FUNKCIJSKO PROGRAMIRANJE

2023/24

repna rekurzija

funkcije višjega reda

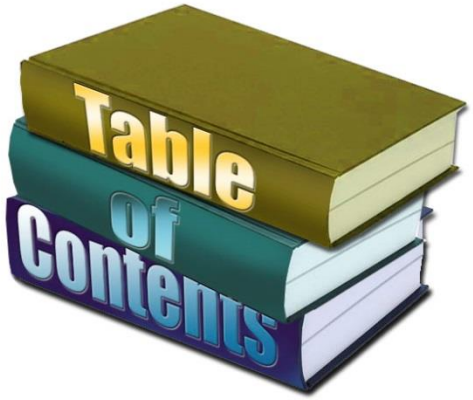
currying, delna aplikacija

mutacija

vzajemna rekurzija

moduli

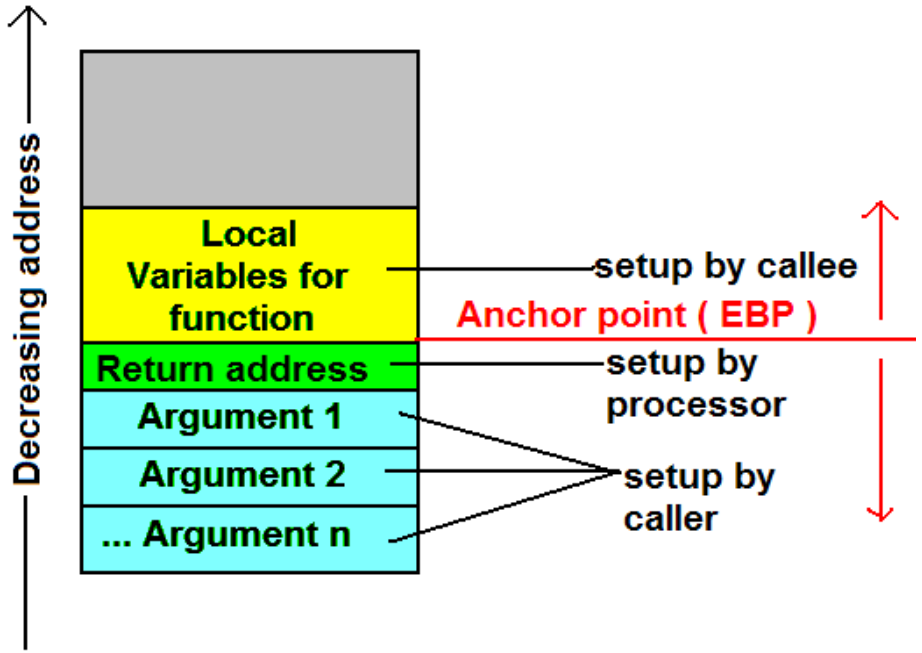
Pregled

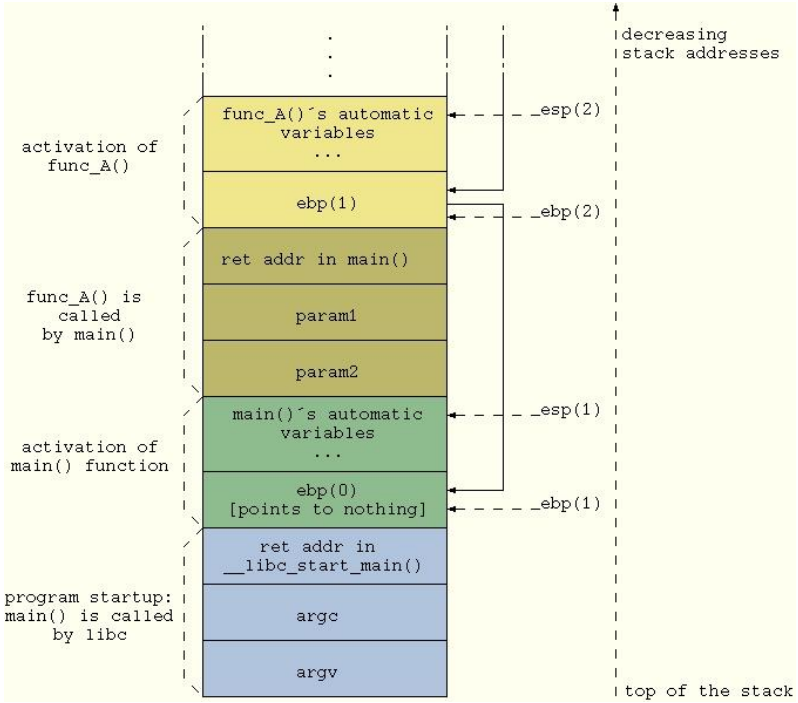


- repna rekurzija
- funkcije višjega reda
  - funkcije kot argumenti funkcij
  - funkcije, ki vračajo funkcije
  - map/filter/fold
- doseg vrednosti
- currying, delna aplikacija
- mutacija
- določanje podatkovnih tipov
- vzajemna rekurzija
- moduli

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 kličoča funkcija konteksta ne potrebuje več

