FUNKCIJSKO PROGRAMIRANJE

2023/24

podatkovni tipi

vezave

ujemanje vzocev

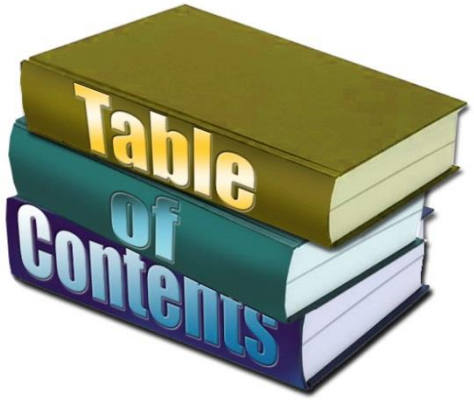
polimorfizem

izjeme

Ponovimo

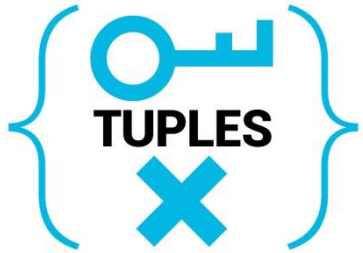
- prednosti funkcijskega programiranja
- vezave spremenljivk in funkcij
- statično in dinamično okolje
- enostavni izrazi (seštevanje, if-then-else)
- sintaktična in semantična evalvacija konstruktov programskega jezika
- podatkovni tipi:
  - int
  - bool
  - real
  - int * int -> int
- uporaba rekurzije

Pregled



- sestavljeni podatkovni tipi
  - terke (angl. tuples)
  - seznami (angl. lists)
  - zapisi (angl. records)
- vezave v lokalnem okolju
- podatkovni tip „opcija“ (angl. *option*)
- sinonimi za podatkovne tipe
- izdelava lastnih podatkovnih tipov
- ujemanje vzorcev s stavkom *case*
- definicija seznama in opcije
- polimorfizem podatkovnih tipov
- ujemanje vzorcev pri deklaracijah
- rekurzivno ujemanje vzorcev
- sklepanje na podatkovni tip
- izjeme

Terka (angl. *tuple*)



- podatkovni tip **nespremenljive** dolžine, sestavljen iz komponent **različnih** podatkovnih tipov
- **zapis terke**:

```
(e1, e2, …, en)
```

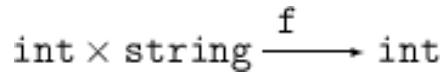
  če je podatkovni tip `e1: t1, …, en: tn`,
  je terka tipa `t1 * t2 * … * tn`

- **dostop do elementov** terke `e`

```
#n e
```

  kjer je `n` številka zaporedne komponente, `e` pa izraz-terka

Primeri uporabe terk



Napiši funkcije, s katerimi:

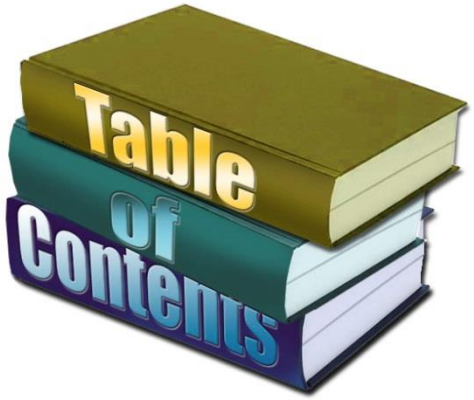
1. seštej števili, predstavljeni s terko (parom)
   ```
   fn : int * int -> int
   ```

2. obrni komponenti terke
   ```
   fn : int * int -> int * int
   ```

3. prepleti dve trimestni terki
   ```
   fn : (int * int * int) * (int * int * int)
   -> int * int * int * int * int * int
   ```

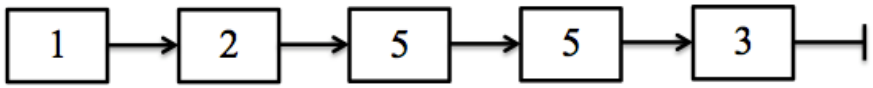
4. sortiraj komponenti terk po velikosti
   ```
   fn : int * int -> int * int
   ```

Pregled



- sestavljeni podatkovni tipi
  - terke (angl. tuples)
  - seznami (angl. lists)
  - zapisi (angl. records)
- vezave v lokalnem okolju
- podatkovni tip „opcija“ (angl. *option*)
- sinonimi za podatkovne tipe
- izdelava lastnih podatkovnih tipov
- ujemanje vzorcev s stavkom *case*
- definicija seznama in opcije
- polimorfizem podatkovnih tipov
- ujemanje vzorcev pri deklaracijah
- rekurzivno ujemanje vzorcev
- sklepanje na podatkovni tip
- izjeme

Seznam (angl. *list*)



- podatkovni tip **poljubne** dolžine, sestavljen iz komponent **enakih** podatkovnih tipov

- **zapis seznama s komponentami**:

  `[v1, v2, …, vn]`

  vsi elementi so istega podatkovnega tipa `t`

- **zapis seznama s sintakso** `glava::rep`

  če je glava vrednost `v0` in rep vrednost `[v1, v2, …, vn]`,
  ima zapis `glava::rep` vrednost `[v0, v1, …, vn]`
  pozor: glava je element, rep je seznam!

