PROGRAMIRANJE
2013/14

leksikalni doseg

currying, delna aplikacija

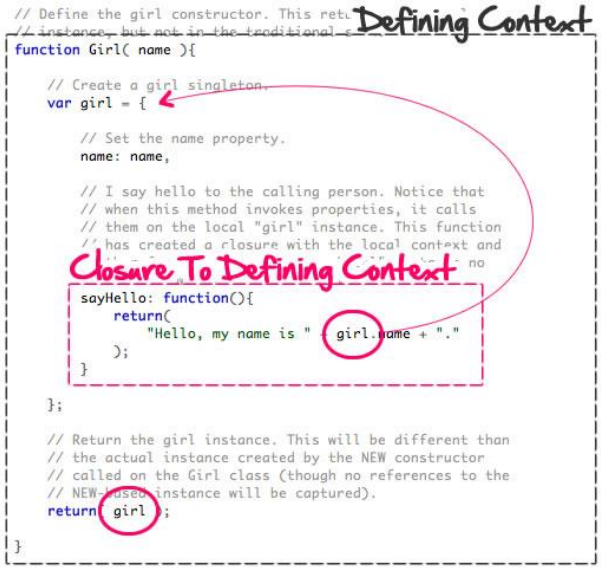
mutacija

določanje podatkovnih tipov

vzajemna rekurzija

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

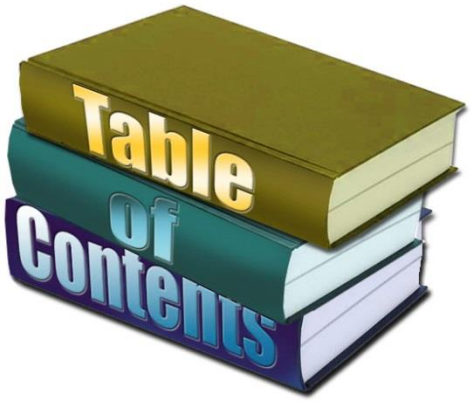
3. Tip funkcije lahko določimo ob njeni deklaraciji

```
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



- leksikalni doseg
- currying, delna aplikacija
- mutacija
- določanje podatkovnih tipov
- vzajemna rekurzija

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)
```

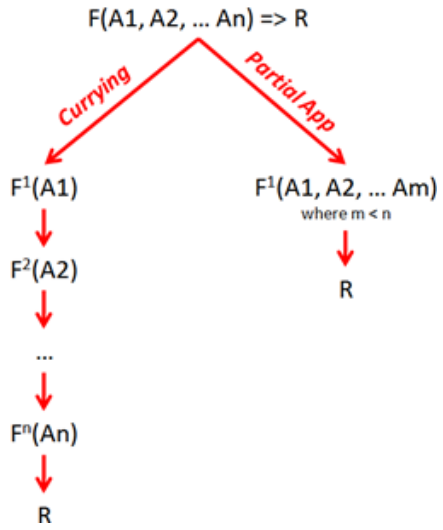
- klic

```
- (((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

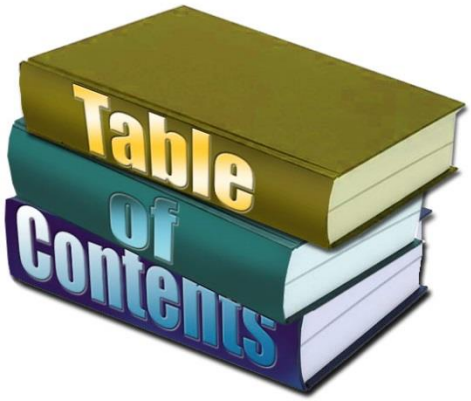
- zapis s terko ⇔ currying

```
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

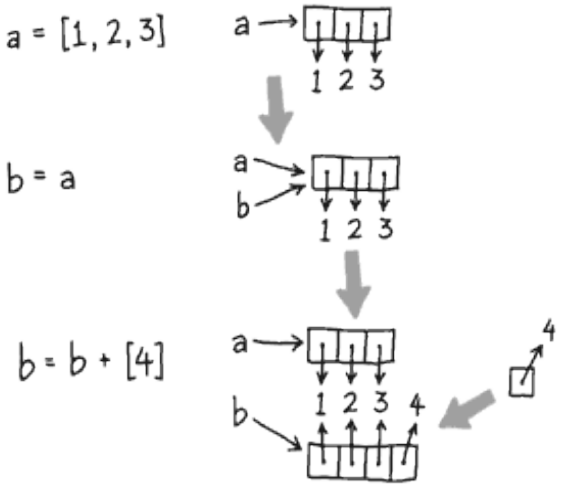


- leksikalni doseg
- currying, delna aplikacija
- mutacija
- določanje podatkovnih tipov
- vzajemna rekurzija