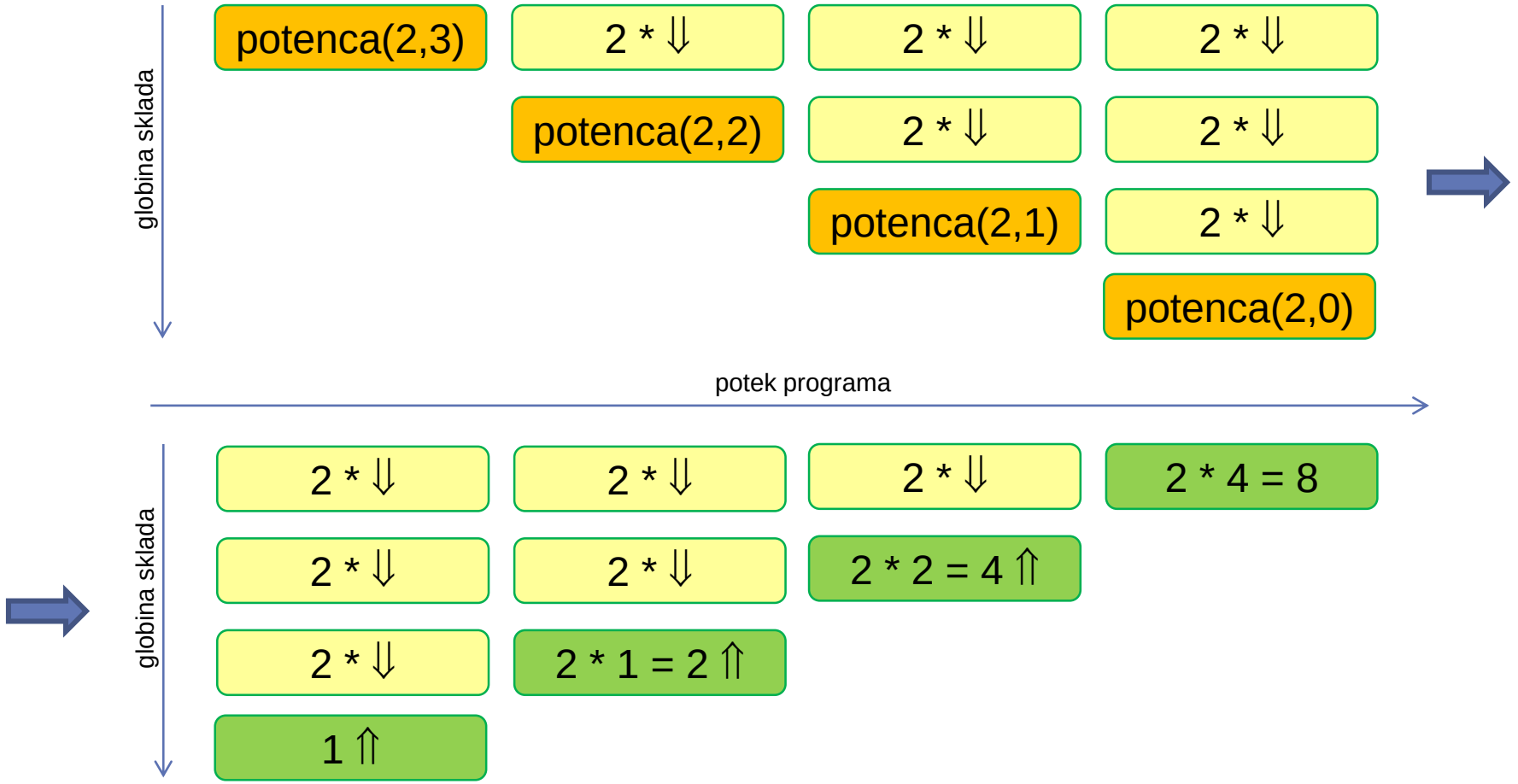




Izvedba rekurzije

Primer "navadne" rekurzije:

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

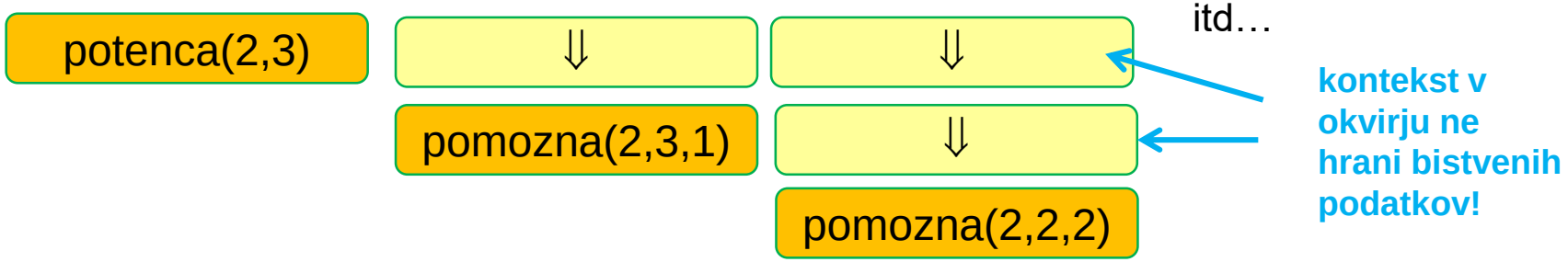


Drugačna implementacija

- 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 pomozna (x,y,acc) =
            if y=0
            then acc
            else pomozna(x, y-1, acc*x)
    in
        pomozna(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:

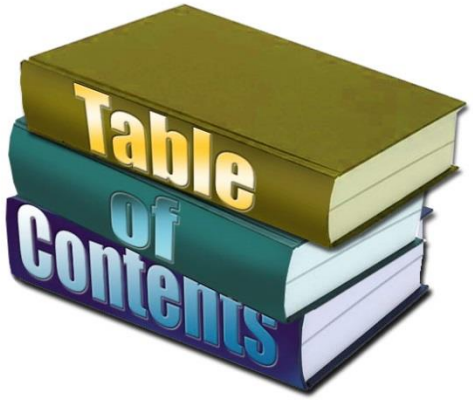
potenca(2,3) | pomozna(2,3,1) | pomozna(2,2,2) | pomozna(2,1,4) | pomozna(2,0,8)

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



- repna rekurzija
- funkcije višjega reda
  - funkcije kot argumenti funkcij
  - funkcije, ki vračajo funkcije
  - map/filter/fold
- doseg vrednosti
- currying, delna aplikacija
- mutacija
- določanje podatkovnih tipov
- vzajemna rekurzija
- moduli

Funkcije višjega reda

- tudi funkcije so **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 sestej(x,y) = x+y
fun rep(x,y) = tl y

fun zmnozi_nkrat_kratka (x,n) = nkrat(pomnozi, x, n)
fun sestej_nkrat_kratka (x,n) = nkrat(sestej, 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?

Funkcije višjega reda

- funkcije ki sprejemajo/vračajo funkcije
- refaktorizacija kode

```
fun nkrat (f, x, n) =
    if n=0
    then x
    else f(x, nkrat(f, x, n-1))

fun zmnozi_nkrat_mega (x,n) = nkrat(fn (x,y) => x*y, x, n)
fun sestej_nkrat_mega (x,n) = nkrat(fn(x,y) => x+y, x, n)
fun rep_nti_mega (x,n) = nkrat(fn(_,x)=>tl x, x, n)
```

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

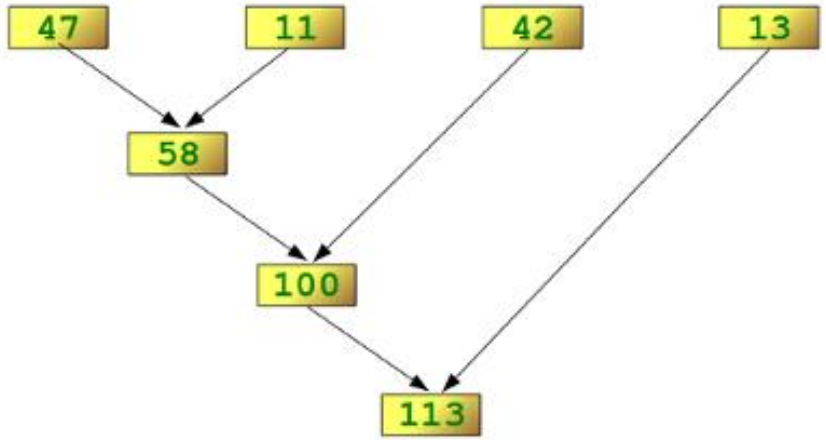
```
- nal4 [[1,2,3],[5,23],[33,42],[1,2,5,6,3]];
val it = [4,28,33,9] : int list
```

Funkcija *Fold*

- znana tudi pod imenom *reduce*
- združi elemente seznama v končni rezultat
- na elementih seznama izvede funkcijo f, ki upošteva trenutni rezultat in vrednost naslednjega elementa