- podatkovni tipi seznama:

  `int list, real list,(int * bool) list, int list list, …`

Dostop do elementov seznama

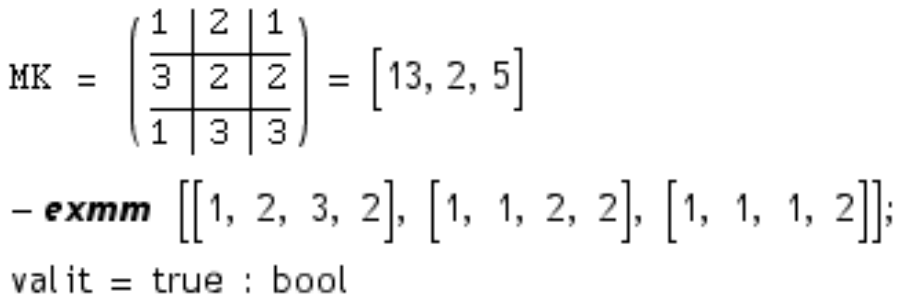
- **`null e`**
  vrne true, če je seznam prazen – `[ ]`
  ```
  fn : 'a list -> bool
  ```

- **`hd e`**
  vrne glavo seznama (element)
  ```
  fn : 'a list -> 'a
  ```

- **`tl e`**
  vrne rep seznama (ki je seznam)
  ```
  fn : 'a list -> 'a list
  ```

  - `hd` in `tl` prožita izjemo (exception), če je seznam prazen

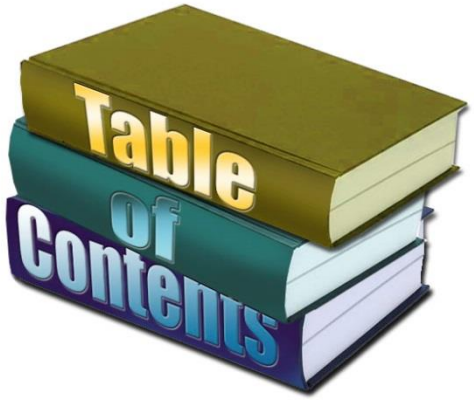
Primeri seznamov



Naloge:

1. preštej število elementov v seznamu
   ```
   fn : int list -> int
   ```
2. izračunaj vsoto elementov v seznamu
   ```
   fn : int list -> int
   ```
3. vrni n-ti zaporedni element seznama
   ```
   fn : int list * int -> int
   ```
4. združi dva seznama
   ```
   fn : int list * int list -> int list
   ```
5. prepleti elemente obeh seznamov (do dolžine krajšega seznama)
   ```
   fn : int list * int list -> (int * int) list
   ```
6. izračunaj vsote elementov v terkah (parih števil) v seznamu
   ```
   fn : (int * int) list -> int list
   ```
7. filtriraj seznam predmetov glede na pozitivno oceno izpita
   ```
   fn : (string * int) list -> string list
   ```

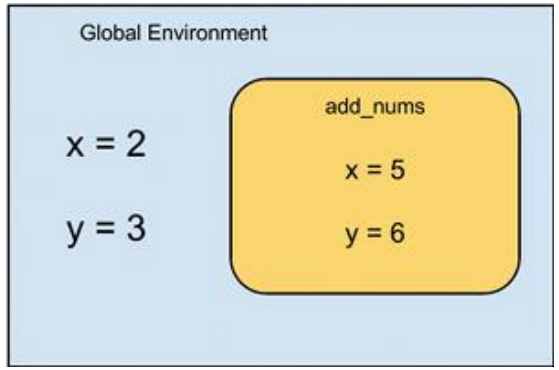
Pregled



- sestavljeni podatkovni tipi
  - terke (angl. tuples)
  - seznami (angl. lists)
  - zapisi (angl. records)
- vezave v lokalnem okolju
- podatkovni tip „opcija“ (angl. *option*)
- sinonimi za podatkovne tipe
- izdelava lastnih podatkovnih tipov
- ujemanje vzorcev s stavkom *case*
- definicija seznama in opcije
- polimorfizem podatkovnih tipov
- ujemanje vzorcev pri deklaracijah
- rekurzivno ujemanje vzorcev
- sklepanje na podatkovni tip
- izjeme

Lokalno okolje

- funkcije uporabljajo globalno statično/dinamično okolje ➔ potrebujemo konstrukt za izvedbo **lokalnih vezav** v funkciji
  - lepše programiranje
  - potrebne so samo lokalno
  - zaščita pred spremembami izven lokalnega okolja
  - v določenih primerih: nujno za performanse (sledi…)!



