

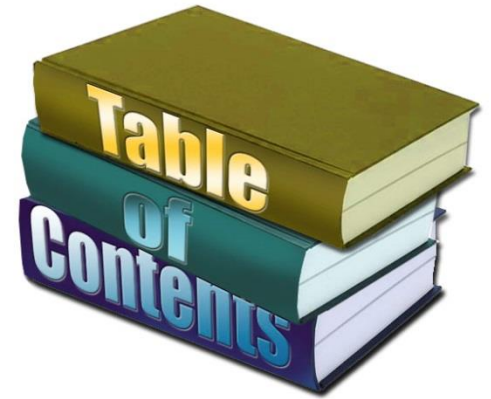
PROGRAMIRANJE

2014/15

repna rekurzija
funkcije višjega reda

Pregled

- rekurzivno ujemanje vzorcev
- sklepanje na podatkovni tip
- izjeme
- repna rekurzija
- funkcije višjega reda
- map in filter




Rekurzivno ujemanje vzorcev

- namesto vgnezenih stavkov `case` lahko vgnezdimo vzorce v vzorce (pri gnezdenju se tudi spremenljivke prilagodijo pravim vrednostim)

```
(glava1::rep1, glava2::rep2)
(glava::(drugi::(tretji::rep)))
((a1,b1)::rep)
...
```

- pri zapisovanju vzorcev lahko uporabimo anonimno spremenljivko `"_"`, ki se prilagodi delu izraza, ne veže pa rezultata na ime spremenljivke

```
fun dolzina (sez:int list) =
  case sez of
    [] => 0
  | _::rep => 1 + dolzina rep
```



anonimna spremenljivka (pri računanju dolžine seznama vrednosti elementov niso pomembne)

Primeri gnezdenja

Napiši naslednje programe:

1. Podana sta seznama `sez1` in `sez2`. Seštej njune istoležne komponente v novi seznam. Da program uspe, morata biti oba seznama enako dolga.

```
fn : int list * int list -> int list
```

2. Podan je seznam, ki predstavlja zaporedje, izračunano po Fibonaccijevem zakonu. Preveri, ali je seznam veljavno takšno zaporedje.

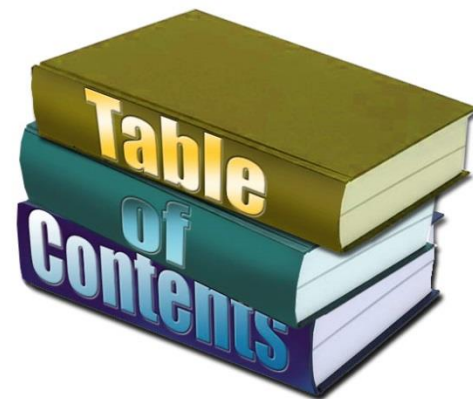
```
fn : int list -> bool
```

3. Napiši program, ki za dve celi števili pove, ali je rezultat po njunem seštevanju sodo število, liho število ali ničla.

```
fn : int * int -> sodost
```

Pregled

- rekurzivno ujemanje vzorcev
- sklepanje na podatkovni tip
- izjeme
- repna rekurzija
- funkcije višjega reda
- map in filter



Sklepanje na podatkovni tip

- SML ima vgrajen sistem za sklepanje na podatkovni tip definirane funkcije, tudi če ročno ne navajamo vhodnega in izhodnega tipa
- pogoji za delovanje sistema:
 - uporabljati moramo ujemanje vzorcev, s katerim opredelimo vse spremenljivke, ki nastopajo v programski kodi
 - povedano drugače: v programu ne smemo naslavljati komponent spremenljivke z `#zap_št` ali `#ime_polja` (v primeru uporabe `#...` je potrebno eksplicitno navajanje tipov)
 - zakaj?

```
- fun sestej stevili = #1 stevili + #2 stevili;  
stdIn:4.1-5.28 Error: unresolved flex record  
      (need to know the names of ALL the fields in this context)
```

```
- fun sestej (s1, s2) = s1 + s2;  
val sestej = fn : int * int -> int
```