`fold(f, acc, [a,b,c,d])` izračuna `f(d,f(c,f(b,f(a,acc))))`

- primer: seštevanje seznama



Funkcija *Fold*

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

- podatkovni tip funkcije fold

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

- primer:

```
(* seštej elemente v seznamu *)
- fold(fn (x,y) => x+y, 0, [1,2,3,4,5]);
val it = 15 : int

(* dolžina seznama *)
- fold(fn (x,y) => y+1, 0, [1,2,3,4,5]);
val it = 5 : int
```

Primeri

Uporabi map/filter/fold za zapis naslednjih funkcij:

1. Seštej elemente v celoštevilskem seznamu.
   ```
   fn : int list -> int
   ```

2. Preštej število elementov v seznamu.
   ```
   fn : 'a list -> int
   ```

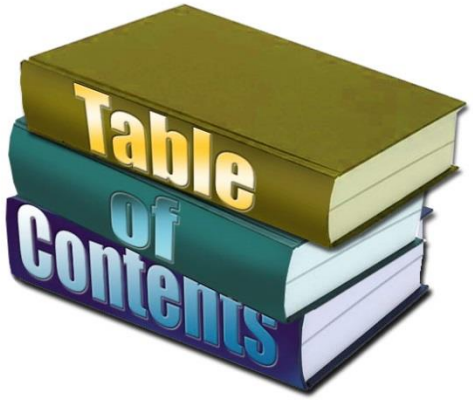
3. Vrni zadnji element v seznamu.
   ```
   fn : 'a list -> 'a
   ```

4. Izračunaj skalarni produkt dveh vektorjev.
   ```
   fn : int list list -> int
   ```

5. Vrni n-ti element v seznamu.
   ```
   fn : int list * int -> int
   ```

6. Obrni elemente v seznamu.
   ```
   fn : int list -> int list
   ```

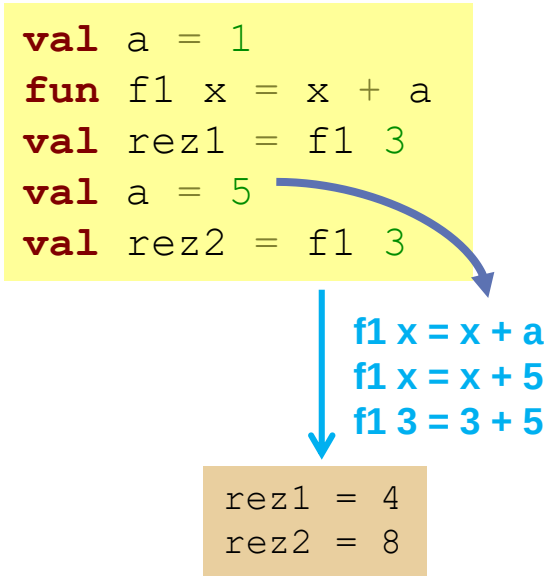
Pregled

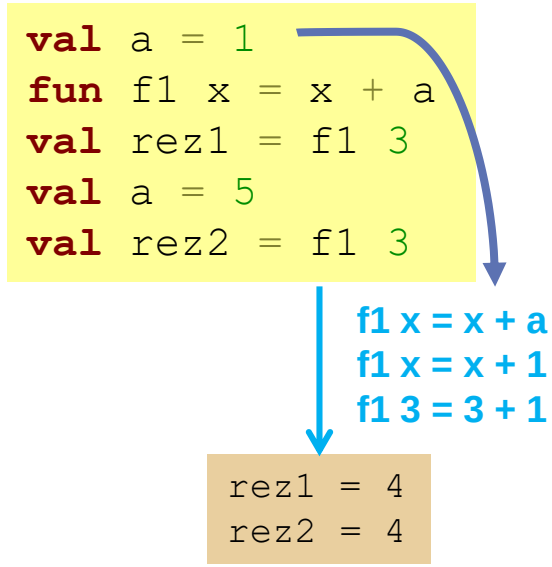


- repna rekurzija
- funkcije višjega reda
  - funkcije kot argumenti funkcij
  - funkcije, ki vračajo funkcije
  - map/filter/fold
- doseg vrednosti
- currying, delna aplikacija
- mutacija
- določanje podatkovnih tipov
- vzajemna rekurzija
- moduli

Doseg vrednosti

- funkcije kot prvo-razredni objekti so zmogljivo orodje
- definirati moramo semantiko pri določanju vrednosti spremenljivk v funkciji
- imamo dve možnosti:

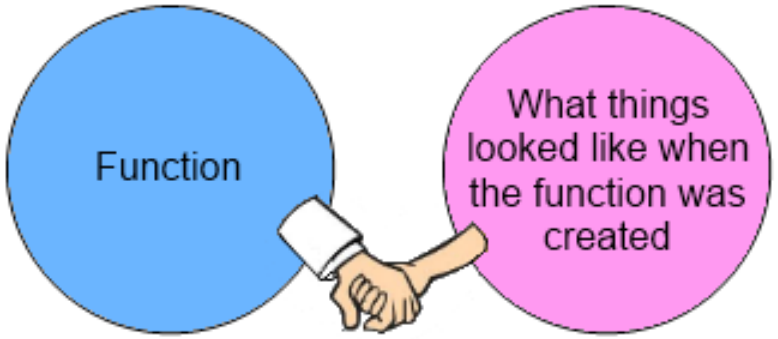




DINAMIČNI DOSEG (angl. *dynamic scope*): funkcija uporablja vrednosti spremenljivk v okolju, kjer jo kličemo

LEKSIKALNI DOSEG (angl. *lexical scope*): funkcija uporablja vrednosti spremenljivk v okolju, kjer je bila definirana

Funkcijska ovojnica



- angl. *function closure*
- pri deklaraciji funkcije torej ni dovolj, da shranimo le programsko kodo funkcije, temveč je potrebno shraniti tudi trenutno okolje
- **FUNKCIJSKA OVOJNICA** = koda funkcije + trenutno okolje
- klic funkcije = evalvacija kode `f` v okolju `env`, ki sta del funkcijske ovojnice `(f, env)`

