitevő kettes számrendszerbeli alakja szerint hatványoz.
- A megoldás imperatív C-ben és Céklában:

```
// hatv(A, H) = A^H
int hatv(int A, int H) {
  int E = 1;
  while (H > 0) {
    if (H % 2) E *= A;
    A *= A; H /= 2;
  }
  return E;           }
```

```
// hatv(A, H, E) = E * A^H
int hatv(const int A, const int H, const int E) {

  if (H <= 0) return E;   // ciklus vége
  const int E1 = H%2 ? E*A : E;
  return hatv(A * A, H / 2, E1);
}
```

- A jobboldalon a while ciklusnak megfelelő segédfüggvény van, meghívása: `hatv(A, H, 1)`
- A segédfüggvény argumentumai a ciklusban szereplő változók: `A`, `H`, `E`
- Az `A` és `H` változók végértékére nincs szükség
- Az `E` változó végső értéke szükséges (ez a függvény eredménye)

Gyűjtőargumentumok: több kimenő változót tartalmazó ciklus

- A segédeljárás kimenő változóit egy listába „csomagolva” adjuk vissza
- A segédeljárás visszatérési értékét szétszedjük.

```
// poznegki(L): kiírja
// L >0 és <0 elemeinek
// a számát
int poznegki(list L)   {
  int P = 0, N = 0;
  while (L != nil)
    { int Fej = hd(L);
      P += (Fej > 0);
      N += (Fej < 0);
      L = tl(L);




    }
  write(P); write("-");
  writeln(N);          }
```

```
// pozneg(L, P0, N0) = [P,N], ahol P-P0, ill.
// N-N0 az L >0, ill. <0 elemeinek a száma
list pozneg(list L, int P0, int N0)          {
  if (L == nil) return cons(P0,cons(N0,nil));
  const int Fej = hd(L);
  const int P = P0+(Fej>0);
  const int N = N0+(Fej<0);
  return pozneg(tl(L), P, N);                }

int poznegki(const list L)                   {
  const list PN = pozneg(L, 0, 0);
  const int P = hd(PN), N = hd(tl(PN));
  write(P); write("-");
  writeln(N);                                }
```


Gyűjtőargumentumok: append kiküszöbölése

- A feladat: adott `N`-re `N` db `a`, majd `N` db `b` karakterből álló sztring előállítása
- Első változat, `append` felhasználásával

```
list anbn(int N)          {
  list LA = nil, LB = nil;
  while (N-- > 0)
    { LA = cons('a', LA);
      LB = cons('b', LB);
    }
  return append(LA, LB);  }
```

```
// an(A, N) = [A,<-N->,A]
list an(const int A, const int N)     {
  if (N == 0) return nil;
  else return cons(A, an(A, N-1));    }

list anbn(const int N)                {
  return append(an('a',N), an('b',N));}
```

- Második változat, `append` nélkül

```
list anbn(int N)          {
  list L = nil; int M = N;
  while (N-- > 0)
    L = cons('b', L);
  while (M-- > 0)
    L = cons('a', L);
  return L;               }
```

```
// an(A, N, L) = [A,<-N->,A] ⊕ L
list an(int A, int N, list L)         {
  if (N == 0) return L;
  else return an(A, N-1, cons(A, L)); }

list anbn(const int N)                {
  return an('a', N, an('b', N, nil)); }
```

- A gyűjtőargumentumok előnye nemcsak a jobbrekurzió, hanem az `append` kiküszöbölése is.

Tartalom

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 1;            // triviális
    else {
        if (P(hd(L)) == 0) return 0;   // 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 0;               // 48 == '0'
  if (X > '9') return 0;               // 57 == '9'
  return 1; }
int szamjegyek(const list L) { // Igaz, ha L egy szám sztringje
  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 négyzetre emelve: `[100,400,900]`

```
int sqr(const int X) { return X*X; }
```

Így a `map(sqr, L)` kifejezés értéke `[100,400,900]`.

- `filter(P,L)`: az `L` lista lista azon `X` elemei, melyekre `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: a `fold` család

- Hajtogatás balról – Erlang stílusban

```
// foldl(F, a, [x1,...,xn]) = F(xn, ..., 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));        }  // (*)
```

- Hajtogatás balról – Haskell stílusban (csak a `(*)` sor változik)

```
// foldlH(F, a, [x1,...,xn]) = F(...(F(a, x1), x2), ..., xn)
int foldlH(const fun2 F, const int Acc, const list L) {
  if (L == nil) return Acc;
  else return foldlH(F, F(Acc,hd(L)), 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
foldlH(xmy, 0, L) = ((((0-1)-5)-3)-8) = -17
foldl (ymx, 0, L) = ((((0-1)-5)-3)-8) = -17
foldlH(ymx, 0, L) = (8-(3-(5-(1-0)))) = 9
```


A `fold` család – folytatás

- Hajtogatás jobbról (Haskell és Erlang egyaránt:-)

```
// foldr(F, a, [x1, ..., xn]) = F(x1, ..., 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 példa, `L = [1,5,3,8]`,

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

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

- Egyes funkcionális nyelvekben (pl. Haskell-ben) a hajtogató függvényeknek létezik 2-argumentumú változata is: `foldl1` ill. `foldr1`
- Itt a lista első vagy utolsó eleme lesz az `Acc` akkumulátor-argumentum kezdőértéke, pl.
  `foldr1(F, [x1,...,xn]) = F(x1, F(x2, ..., F(xn-1,xn)...))`

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

Egy S szám A alapú *kevertje*:

1. Képezzük az S szám A alapú számrendszerben vett jegyeinek sorozatát (balról jobbra), legyen ez $J_1, J_2, J_3, J_4, \ldots, J_N$
2. Átrendezzük: a páratlan sorszámúakat előrevesszük:
   - Ha N páros: $J_1, J_2, J_3, J_4, \ldots, J_N \longrightarrow J_1, J_3, \ldots, J_{N-1}, J_2, J_4, \ldots, J_N$
   - Ha N páratlan: $J_1, J_2, J_3, J_4, \ldots, J_N \longrightarrow J_1, J_3, \ldots, J_N, J_2, J_4, \ldots$
3. Az előállt számjegysorozatot A számrendszerbeli számnak tekintjük, és képezzük ennek a számnak az értékét

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

1. A 10 (decimális) szám 2-es alapú számjegyei $1, 0, 1, 0$
2. átsorrendezve $1, 1, 0, 0$
3. ezek 2-es számrendszerben a 12 (decimális) szám számjegyei.