Polimorfizem pri sklepanju na tip

- lahko se zgodi, da SML ugotovi, da so napisane funkcije bolj splošne, kot smo želeli
- tip je bolj splošen kot drugi tip, če lahko v njemu konsistentno zamenjamo bolj splošne tipe ('a, 'b, 'c) z manj splošnimi tipi (npr. vse 'a za int, vse 'b za string itd.)

```
fun zdruzi (sez1, sez2) =  
  case sez1 of  
    [] => sez2  
  | glava::rep => glava::zdruzi(rep, sez2)
```

```
val zdruzi = fn : 'a list * 'a list -> 'a list
```

```
fun sestej_zapis {prvi=a, drugi=b, tretji=c, cetrti=d, peti=e}  
= a+d
```

```
val sestej_zapis = fn  
  : {cetrti:int, drugi:'a, peti:'b, prvi:int, tretji:'c} -> int
```

Primeri specifičnih tipov

Kateri od naslednjih tipov so bolj specifični od tipa

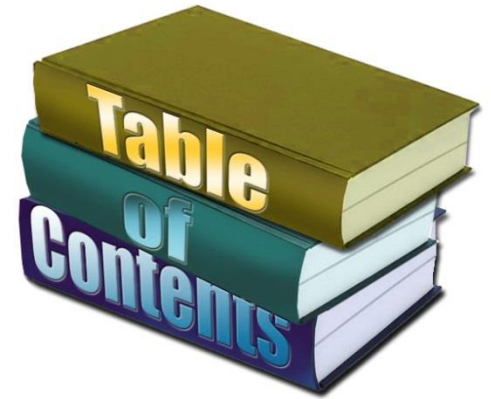
'a **list** * ('b * 'a) **list** -> 'a ?

1. string **list** * ((int*int) * string) **list** -> string
2. int **list** * (int * int) **list** -> int
3. (int*bool) **list** * (bool * (int*bool)) **list** -> (int*bool)
4. int **list list** * (bool * int **list**) **list** -> int **list**
5. int **option list** * (bool * int **list**) **option** -> int **list**
6. real **list** * string **list** -> real



Pregled

- rekurzivno ujemanje vzorcev
- sklepanje na podatkovni tip
- izjeme
- repna rekurzija
- funkcije višjega reda
- map in filter



Izjeme

- sporočajo o neveljavnih situacijah, do katerih je prišlo med izvajanjem programa
- definicija izjeme

```
exception MojaIzjema  
exception MojaIzjema of int
```

- klic izjeme

```
raise MojaIzjema  
raise MojaIzjema (7)
```

- obravnava izjeme

```
e1 handle MojaIzjema => e2  
e1 handle MojaIzjema (x) => e2
```

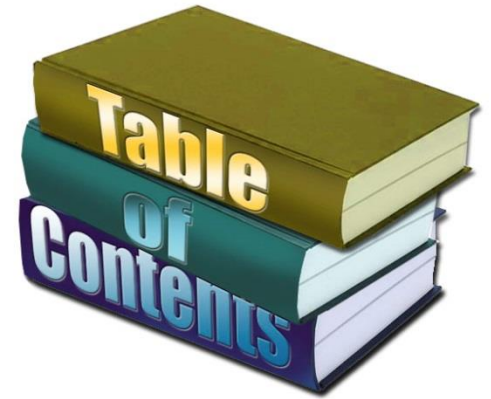
Izjeme

- izjeme so podatkovnega tipa `exn`
- uporabimo jih lahko tudi v argumentih funkcij
 - izjema v argumentu še ne proži izjeme, temveč jo samo opredeli
 - primer tipa funkcije: `fn : int * exn -> int list`
- stavek `handle` se uporablja za obravnavo izjem; uporablja lahko ujemanje vzorcev (kot stavek `case`) in ima lahko več različnih vej

```
fun deli (a1, a2, napaka) =  
  if a2 = 0  
  then raise napaka  
  else a1 div a2  
  
fun naredinekaj (stevilo, moja_napaka) =  
  deli(stevilo, 0, moja_napaka)  
  handle moja_napaka => ~99999999
```