Vaja

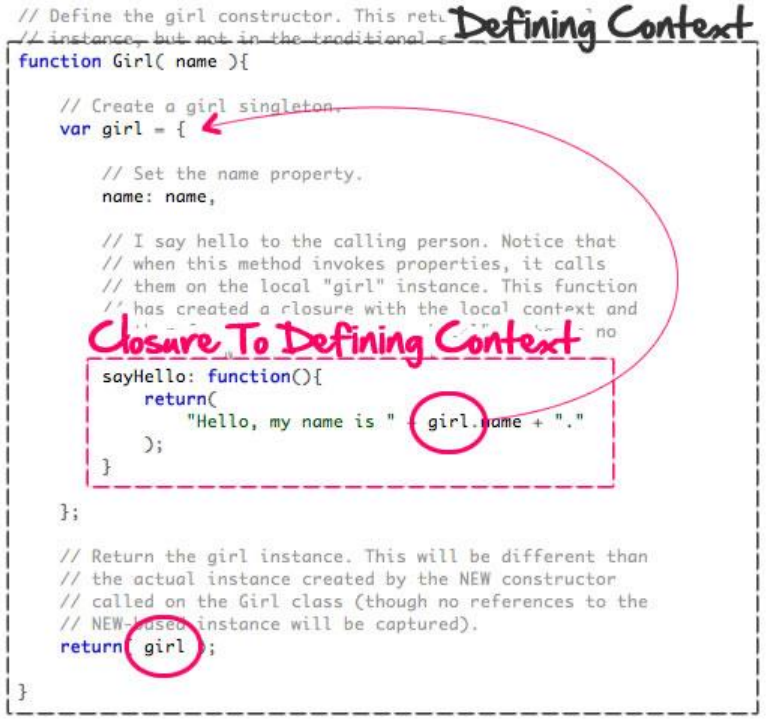
Kaj je rezultat naslednjih deklaracij, upoštevajoč leksikalni in dinamični doseg?

```
val u = 1
fun f v =
    let
        val u = v + 1
    in
        fn w => u + v + w
    end
val u = 3
val g = f 4
val v = 5
val w = g 6
```

leksikalni: 15
dinamični: ?

Leksikalni doseg

- funkcija uporablja vrednosti spremenljivk v okolju, **kjer je definirana**
- v zgodovini sta bili v programskih jezikih uporabljeni obe možnosti, danes prevladuje odločitev, da uporabljamo leksikalni doseg
- leksikalni doseg je bolj zmogljiv
  ➔ razlogi v nadaljevanju
- dinamičen doseg
  - pogost pri skriptnih jezikih (Lisp, bash, Logo, delno Perl)
  - včasih bolj primeren (proženje izjem, izpisovanje v statične datoteke, …)
  - nekateri sodobni jeziki imajo "posebne" spremenljivke, ki hranijo vrednosti v dinamičnem dosegu



Prednosti leksikalnega dosega

1. Imena spremenljivk v funkciji so neodvisna od imen zunanjih spremenljivk

```
fun fun1 y =
    let val x = 3
    in  fn z => x + y + z
    end
val a1 = (fun1 7) 4
val x = 42 (* nima vpliva *)
val a2 = (fun1 7) 4
```

2. Funkcija je neodvisna od imen uporabljenih spremenljivk

```
fun fun1 y =
    let
        val x = 3
    in
        fn z => x + y + z
    end
```

```
fun fun2 y =
    let
        val q = 3
    in
        fn z => q + y + z
    end
```

Prednosti leksikalnega dosega

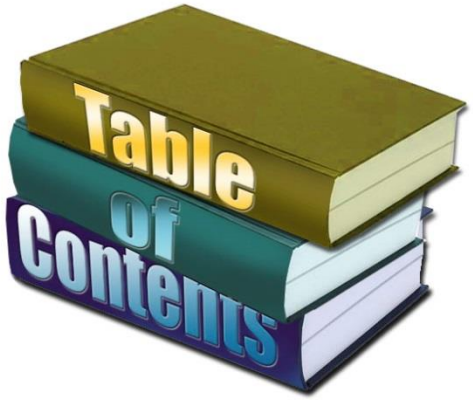
3. Tip funkcije lahko določimo ob njeni deklaraciji

```
val x = 1
fun fun3 y =
    let val x = 3
    in fn z => x + y + z end    (* int -> int -> int *)
val x = false   (* ne vpliva na tip funkcije ob izvedbi *)
val g = fun3 10
val z = g 11
```

4. Ovojnica shrani podatke, ki jih potrebuje za kasnejšo izvedbo.

```
fun filter (f, sez) =
    case sez of
        [] => []
      | x::rep => if (f x)
                  then x::filter(f, rep)
                  else filter(f, rep)
fun vecjiOdX x = fn y => y > x
fun brezNegativnih sez = filter(vecjiOdX ~1, sez)
```

Pregled



- repna rekurzija
- funkcije višjega reda
  - funkcije kot argumenti funkcij
  - funkcije, ki vračajo funkcije
  - map/filter/fold
- doseg vrednosti
- currying, delna aplikacija
- mutacija
- določanje podatkovnih tipov
- vzajemna rekurzija
- moduli

Currying

- *Currying* – ime metode, naziv dobila po matematiku z imenom Haskell Curry
- spomnimo se: funkcije sprejemajo natanko en argument
  - če želimo podati več vrednosti v argumentu, smo jih običajno zapisali v terko
- alternativna možnost: če imamo več argumentov, naj funkcija sprejme samo en argument in vrne funkcijo, ki sprejme preostanek argumentov (nadaljevanje na enak način)

f: A × B × C -> D	*non curried.*
f: A -> B -> C -> D	*curried.*

f:	A -> (B -> (C -> D))
f a:	B -> (C -> D)
f a b:	C -> D

f:	A -> B -> C -> D
f a:	B -> C -> D
f a b:	C -> D

Currying

- "stari način": funkcija, ki sprejema terko argumentov

```
fun vmejah_terka (min, max, sez) =
    filter(fn x => x>=min andalso x<=max, sez)
```

- ***currying***: funkcija, ki vrača funkcijo…

```
fun vmejah_curry min =
    fn max =>
       fn sez =>
          filter(fn x => x>=min andalso x<=max, sez)
```

- klici:

```
- vmejah_terka (5, 15, [1,5,3,43,12,3,4]);

- (((vmejah_curry 5) 15) [1,5,3,43,12,3,4]);
```

Currying: sintaktične olepšave

- deklaracijo funkcije

```
fun vmejah_curry min =
    fn max =>
       fn sez =>
          filter(fn x => x>=min andalso x<=max, sez)
```

lahko lepše zapišemo s presledki med argumenti

```
fun vmejah_lepse min max sez =
    filter(fn x => x>=min andalso x<=max, sez)
```

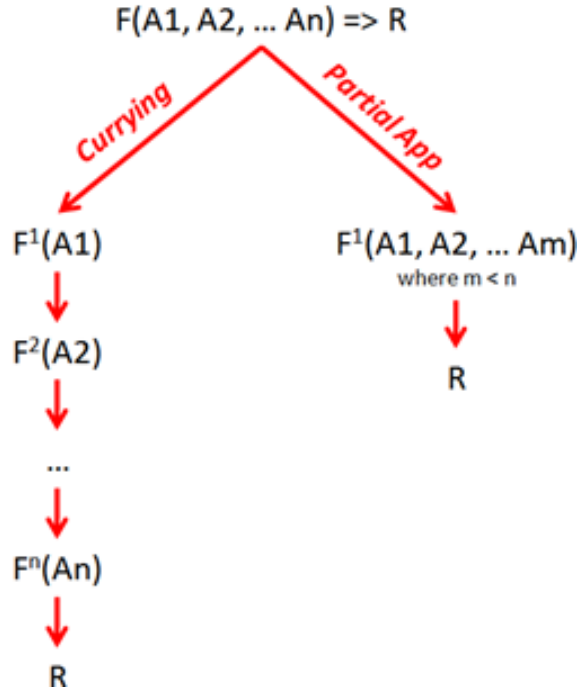
- klic

```
- (((vmejah_curry 5) 15) [1,5,3,43,12,3,4]);
```

lahko lepše zapišemo brez oklepajev

```
- vmejah_curry 5 15 [1,5,3,43,12,3,4];
```