- **izraz „let“:**
  - je samo izraz, torej je lahko vsebina funkcije
  - **sintaksa**: `let d1 d2 … dn in e end`
  - **preverjanje tipov**: preveri tip vezav `d1, …, dn` in telesa `e` v zunanjem statičnem okolju. Tip celega izraza let je tip izraza `e`.
  - **evalvacija**: evalviraj zaporedoma vse vezave in telo `e` v zunanjem okolju. Rezultat izraza let je rezultat evalvacije telesa `e`.

Lokalno okolje

- novost: uvedemo pojem dosega spremenljivke (angl. *scope*)
- v lokalnem okolju imamo lahko tudi vezave lokalnih funkcij

```
fun sestej(c: int) =
    let
        val a = 5
        val b = a+c+1
    in
        a+b+c
    end
```

```
fun povprecje(sez: int list) =
    let
        fun stevilo_el(sez: int list) =
            if null sez
            then 0
            else 1 + stevilo_el(tl sez)
        fun vsota_el(sez: int list) =
            if null sez
            then 0
            else hd sez + vsota_el(tl sez)
        val vsota = Real.fromInt(vsota_el(sez))
        val n = Real.fromInt(stevilo_el(sez))
    in
        vsota/n
    end
```

Lokalno okolje

- notranje funkcije lahko uporabljajo zunanje vezave, odvečne (podvojene) reference lahko torej odstranimo

```
fun sestej1N (n: int) =
    let
       fun sestejAB (a: int, b: int) =
           if a=b then a else a + sestejAB(a+1, b)
    in
       sestejAB(1, n)
    end
```

b je vedno enak n in se med rekurzijo ne spreminja!

```
fun sestej1N (n: int) =
    let
       fun sestejAB (a: int) =
           if a=n then a else a + sestejAB(a+1)
    in
       sestejAB(1)
    end
```

(Ne)učinkovitost rekurzije

- težave lahko nastopijo pri večkratnih rekurzivnih klicih

```
fun najvecji_el (sez : int list) =
    if null sez
    then 0  (* maksimum praznega seznama je 0? *)
    else if null (tl sez)
    then hd sez
    else if hd sez > najvecji_el(tl sez)
    then hd sez
    else najvecji_el(tl sez)
```

- (brez težav) izvedba v primeru klica:
  `najvecji_el [30,29,28,27,26,…,7,6,5,4,3,2,1]`
  Vedno se kliče samo prvi rekurzivni klic (glava je večja od maksimuma v repu), torej:

  `najvecji_el[30,…,1]` ➔ `najvecji_el[29,…,1]`
  ➔ `najvecji_el[28,…,1]` ➔ … ➔ `najvecji_el[1]` ➔ konec

Učinkovitost rekurzije

```
fun najvecji_el (sez : int list) =
    if null sez
    then 0  (* maksimum praznega seznama je 0? *)
    else if null (tl sez)
    then hd sez
    else if hd sez > najvecji_el(tl sez)
    then hd sez
    else najvecji_el(tl sez)
```

- Kaj pa izvedba v primeru klica:
  `najvecji_el [1,2,3,4,5,6,7,8,9,10,…,26,27,28,29,30]`
- Vedno se kličeta oba rekurzivna klica, torej:

```
najvecji_el [1,2,…,30]
➢   najvecji_el [2,…,30]
      ➢   najvecji_el [3,…,30]
            ➢   …
            ➢   …
      ➢   najvecji_el [3,…,30]
            ➢   …
            ➢   …
➢   najvecji_el [2,…,30]
      ➢   …
            ➢   …
```

namesto 30 klicev
jih imamo ... koliko?

Učinkovitost rekurzije

- rešitev: uporaba lokalne spremenljivke, ki hrani rezultat rekurzivnega klica

```
fun najvecji_el (sez : int list) =
    if null sez
    then 0
    else if null (tl sez)
    then hd sez
    else let val max_rep = najvecji_el (tl sez)
         in
              if hd sez > max_rep
              then hd sez
              else max_rep
         end
```

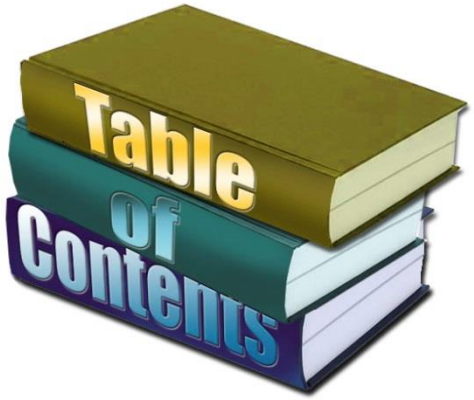
Problem

- v premislek:
  - Kateri je minimalni element praznega seznama?
  - Katero je zaporedno mesto (pozicija v seznamu) podanega elementa, ki ga v seznamu ni?

- kaj vrniti kot odgovor?
  - -1 ?
  - [ ] ?
  - null ?
  - prožiti izjemo?

