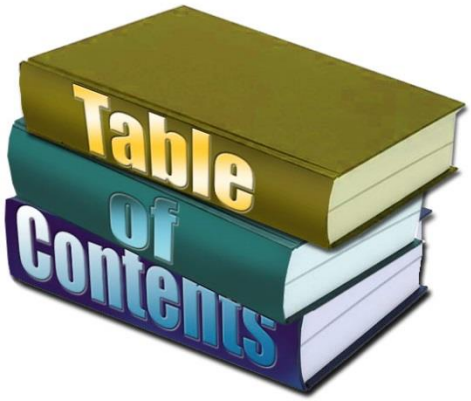
PROGRAMIRANJE
2014/15

funkcije višjega reda
leksikalni doseg
funkcijske ovojnice
delna aplikacija

Pregled



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

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 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?

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

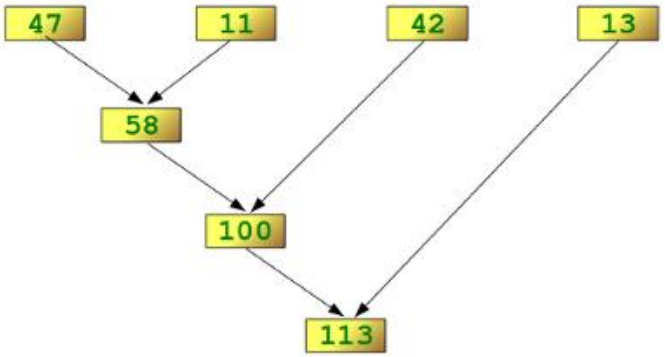
```
- nal4 [[1,2,3],[5,23],[33,42],[1,2,5,6,3]];
val it = [4,28,33,9] : int list
```

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(f(f(f(acc,a),b),c),d)`

- primer: seštevanje seznama



Funkcija *Fold*

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

- podatkovni tip funkcije fold

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

- 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) => x+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
   ```

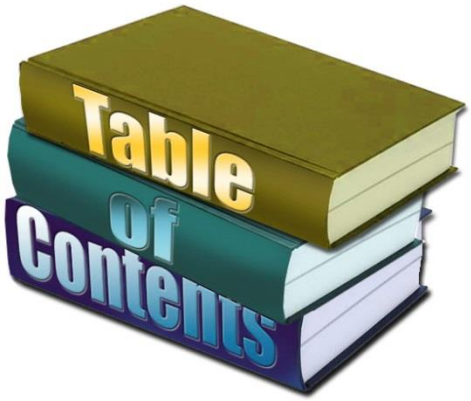
3. Vrni zadnji element v seznamu.
   ```
   fn : 'a list -> 'a
   ```

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
   ```