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

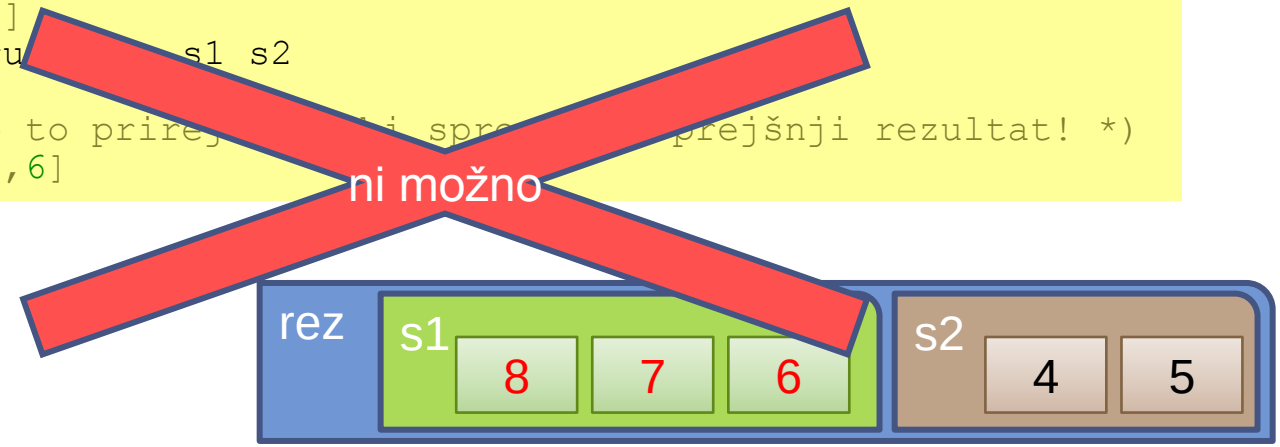
- *ali je to sploh pomembno?*

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          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):

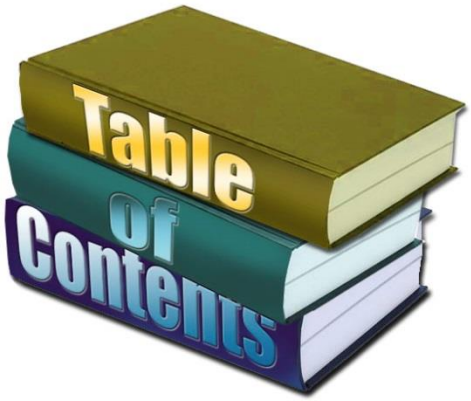
```
ref e     (* izdelava spremenljivke *)
e1 := e2  (* sprememba vsebine *)
!e        (* vrne vrednost *)
```

- 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*)

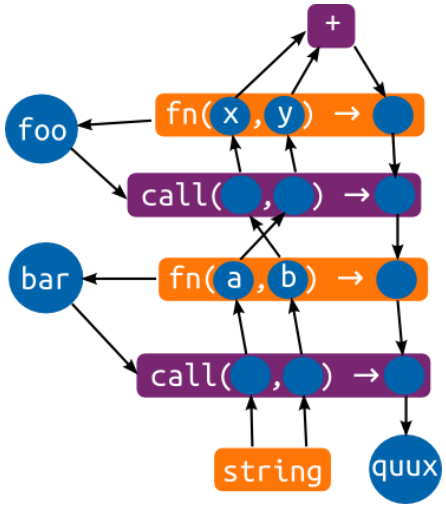
Pregled



- leksikalni doseg
- currying, delna aplikacija
- mutacija
- določanje podatkovnih tipov
- vzajemna rekurzija

Določanje podatkovnih tipov

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



- tipizacija glede na <u>statičnost</u>**:**
  - **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
- tipizacija glede na <u>implicitnost</u>**:**
  - **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 ne
  - kakšna je prednost, če uporablja?
- vendar pa: kombinacija polimorfizma in mutacije lahko prinese težave pri določanju tipov
  - 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];     (* seznam dodamo int *)
- sez := !sez @ [true];  (* pozabari pravilnost tipa seznama! *)
```

- rešitev: spremenljivka ima lahko polimorfen tip samo, če je na desni strani deklaracije vrednost ali spremenljivka. 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 ali spremenljivka
- odgovor ML:
  - ML določi spremenljivkam neveljaven tip (dummy type), ki ga ne moremo uporabljati za funkcijske klice

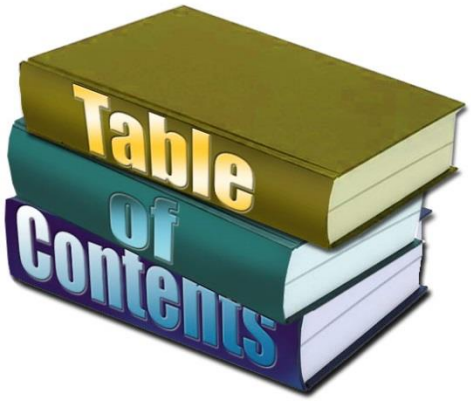
```
- val sez = ref [];
stdIn:10.5-10.17 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val sez = ref [] : ?.X1 list ref
```

- 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



- leksikalni doseg
- currying, delna aplikacija
- mutacija
- določanje podatkovnih tipov
- vzajemna rekurzija

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


Moduli in Racket!