- rešitev v SML: opcija, vezana na podatkovni tip:
  - `SOME <rezultat>`, če rezultat obstaja
  - `NONE`, če rezultat ni veljaven

Pregled



- sestavljeni podatkovni tipi
  - terke (angl. tuples)
  - seznami (angl. lists)
  - zapisi (angl. records)
- vezave v lokalnem okolju
- podatkovni tip „opcija“ (angl. *option*)
- sinonimi za podatkovne tipe
- izdelava lastnih podatkovnih tipov
- ujemanje vzorcev s stavkom *case*
- definicija seznama in opcije
- polimorfizem podatkovnih tipov
- ujemanje vzorcev pri deklaracijah
- rekurzivno ujemanje vzorcev
- sklepanje na podatkovni tip
- izjeme

Opcije

- tip `t option` (npr. `int option`, `string option`, …)
  - podobno kot "`list`" v primerih: `int list`, `(int*bool) list` itd.

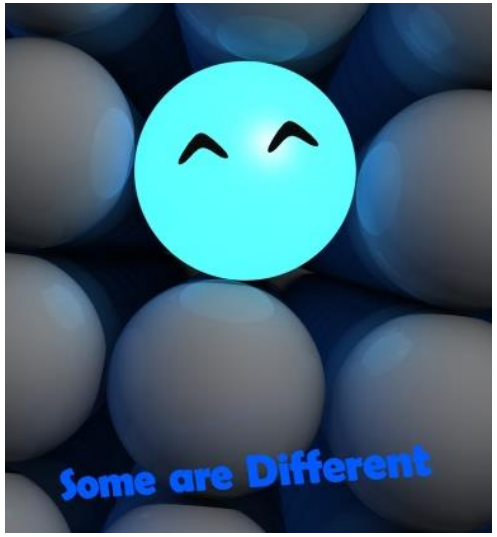
- zapis opcije
  - `SOME e` ➔ če je `e` tipa `t`, je `SOME e` tipa `t option`
  - `NONE` ➔ je tipa `'a option`

- dostop do opcije
  - `isSome`: preveri, ali je opcija v obliki `SOME`

```
val it = fn : 'a option -> bool
```

  - `valOf`: vrne vrednost e opcije `SOME e`

```
val it = fn : 'a option -> 'a
```



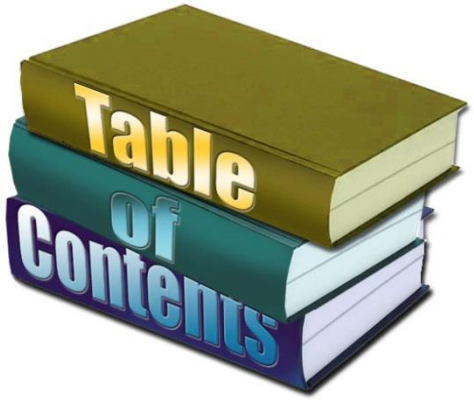
Izboljšava iskanja elementa

- primer: iskanje prve lokacije podanega elementa

```
(* poiscemo prvo lokacijo pojavitve elementa el *)
(* (int list * int) -> int option *)

fun najdi(sez: int list, el: int) =
    if null sez
    then NONE
    else if (hd sez = el)
    then SOME 1
    else let val preostanek = najdi (tl sez, el)
         in if isSome preostanek
            then SOME (1+ valOf preostanek)
            else NONE
         end
```

Pregled



- sestavljeni podatkovni tipi
  - terke (angl. tuples)
  - seznami (angl. lists)
  - zapisi (angl. records)
- vezave v lokalnem okolju
- podatkovni tip „opcija“ (angl. *option*)
- sinonimi za podatkovne tipe
- izdelava lastnih podatkovnih tipov
- ujemanje vzorcev s stavkom *case*
- definicija seznama in opcije
- polimorfizem podatkovnih tipov
- ujemanje vzorcev pri deklaracijah
- rekurzivno ujemanje vzorcev
- sklepanje na podatkovni tip
- izjeme

Podatkovni tipi – do sedaj

- enostavni PT
  - `int`
  - `bool`
  - `real`
  - `string`
  - `char`

- sestavljeni (kompleksni) podatkovni tipi
  - terke `(e1, e2, …, en)` – tip `t1 * t2 * … *tn`
  - seznami `[e1, e2, …, en]` – tip `'a list`
  - opcije `SOME e, NONE` – tip `'a option`
  - zapisi

- izdelava lastnih podatkovnih tipov?

} tema za danes

Zapis (angl. *record*)

- podatkovni tip s **poljubnim** številom **imenovanih** polj, ki hranijo vrednosti (lahko različnih podatkovnih podtipov)
- zapis zapisa:

```
{polje1 = e1, polje2 = e2, …, poljen = en}
```

  - če je podatkovni tip komponent enak `e1: t1, …, en: tn`, ima celotni zapis podatkovni tip `{polje1: t1, …, poljen: tn}`
    - vrstni red polj ni pomemben (SML prikaže v abecednem vrstnem redu)
    - tipi so lahko enostavni ali sestavljeni
    - podani so lahko izrazi, ki se pri deklaraciji evalvirajo v vrednosti
    - SML implicitno deklarira novi tip zapisa (ni treba tega narediti nam)