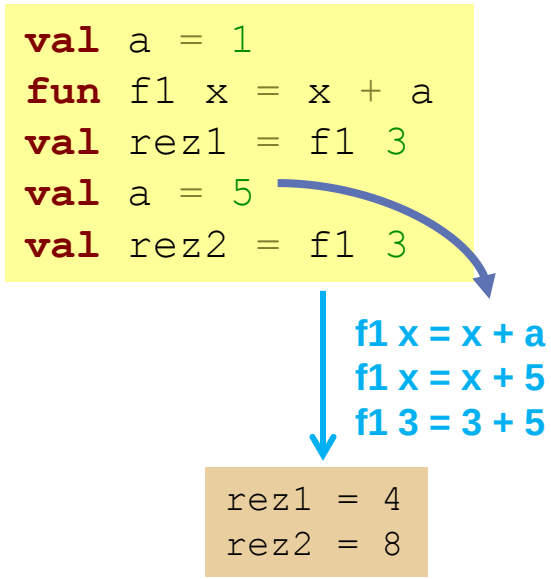
Pregled



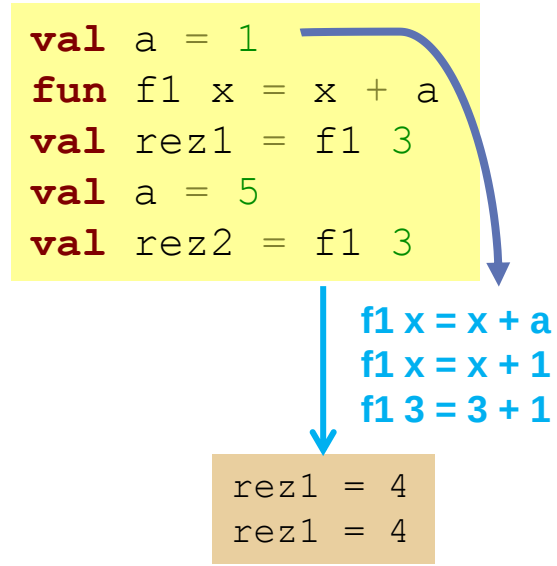
- map, filter, fold
- doseg vrednosti
- funkcijska ovojnica
- currying

Doseg vrednosti

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

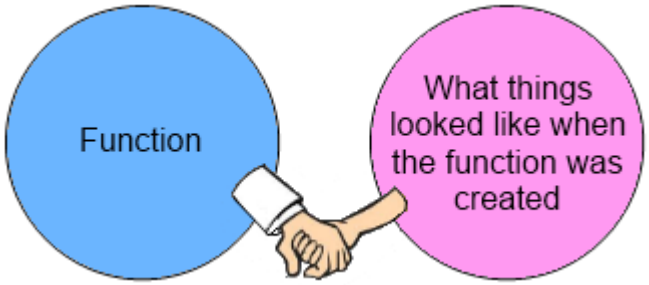


DINAMIČNI DOSEG (angl. *dynamic scope*): funkcija uporablja vrednosti spremenljivk v okolju, kjer jo kličemo



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

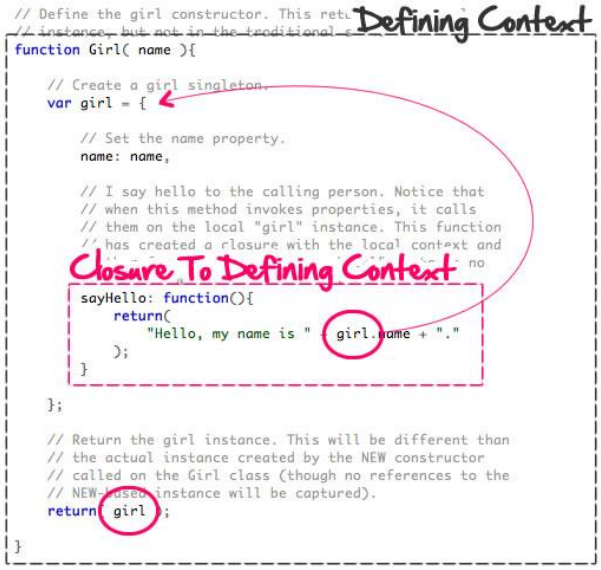
Kaj je rezultat naslednjih deklaracij, upoštevajoč leksikalni in dinamični doseg?

```
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

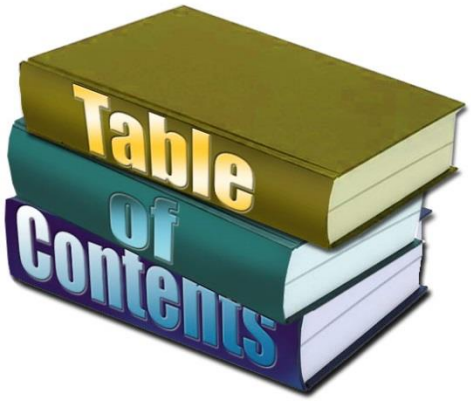
3. Tip funkcije lahko določimo ob njeni deklaraciji

```
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



- map, filter, fold
- doseg vrednosti
- funkcijska ovojnica
- currying
- delna aplikacija

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,
vzajemna rekurzija,
mutacija, moduli