Delna aplikacija funkcij



- ko uporabljamo *currying*, lahko pri klicu funkcije podamo manj argumentov, kot jih funkcija ima
- rezultat: *delna aplikacija* funkcije oz. funkcija, ki "čaka" na preostale argumente
- prednost: klic lahko posplošimo v drugo funkcijo

- sintaksa: spomnimo se, da lahko zapišemo

```
val f = g
```

če sta f in g funkciji; ta zapis je enakovreden (sintaktično slabše):

```
fun f x = g x
```

- primer:

```
val prva_desetica = vmejah_curry 1 10;
(* vrne samo števila od 1 do 10 *)

- prva_desetica [1,14,3,23,4,23,12,4];
val it = [1,3,4,4] : int list
```

Delna aplikacija funkcij

- zato, da lahko izvajamo delno aplikacijo, vgrajene funkcije `List.map`, `List.filter` in `List.fold` uporabljajo *currying*

```
(* poveča vse elemente v seznamu za 1 *)
val povecaj = List.map (fn x => x + 1);
```

- v SML/NJ je bolj učinkovita uporaba funkcij s terkami kot če uporabljamo *currying*. Zakaj?
  - slednje ne velja nujno tudi za druge programske jezike (optimizacija kode v prevajalniku)

- (pomoč v REPL glede argumentov funkcij)

```
- structure X = ListPair;    (* povprašamo po nazivu knjižnice *)
structure X : LIST_PAIR

-  signature X = LIST_PAIR;   (* zahtevamo izpis povzetka *)
```

Prevedba med zapisi funkcij

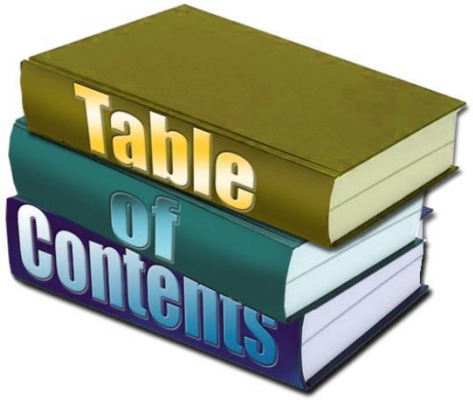
- zapis s terko ⇔ currying

```
fun curry f x y = f (x,y)
fun uncurry f (x,y) = f x y
```

- zamenjava vrstnega reda argumentov

```
fun zamenjaj f x y = f y x
```

Pregled

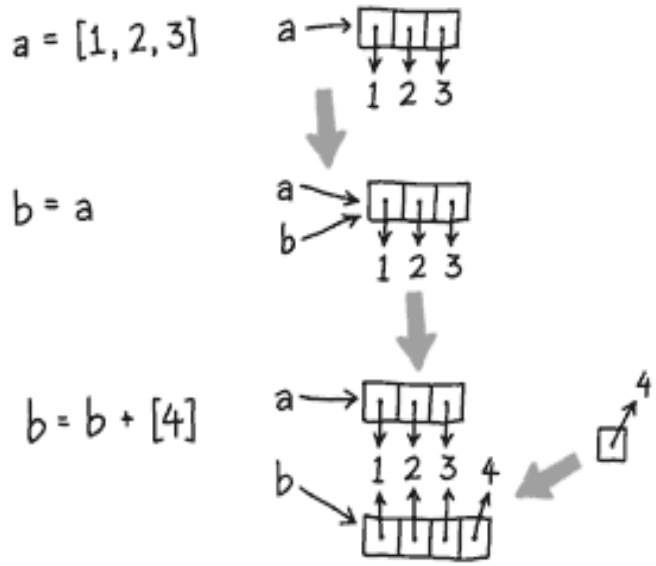


- repna rekurzija
- funkcije višjega reda
  - funkcije kot argumenti funkcij
  - funkcije, ki vračajo funkcije
  - map/filter/fold
- doseg vrednosti
- currying, delna aplikacija
- mutacija
- določanje podatkovnih tipov
- vzajemna rekurzija
- moduli

Mutacija vrednosti

- kot prednost funkcijskega programiranja smo omenili izogibanje "stranskim učinkom" programa, kot je spreminjanje vrednosti spremenljivkam

> ***Wiki (side effects):***
> *In the presence of side effects, a program's behavior depends on history; that is, the* ***order of evaluation*** *matters. Understanding and debugging a function with side effects* ***requires knowledge about the context*** *and its possible histories.*



- kje tiči prednost v tem?
  - preprosto **ponovljivo testiranje** funkcij (neodvisne od konteksta)
  - **neodvisnost naše kode** od implementacije algoritmov in podatkovnih struktur

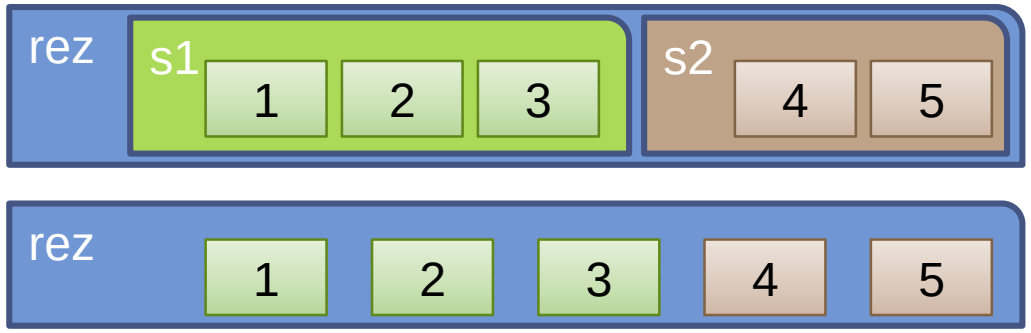
Neodvisnost od implementacije

- primer: funkcija za združevanje dveh seznamov

```
(* združi seznama sez1 in sez2 v skupni seznam *)
fun zdruzi_sez sez1 sez2 =
    case sez1 of
         [] => sez2
       | g::rep => g::(zdruzi_sez rep sez2)

val s1 = [1,2,3]
val s2 = [4,5]
val rezultat = zdruzi_sez s1 s2
```

- rešitev zadnjega klica je (očitno) seznam `[1,2,3,4,5]`, vendar pa: ali je združevanje uporablja referenci na `s1` in `s2` ali kopira elemente?
  - referenca
  - kopija