- dostop do elementov zapisa e

```
#ime_polja e
```

Primer uporabe zapisa

```
val zapis = {ime="Dejan", starost=21, absolvent=false,
    ocene=[("angl",8),("ars",10)] }

#absolvent zapis;

#ocene zapis;
```

```
(#ime zapis) ^ " je star "
             ^ Int.toString(#starost zapis) ^ " let."
```

Sinonimi za podatkovne tipe

Pogosto uporabljene in kompleksne (dolge) nazive podatkovnih tipov lahko poimenujemo z lastnim imenom in si poenostavimo delo.

```
type novo_ime = tip
```

```
fun izpis_studenta (zapis: {absolvent:bool, ime:string,
                            ocene:(string * int) list, starost:int}) =
    (#ime zapis) ^ " je star " ^  Int.toString(#starost zapis) ^ " let."
```

```
type student = {absolvent:bool, ime:string,
                ocene:(string * int) list, starost:int}
```

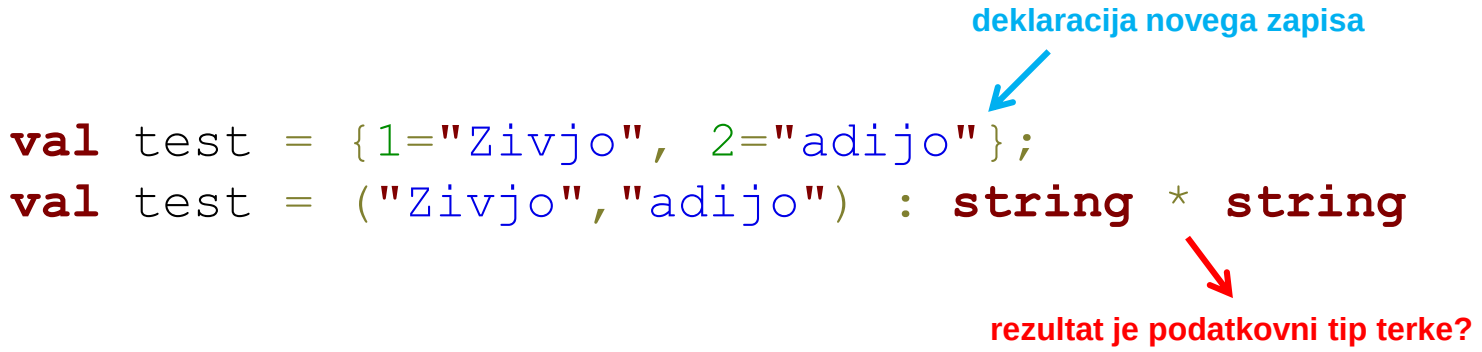
```
fun izpis_studenta2 (zapis: student) =
    (#ime zapis) ^ " je star " ^  Int.toString(#starost zapis) ^ " let."
```

- obe imeni tipov sta ekvivalentni
- SML lahko pri zapisovanju funkcij uporablja novo ali staro (dolgo) ime tipa (nepomembno)

```
val izpis_studenta2 = fn : student -> string
```

Terke in zapisi

- poglejmo si zanimiv primer…



- poseben tip "terka" torej v programskem jeziku ne obstaja! Terka je torej samo sintaktična olepšava/bližnjica za posebno obliko zapisa:
  - zapis `(e1,…,en)` namesto `{1=e1,…,n=en}`
  - zapis podatkovnega tipa `t1*…*tn` namesto `{1:t1, …, n:tn}`

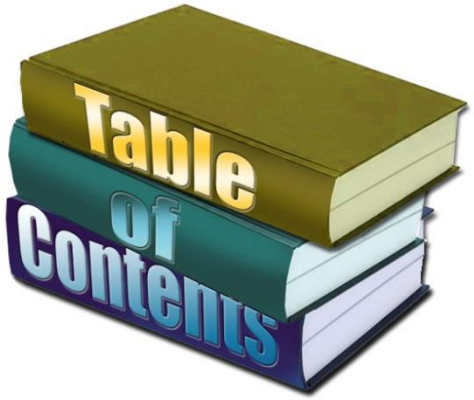
- sintaktična olepšave nam omogočajo lažje delo s programskim jezikom (razumevanje jezika in implementacijo lastnih programov)

Še več o terkah...

- terka – naslavljanje po vrstnem redu argumentov;
  zapis – naslavljanje po imenih argumentov
  - kdaj pri programiranju uporabljamo enega in drugega?

- terke ali polja?
  - pri majhnem številu elementov nam ni potrebno pomniti imen polj,
  - pri velikem številu elementov lažje pomnimo komponente po imenu kot po vrstnem redu

Pregled



- sestavljeni podatkovni tipi
  - terke (angl. tuples)
  - seznami (angl. lists)
  - zapisi (angl. records)
- vezave v lokalnem okolju
- podatkovni tip „opcija“ (angl. *option*)
- sinonimi za podatkovne tipe
- izdelava lastnih podatkovnih tipov
- ujemanje vzorcev s stavkom *case*
- definicija seznama in opcije
- polimorfizem podatkovnih tipov
- ujemanje vzorcev pri deklaracijah
- rekurzivno ujemanje vzorcev
- sklepanje na podatkovni tip
- izjeme

Izdelava lastnih podatkovnih tipov

- deklaracija novega podatkovnega tipa, ki predstavlja **alternativo** med podatkovnimi tipi, iz katerih je sestavljen:

```
datatype prevozno_sredstvo = Bus of int
                           | Avto of string * string
                           | Pes
```

konstruktorji

vsebina podatkovnega tipa

- (ali obstaja kaj podobnega v drugih programskih jezikih?)