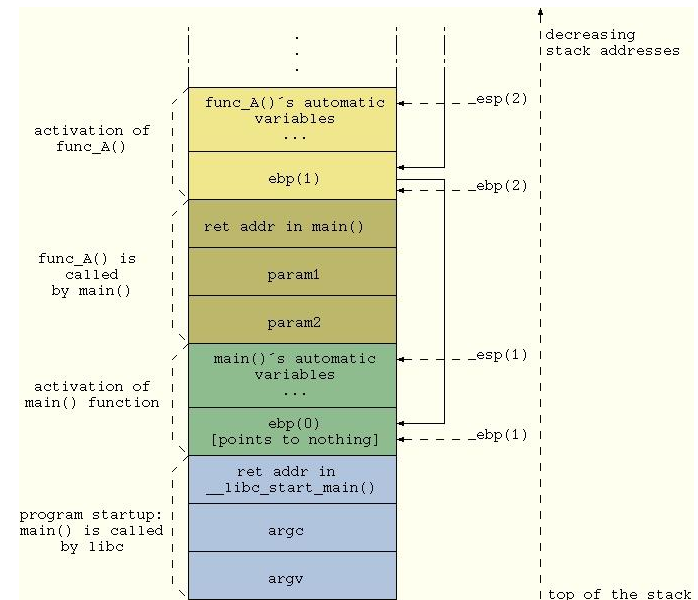
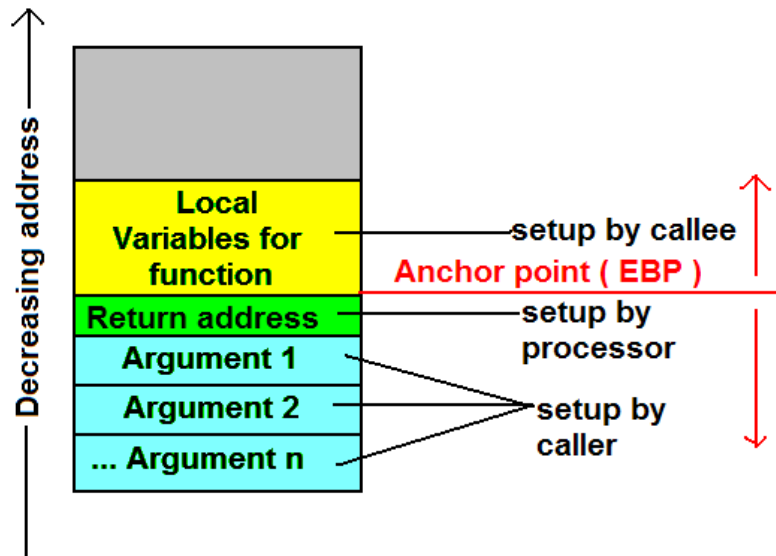
Pregled

- rekurzivno ujemanje vzorcev
- sklepanje na podatkovni tip
- izjeme
- repna rekurzija
- funkcije višjega reda
- map in filter