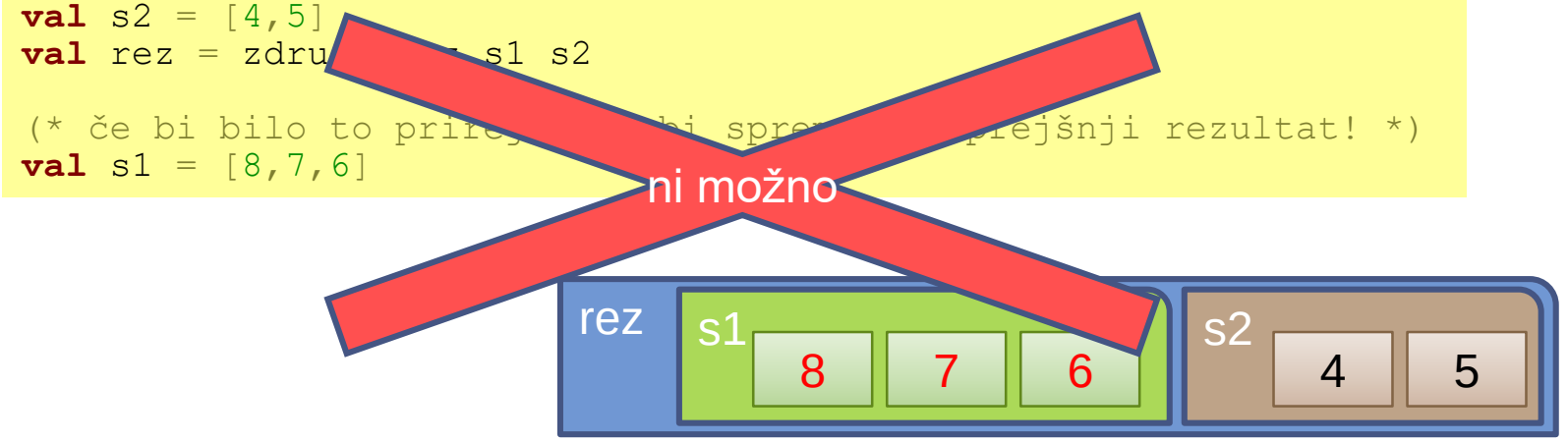
- *ali je to sploh pomembno?*

Neodvisnost od implementacije

- SML sicer uporablja reference (varčevanje s prostorom), vendar to ni pomembno, ker brez mutacij ne moremo povzročiti nepričakovanih rezultatov, kot je ta:

```
val s1 = [1,2,3]
val s2 = [4,5]
val rez = zdru    s1 s2

(* če bi bilo to prire    bi spre    prejšnji rezultat! *)
val s1 = [8,7,6]
```



- v jezikih z mutacijo je zgornje vir številnih nepredvidenih semantičnih napak (Java?)

- resnica:

SML tudi lahko uporablja mutacijo

Mutacija

- priročen pristop, kadar potrebujemo spremenljivo **globalno stanje** v programu
- za mutacijo vpeljemo novi podatkovni tip `t ref` (`t` je poljubni tip):

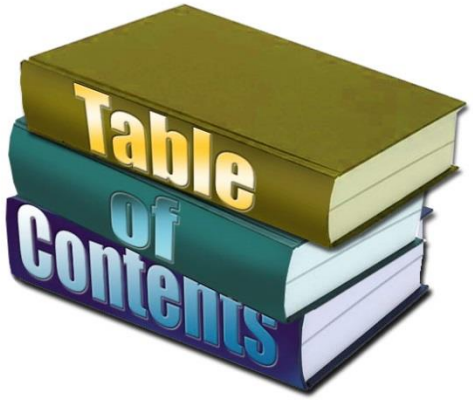
```
ref e     (* izdelava spremenljivke *)
e1 := e2  (* sprememba vsebine *)
!e        (* vrne vrednost *)
```

- primer:

```
- val x = ref 15;
val x = ref 15 : int ref
- val y = ref 2;
val y = ref 2 : int ref
- (!x)+(!y);
val it = 17 : int
- x:=7;
val it = () : unit
- (!x)+(!y);
val it = 9 : int
```

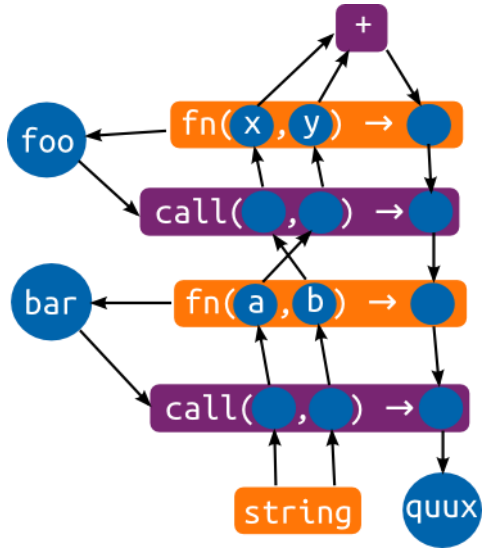
- mutacije ne uporabljamo, razen če ni nujno potrebno: povzročajo stranske učinke in težave pri določanju podatkovnih tipov! (→ *kasneje več o tem*)

Pregled



- repna rekurzija
- funkcije višjega reda
  - funkcije kot argumenti funkcij
  - funkcije, ki vračajo funkcije
  - map/filter/fold
- doseg vrednosti
- currying, delna aplikacija
- mutacija
- določanje podatkovnih tipov
- vzajemna rekurzija
- moduli

Določanje podatkovnih tipov



- angl. *type inference*
- **cilj**: vsaki deklaraciji (zaporedoma) določiti tip, ki bo skladen s tipi preostalih deklaracij

- tipizacija glede na statičnost**:**
  - **statično tipizirani** jeziki (ML, Java, C++, C#): preverjajo pravilnost podatkovnih tipov in opozorijo na napake v programu **pred izvedbo**
  - **dinamično tipizirani jeziki** (Racket, Python, JavaScript, Ruby): izvajajo manj (ali nič) preverb pravilnosti podatkovnih tipov, večino preverjanj se izvede **pri izvajanju**

- tipizacija glede na implicitnost**:**
  - **implicitno tipiziran** jezik (ML): podatkovnih tipov nam ni potrebno eksplicitno zapisati (kdaj smo jih že morali pisati?)
  - **eksplicitno tipiziran** jezik (Java, C++, C#): potreben ekspliciten zapis tipov

Ker je ML implicitno tipiziran jezik, ima vgrajen mehanizem za samodejno določanje podatkovnih tipov.