- rezultat:
  - v okolju definiramo novi podatkovni tip `prevozno_sredstvo`
  - v okolju definiramo konstruktorje za izdelavo novih podatkovnih tipov: `Bus`, `Avto` in `Pes`

Vrednosti lastnih podatkovnih tipov

- vrednost novega podatkovnega tipa je vedno sestavljena z **oznako konstruktorja** (+ **vrednost**), npr:
  - `Bus 1`
  - `Avto ("fiat", "modri")`
  - `Pes`

- konstruktorja Bus in Avto sta funkciji, ki vrneta vrednost novega podatkovnega tipa:

```
fn : int -> prevozno_sredstvo
fn : string * string -> prevozno_sredstvo
```

- konstruktor Pes ne potrebuje argumenta in že sam predstavlja vrednost

```
val it = Pes : prevozno_sredstvo
```

- vrednost novega pod. tipa lahko opredelimo tudi z izrazom, npr. `Bus (1+5)`

Prednosti?

- omogoča definiranje različnih **alternativ** za zapis podatka
  - namesto redundantnih zapisov:

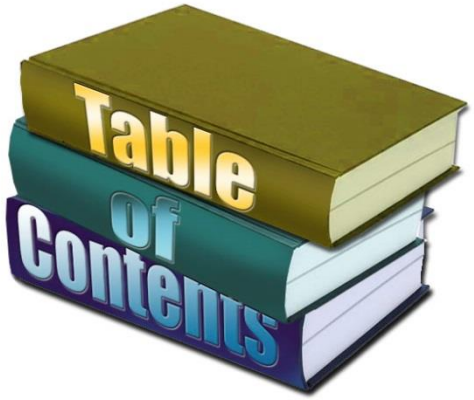
```
(* ce nacin =1 glej polje bus;
   ce = 2, glej avto; ce je 3 glej pes *)
 { nacin: int,
   bus: int,
   avto: string*string,
   pes: boolean}
```

ustvarimo eleganten (izključujoč) podatkovni tip

```
datatype prevozno_sredstvo = Bus of int
                           | Avto of string * string
                           | Pes
```

- omogoča **rekurzivno definiranje** tipa (pomembno za sezname, kasneje podrobno o tem…)

Pregled



- sestavljeni podatkovni tipi
  - terke (angl. tuples)
  - seznami (angl. lists)
  - zapisi (angl. records)
- vezave v lokalnem okolju
- podatkovni tip „opcija“ (angl. *option*)
- sinonimi za podatkovne tipe
- izdelava lastnih podatkovnih tipov
- ujemanje vzorcev s stavkom *case*
- definicija seznama in opcije
- polimorfizem podatkovnih tipov
- ujemanje vzorcev pri deklaracijah
- rekurzivno ujemanje vzorcev
- sklepanje na podatkovni tip
- izjeme

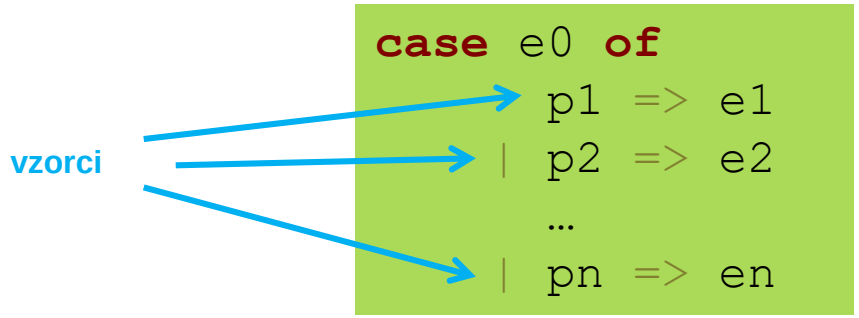
Delo z lastnimi podatkovnimi tipi

- lastni podatkovni tipi predstavljajo **alternativne** komponente

- (teoretično) imamo dve možnosti načina uporabe:
  1. pri programiranju **sproti preverjati**, s katerim podtipom dejansko delamo (ali je tip `prevozno_sredstvo` dejansko vrste `Bus`, `Avto` ali `Pes`?)
     - uporaba funkcij, kot bi bile `isBus`, `isAvto` (podobno kot `isSome` in `null`), in pridobiti podatke npr. z `getBusInt`, `getAvtoStrStr` (podobno kot `hd`, `tl`, `valOf`)
     - tak način je pogosto prisoten v dinamično tipiziranih jezikih (kako je s tem pri Javi?)
  2. podatek **primerjati z različnimi vzorci**
     - SML uporablja sistem primerjanja z vzorci!
     - stavek `case`