Repna rekurzija

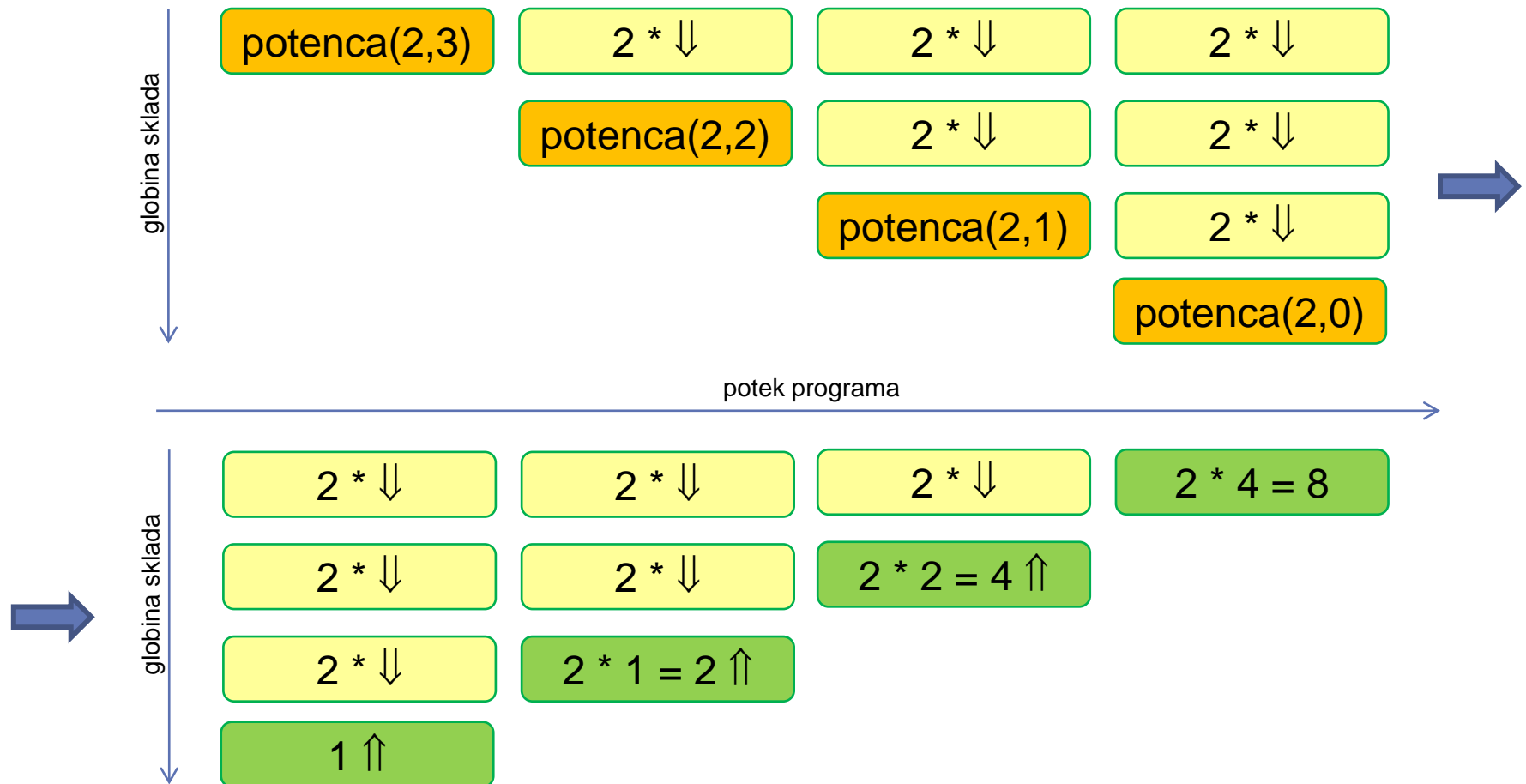
- **repna rekurzija** je bolj učinkovita od drugih oblik rekurzije
- razlog:
 - (v splošnem): pri vsakem klicu funkcije se **funkcijski okvir s kontekstom** potisne na sklad; ko se funkcija zaključi, se kontekst odstrani s sklada
 - pri repni rekurziji se okvir samo **zamenja** z novim (prihranek na prostoru in času), ker kličeča funkcija konteksta ne potrebuje več



Izvedba rekurzije

Primer "navadne" rekurzije:

```
fun potenca (x,y) = if y=0 then 1 else x * potenca (x, y-1)
```