Postopek

- postopek določanja podatkovnega tipa za vsako deklaracijo:

1. Za deklaracijo (`val` ali `fun`) naredi seznam omejitev.
2. Analiziraj omejitve in določi tipe.
3. Rezultat:
   a) če so omejitve v **protislovju** → vrni napako
   b) če iz **presplošnih** omejitev ni možno določiti konkretnega tipa → uporabi zanje spremenljivko (za polimorfizem: `'a`, `'b`, …)
   c) uporabi **omejitev vrednosti** (angl. *value restriction*) (o tem kasneje)

- primer

```
fun f (q, w, e) =              (* 1.  f: 'a * 'b * 'c -> 'd *)
                               (* 3.  f: ('f * 'g) list * 'b * 'c -> 'd *)
                               (* 5.  f: ('f * 'g) list * bool list * 'c -> 'd *)
                               (* 8.  f: ('f * int) list * bool list * 'c -> int *)
    let val (x,y) = hd(q)      (* 2.  'a = 'e list;  'e = ('f * 'g);  'a = ('f * 'g) list *)
    in if hd w                 (* 4.  'b = 'h list;  'h = bool;  'b = bool list *)
       then y mod 2            (* 6.  y: int; 'd = int *)
       else y*y                (* 7.  skladno s 6 velja y: int; 'd = int *)
    end
```

Premislek...

- če programski jezik izvaja določanje podatkovnega tipa, lahko uporablja spremenljivke tipov (`'a`, `'b`, `'c`, …) ali pa tudi ne
  - kakšna je prednost, če uporablja?
- vendar pa: kombinacija polimorfizma in mutacije lahko prinese težave pri določanju tipov, če bi pomenila spremembo določenega podatkovnega tipa
  - legalen primer (brez polimorfizma):

```
- val sez = ref [1,2,3];
val sez = ref [1,2,3] : int list ref
- sez := (!sez) @ [4,5];
- !sez;
val it = [1,2,3,4,5] : int list
```

  - problematičen primer (uporablja polimorfen tip):

```
- val sez = ref [];      (* sez je tipa 'a list ref *)
- sez := !sez @ [5];     (* v seznam dodamo int *)
- sez := !sez @ [true];  (* pokvari pravilnost tipa seznama! *)
```

- rešitev: spremenljivka ima lahko polimorfen tip samo, če je na desni strani deklaracije vrednost (konstanta), spremenljivka ali nepolimorfna funkcija. To imenujemo ***omejitev vrednosti***.
  - *ref* ni vrednost/spremenljivka, ampak funkcija (konstruktor)

Omejitev vrednosti

- deklaracije spremenljivk polimorfnih tipov dopustimo le, če je na desni strani vrednost (konstanta), spremenljivka ali nepolimorfna funkcija
- odgovor ML:
  - ML določi spremenljivkam neveljaven tip (dummy type), ki ga ne moremo uporabljati za funkcijske klice

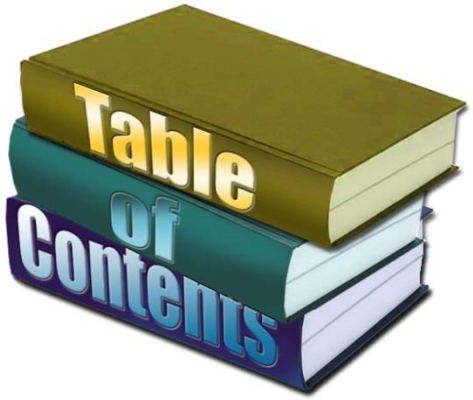
```
- val sez = ref [];
stdIn:10.5-10.17 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val sez = ref [] : ?.X1 list ref
```

- dve možni rešitvi:
  1. ročna opredelitev podatkovnih tipov
  2. ovijanje deklaracije vrednosti v deklaracijo funkcije (za njih ne velja omejitev vrednosti)

```
- val mojaf1 = map (fn x => 1);
stdIn:11.5-11.17 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
- mojaf1 [1,2,3];
stdIn:18.1-18.15 Error: operator and operand don't agree [literal]
  operator domain: ?.X1 list
  operand:         int list
```

```
- fun mojaf2 sez = map (fn x => 1) sez;
val mojaf2 = fn : 'a list -> int list
- mojaf2 [1,2,3];
val it = [1,1,1] : int list
```

Pregled



- repna rekurzija
- funkcije višjega reda
  - funkcije kot argumenti funkcij
  - funkcije, ki vračajo funkcije
  - map/filter/fold
- doseg vrednosti
- currying, delna aplikacija
- mutacija
- določanje podatkovnih tipov
- vzajemna rekurzija
- moduli

Vzajemna rekurzija

- omogočati uporabo funkcij in podatkovnih tipov, ki so deklarirani za trenutno deklaracijo

```
fun fun1 par1 = <telo>
and fun2 par2 = <telo>
and fun3 par3 = <telo>
```

```
datatype tip1 = <definicija>
and tip2 = <definicija>
and tip3 = <definicija>
```

- primer:

```
fun sodo x =
    if x=0
    then true
    else liho (x-1)
and liho x =
    if x=0
    then false
    else sodo (x-1)
```

- v praksi uporabno za opisovanje stanj končnih avtomatov

Vzajemna rekurzija

- izpitna naloga 2013/14

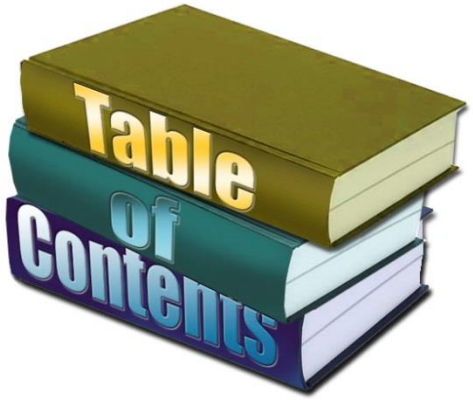
V jeziku SML napiši program check, ki preverja pravilnost vhodnega seznama `sez`. Za vhodni seznam morajo veljati naslednja pravila:

- program naj sprejme prazen seznam,
- seznam hrani vrednosti podatkovnega tipa `datatype datum = A of int | B of int list`
- v seznamu se izmenjujeta podatka, narejena s konstruktorjem `A` in konstruktorjem `B`,
- seznam se mora obvezno začeti z elementom, ki je narejen s konstruktorjem `A` in se lahko konča s poljubnim elementom (konstruktor `A` ali `B`),
- seznami tipa `int list`, ki so argument konstruktorja `B`, vsebujejo elemente z vrednostima 3 in 4,
- seznami tipa `int list`, ki so argument konstruktorja `B`, se morajo vedno končati na 4, njihov začetek pa ni pomemben.