Stavek *case*

- primerja podani izraz `e0` za ujemanje z vzorci `p1, …, pn`
- rezultat je (samo eden) izraz na desni strani vzorca, s katerim se `e0` ujema
- vse veje `e1, …, en` morajo biti istega podatkovnega tipa



- primer:
  - naši vzorci so možne alternative podatkovnega tipa (konstruktor + spremenljivka)
  - spremenljivke v vzorcu dobijo dejanske vrednosti glede na podani argument

```
fun obdelaj_prevoz x =
    case x of
        Bus i => i+10
      | Avto (s1,s2) => String.size s1 + String.size s2
      | Pes => 0
```

Stavek *case*

- prednosti ujemanja vzorcev (in stavka `case`)?
  - okolje nas opozori, če pozabimo na primer vzorca
  - okolje nas opozori, če podvojimo vzorec
  - izognemo se okoliščinam, ko na podatkovnem tipu uporabimo napačno metodo za pridobitev vrednosti (npr. `valOf` na vrednosti `NONE` ali `hd` na seznamu `[]`)
  - lažje delo s funkcijami ➔ sledi v nadaljevanju

- kdaj vendarle uporabiti funkcije za preverjanje PT in ekstrakcijo podatkov (`null`, `hd`, `tl`)?
  - v argumentih funkcijskih klicev
  - kadar je preglednost programa večja

Primer: aritmetični izrazi

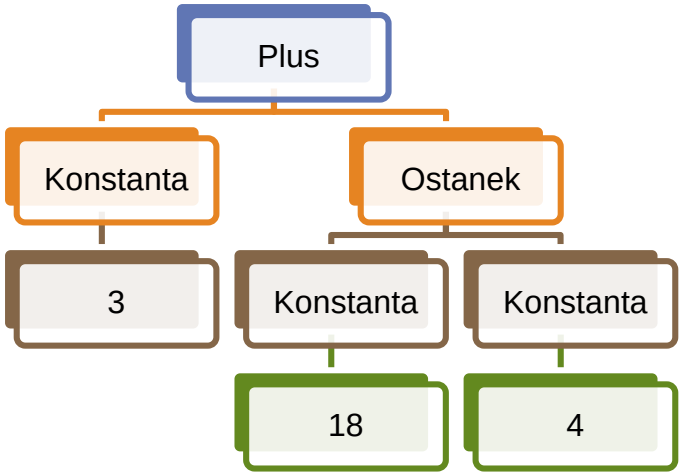
- definirajmo izraz kot **rekurzivni** (!) podatkovni tip

```
datatype izraz =  Konstanta of int
                | Negiraj of izraz
                | Plus of izraz * izraz
                | Minus of izraz * izraz
                | Krat of izraz * izraz
                | Deljeno of izraz * izraz
                | Ostanek of izraz * izraz
```

- primer izraza

```
Plus (Konstanta 3, Ostanek(Konstanta 18, Konstanta 4)
```

- izraze lahko predstavimo z drevesno strukturo

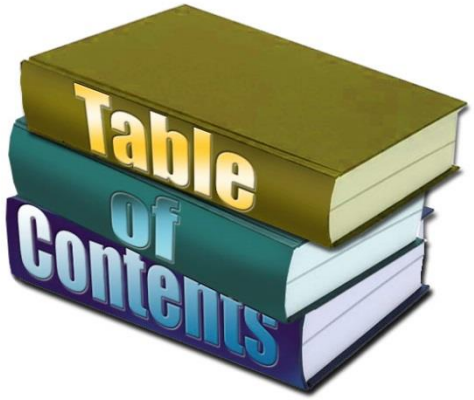


Naloge: aritmetični izrazi

Napiši funkcije tipa `fn : izraz -> int`, s katerimi:

1. evalviraj vrednost aritmetičnega izraza
2. preštej število negacij v izrazu
3. poišči maksimalno konstanto v izrazu (domača naloga za vajo)
4. poišči število primerov, kjer je ostanek pri deljenju enak 0 (domača naloga za vajo)

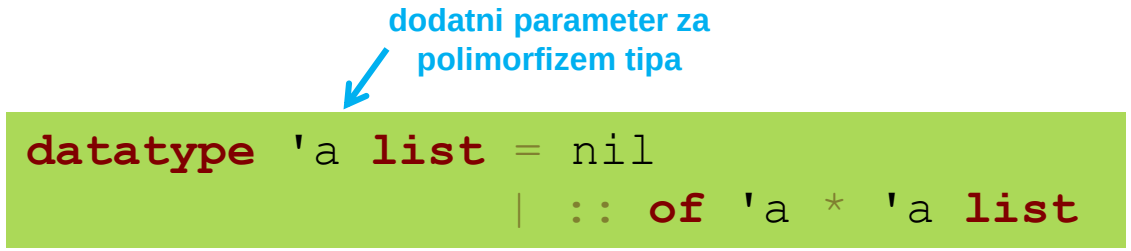
Pregled



- sestavljeni podatkovni tipi
  - terke (angl. tuples)
  - seznami (angl. lists)
  - zapisi (angl. records)
- vezave v lokalnem okolju
- podatkovni tip „opcija“ (angl. *option*)
- sinonimi za podatkovne tipe
- izdelava lastnih podatkovnih tipov
- ujemanje vzorcev s stavkom *case*
- definicija seznama in opcije
- polimorfizem podatkovnih tipov
- ujemanje vzorcev pri deklaracijah
- rekurzivno ujemanje vzorcev
- sklepanje na podatkovni tip
- izjeme