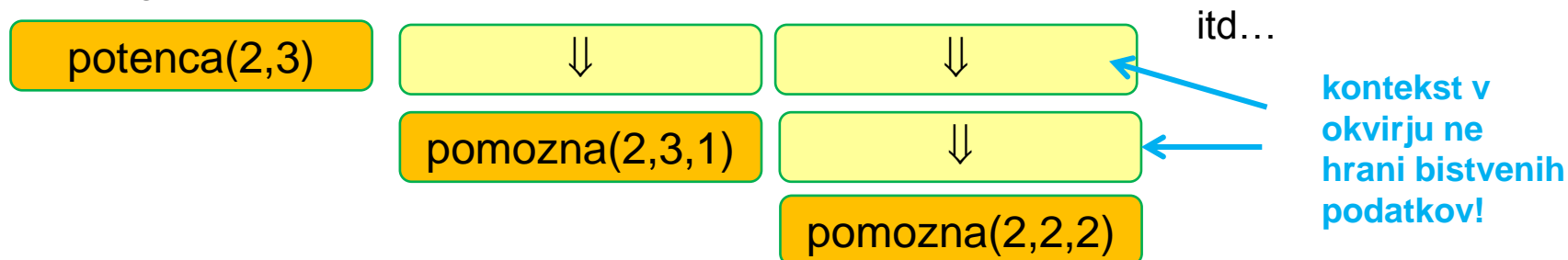


Drugačna implementacija

- alternativa: rekurzivna implementacija z **lokalno pomožno funkcijo**:
 - pomožna funkcija sprejema dodatni argument, imenovan **akumulator**
 - telo glavne funkcije vsebuje **samo klic** pomožne funkcije brez dodatnih operacij
 - klic pomožne funkcije v telesu glavne funkcije vsebuje **začetno vrednost** akumulatorja

```
fun potenca_repna (x,y) =  
  let  
    fun pomocna (x,y,acc) =  
      if y=0  
      then acc  
      else pomocna (x, y-1, acc*x)  
    in  
      pomocna (x,y,1)  
    end
```

izvedba programa:



Repna rekurzija

- repna rekurzija:
 - po izvedbi rekurzivnega klica v repu funkcije, ni potrebno izvesti več nobenih dodatnih operacij (množenje, seštevanje, ...)
 - rep funkcije definiramo rekurzivno:
 - v izrazu `fun f p = e` je telo `e` v repu
 - v izrazu `if e1 then e2 else e3` sta `e2` in `e3` v repu
 - v izraz `let b1 ... bn in e end` je `e` v repu
- pri repni rekurziji programski jeziki optimizirajo izvajanje:
 - namesto hranjenja okvirja ga zamenjajo z okvirjem klicane funkcije
 - kličoča in klicana funkcija uporabljata isti prostor na skladu
- po učinkovitosti enakovredno zankam
- prevedba v repno rekurzijo ni vedno možna (obdelava dreves?)
- torej, dejanska izvedba funkcije z repno rekurzijo:

potenca(2,3)

pomozna(2,3,1)

pomozna(2,2,2)

pomozna(2,1,4)

pomozna(2,0,8)

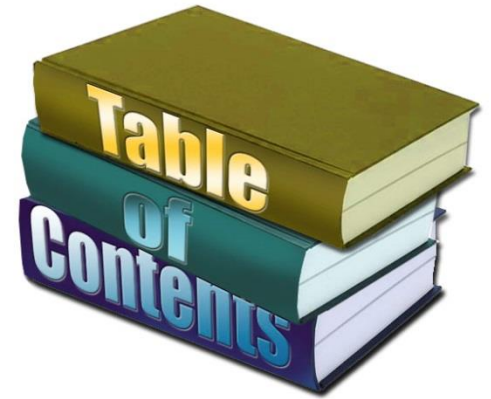
Primeri

Napiši naslednje funkcije, ki uporabljajo repno rekurzijo:

1. Funkcijo, ki obrne elemente v seznamu
2. Funkcijo, ki prešteje število pozitivnih elementov v seznamu
3. Funkcijo, ki sešteje elemente v seznamu

Pregled

- rekurzivno ujemanje vzorcev
- sklepanje na podatkovni tip
- izjeme
- repna rekurzija
- funkcije višjega reda
- map in filter



Funkcije višjega reda

- funkcije so **prvo-razredni objekti** (to pomeni: tudi funkcije so vrednosti, s katerimi lahko delamo enako kot z drugimi preprostimi vrednostmi)
- koristno za ločeno programiranje pogostih operacij, ki jih uporabimo kot zunanjo funkcijo
- funkcijam, ki sprejemajo ali vračajo funkcije, pravimo **funkcije višjega reda** (angl. *higher-order functions*)
- funkcije imajo **funkcijsko ovojnico** (angl. *function closure*) – struktura, v kateri hranijo kontekst, v katerem so bile definirane (vrednosti spremenljivk izven kličoče funkcije!)

```
fun operacija1 x = x*x*x
fun operacija2 x = x + 1
fun operacija3 x = ~x
```

```
fun izvedi (pod, funkcija) =
    funkcija (pod+100)
```

```
- izvedi (2, operacija1);
val it = 1061208 : int
- izvedi (2, operacija2);
val it = 103 : int
- izvedi2 (2, operacija3);
val it = ~102 : int
```

Funkcije kot argumenti funkcij

- funkcije so lahko argumenti drugih funkcij → bolj splošna programska koda

```
fun nkrat (f, x, n) =  
  if n=0  
  then x  
  else f(x, nkrat(f, x, n-1))  
  
fun pomnozi(x,y) = x*y  
fun seštej(x,y) = x+y  
fun rep(x,y) = tl y  
  
fun zmnozi_nkrat_kratka (x,n) = nkrat(pomnozi, x, n)  
fun seštej_nkrat_kratka (x,n) = nkrat(seštej, x, n)  
fun rep_nti_kratka (x,n) = nkrat(rep, x, n)
```