Primeri/:

```
- check [A 1, B [3,4], A 3];
val it = true : bool
- check [A 9, B [3,4], A 4, B [4,3,4,3,4], A 2, B [4]];
val it = true : bool
- check [B [3,4], A 1, B [4,3]];
val it = false : bool        (* has to start with A *)
- check [A 1, B [3,4,3]];
val it = false : bool        (* list given with B has to end with 4 *)
```

Pregled



- repna rekurzija
- funkcije višjega reda
  - funkcije kot argumenti funkcij
  - funkcije, ki vračajo funkcije
  - map/filter/fold
- doseg vrednosti
- currying, delna aplikacija
- mutacija
- določanje podatkovnih tipov
- vzajemna rekurzija
- moduli

Moduli

- omogočajo:
  - organiziranje programske kode v smiselne celote
  - preprečevanje senčenja (isto ime je lahko deklarirano v več modulih)
- znotraj modula se sklicujemo na deklarirane objekte enako, kot smo se v prej v "zunanjem" okolju (brez posebnosti)
- iz "zunanjega" okolja se na deklaracije v modulu sklicujemo z uporabo predpone "*ImeModula.ime*"
- sintaksa za deklaracijo modula:

```
structure MyModule =
struct
   <deklaracije val, fun, datatype, …>
end
```

```
structure Nizi =
struct
  val prazni_niz = ""
  fun prvacrka niz =
      hd (String.explode niz)
end
```

Javno dostopne deklaracije

- modulu lahko določimo, katere deklaracije so na razpolago "javnosti" in katere so zasebne (*public* in *private* v Javi?)
- seznam javnih deklaracij strnemo v *podpis modula* (signature), nato podpis pripišemo modulu

```
signature PolinomP =
sig
    datatype polinom = Nicla | Pol of (int * int) list
    val novipolinom : int list -> polinom
    val mnozi : polinom -> int -> polinom
    val izpisi : polinom -> string
end
```

deklaracija podpisa

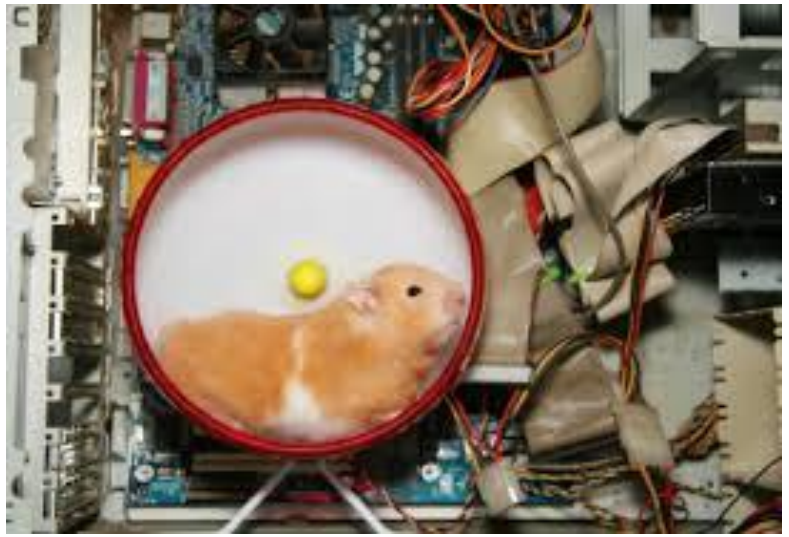
```
structure Polinom :> PolinomP =
struct
    ... deklaracije ...
end
```

podpis pripišemo modulu; uporabimo operator :>

- v podpisu določimo samo podatkovne tipe deklaracij (type, datatype, val, exception)
- podpis mora biti skladen z vsebino modula, sicer preverjanje tipov ne bo uspešno

Skrivanje implementacije

- uporaba podpisov modulov je koristna, ker z njim skrivamo implementacijo, kar je lastnost dobre in robustne programske opreme!
- s skrivanjem implementacije dosežemo:
  1. uporabnik ne pozna načina implementacije operacij; lahko jo tudi kasneje spremenimo brez vpliva na preostalo kodo,
  2. uporabniku onemogočimo, da uporablja modul na napačen način



Primer

Denimo, da specificiramo naslednje želje/zahteve glede uporabe:

- za izdelavo novega polinoma naj se uporablja funkcija `novipolinom`
- koeficienti polinoma so zapisani v padajočem vrstnem redu glede na potenco neodvisne spremenljivke
- vse potence neodvisne spremenljivke so pozitivne
- če je koeficient enak 0, ga ne hranimo
- želimo, da funkcija za množenje ni vidna navzven, je pa na razpolago (morebitnim) internim funkcijam

```
signature PolinomP =
sig
    datatype polinom = Nicla | Pol of (int * int) list
    val novipolinom : int list -> polinom
    val mnozi : polinom -> int -> polinom
    val izpisi : polinom -> string
end
```

če odstranimo funkcijo `mnozi` iz podpisa, ali potem ta podpis ustreza zgornji specifikaciji?

žal ne… poglejmo si primer

Skrivanje podrobnosti

- uporabnik lahko kvari delovanje, predvideno v specifikaciji (glej primer)

```
signature PolinomP =
sig
    datatype polinom = Nicla
         | Pol of (int * int) list
    val novipolinom : int list -> polinom
    val izpisi : polinom -> string
end
```

KORAK 1:
skrijemo funkcijo za množenje

```
signature PolinomP2 =
sig
    type polinom
    val novipolinom : int list -> polinom
    val izpisi : polinom -> string
end
```

KORAK 2:
definiramo abstraktni podatkovni tip, ki ne razkriva podrobnosti implementacije uporabniku:

- skrijemo, da je polinom datatype
- uporabnik še vedno lahko računa s polinomi

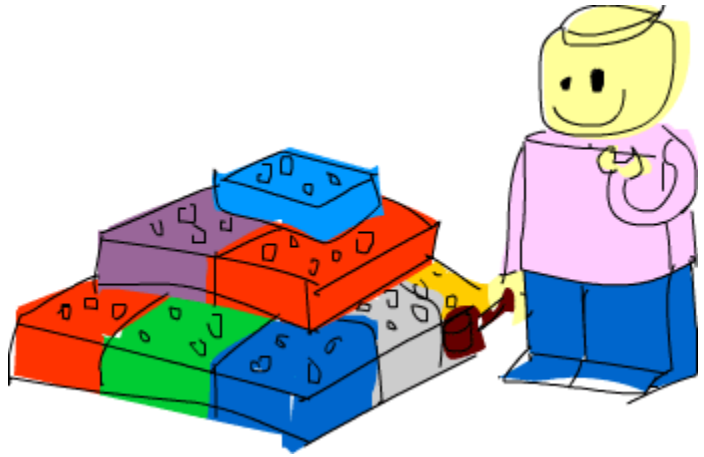
```
signature PolinomP3 =
sig
    type polinom
    val Nicla : polinom
    val novipolinom : int list -> polinom
    val izpisi : polinom -> string
end
```

KORAK 3:
vendar pa ni nič narobe, če razkrijemo samo del podatkovnega tipa (vrednost Nicla) in skrijemo samo konstruktor Pol

Ustreznost modula in podpisa

Podpis lahko uspešno pripišemo modulu (`Modul :> podpis`), če velja:

1. vsi ne-abstraktni tipi, ki smo jih navedli v podpisu, morajo biti implementirani v modulu (`datatype`)
2. vsi abstraktnih tipi iz podpisa (implementirani s `type`) so implementirani v modulu (s `type` ali `datatype`)
3. vsaka deklaracija vrednosti (`val`) v podpisu se nahaja v modulu
   (vendar pa je lahko v modulu bolj splošnega tipa)
4. vsaka izjema (`exception`) v podpisu se nahaja tudi v modulu




Racket!