- funkcije višjega reda so lahko polimorfne (večja splošnost)
val nkrat = **fn** : ('a * 'a -> 'a) * 'a * **int** -> 'a

Funkcije, ki vračajo funkcije

- funkcije so lahko rezultat drugih funkcij
- primer

```
fun odloci x =  
  if x>10  
  then (let fun prva x = 2*x in prva end)  
  else (let fun druga x = x div 2 in druga end)
```

```
- odloci 12;  
val it = fn : int -> int  
- (odloci 12) 10;  
val it = 20 : int  
- (odloci 2) 20;  
val it = 10 : int
```

- tip funkcije odloči je **fn : int -> int -> int**
 - pri izpisu velja desna asociativnost, torej pomeni
fn : int -> (int -> int)

Anonimne funkcije

- namesto ločenih deklaracij funkcij (`fun`), lahko funkcije deklariramo na mestu, kjer jih potrebujemo (brez imenovanja – anonimno)
- sintaksa predstavlja izraz in ne deklaracijo (`fn` namesto `fun` in `=>` namesto `=`):

```
fn arg => telo
```

- primer uporabe: pri podajanju argumenta funkcijam višjega reda
- funkcija je lokalna, imena dejansko ne potrebujemo

```
fun zmnozi_nkrat (x,n) =  
  nkrat(let fun pomnozi (x,y) = x*y in pomnozi end , x, n)
```



enakovredno, lepši zapis

```
fun zmnozi_nkrat (x,n) =  
  nkrat(fn (x,y) => x*y, x, n)
```

- anonimnih funkcij ne moremo definirati rekurzivno - zakaj?

Funkcija *Map*

- preslika seznam v drugi seznam tako, da na vsakem elementu uporabi preslikavo *f* (ciljni seznam ima torej enako število elementov)

```
fun map (f, sez) =  
  case sez of  
    [] => []  
  | glava::rep => (f glava)::map(f, rep)
```

- podatkovni tip funkcije map

```
val map = fn : ('a -> 'b) * 'a list -> 'b list
```

- primer:

```
- map (fn x => Int.toString(2*x)^"a", [1,2,3,4,5,6,7]);  
val it = ["2a", "4a", "6a", "8a", "10a", "12a", "14a"] : string list
```

Funkcija *Filter*

- preslika seznam v drugi seznam tako, da v novem seznamu ohrani samo tiste elemente, za katere je predikat (funkcija, ki vrača bool) resničen

```
fun filter (f, sez) =  
  case sez of  
    [] => []  
  | glava::rep => if (f glava)  
                   then glava::filter(f, rep)  
                   else filter(f, rep)
```

- podatkovni tip funkcije filter

```
val filter = fn : ('a -> bool) * 'a list -> 'a list
```

- primer:

```
- filter(fn x => x mod 3=0, [1,2,3,4,5,6,7,8,9,10]);  
val it = [3,6,9] : int list
```


Primeri

Z uporabo map in filter:

1. preslikaj seznam seznamov v seznam glav vgnezdenih seznamov
- `nal1 [[1,2,3],[5,23],[33,42],[1,2,5,6,3]];`
`val it = [1,5,33,1] : int list`
2. preslikaj seznam seznamov v seznam dolžin vgnezdenih seznamov
- `nal2 [[1,2,3],[5,23],[33,42],[1,2,5,6,3]];`
`val it = [3,2,2,5] : int list`
3. preslikaj seznam seznamov v seznam samo tistih seznamov, katerih dolžina je daljša od 2
- `nal3 [[1,2],[5],[33,42],[1,2,5,6,3]];`
`val it = [[1,2],[33,42],[1,2,5,6,3]] : int list list`
4. preslikaj seznam seznamov v seznam vsot samo lihih elementov vgnezdenih seznamov
- `nal4 [[1,2,3],[5,23],[33,42],[1,2,5,6,3]];`
`val it = [4,28,33,9] : int list`



**Ovojnice, leksikalni dosegi,
currying**