Resnica o seznamih in opcijah

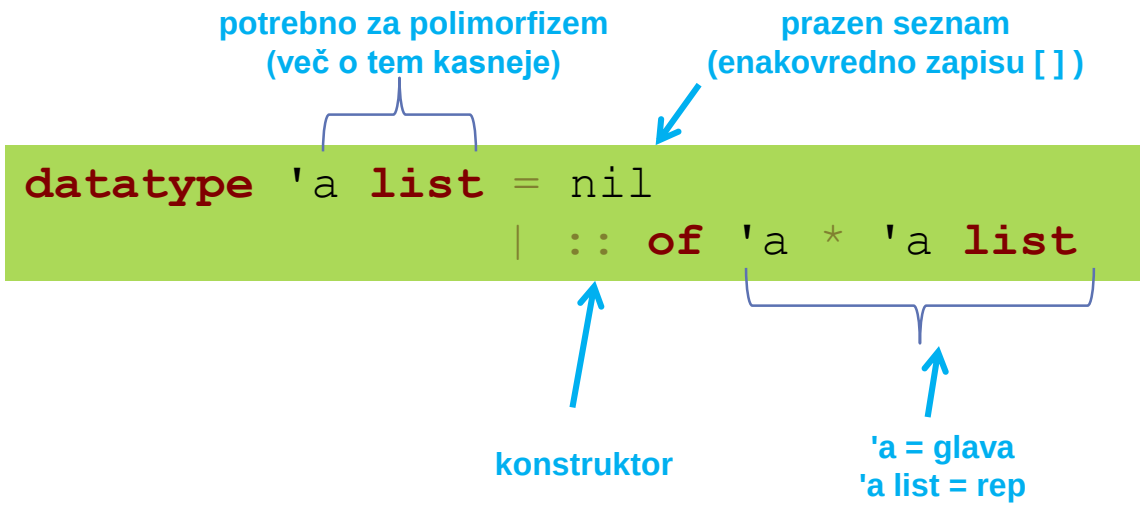
- le sintaktična olepšava v programskem jeziku (niso nujno potrebna komponenta
- definirana sta kot rekurzivna podatkovna tipa
- iz dokumentacije SML:
  - SEZNAM



  - OPCIJA

```
datatype 'a option =  NONE
                    | SOME of 'a
```

Seznami kot rekurzivni podatkovni tip



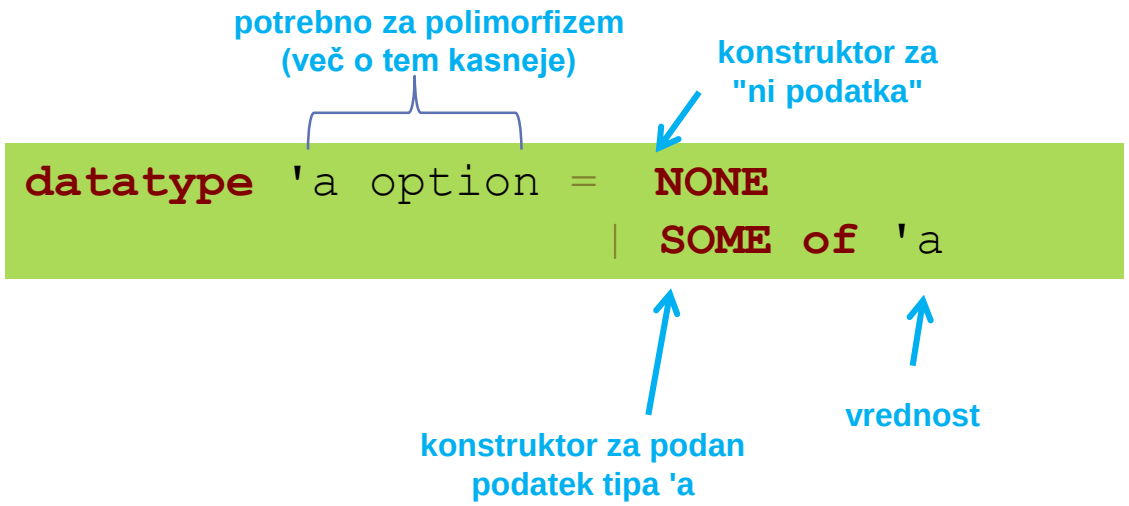
- posebnost: konstruktor `::` je definiran kot infiksni operator (izjema), zato ne moremo zapisati `::(glava, rep)`, temveč pišemo `glava::rep`

```
- 3::5::1::nil;
val it = [3,5,1] : int list
```

- ker seznami uporabljajo konstruktorje, lahko tudi na njih izvajamo ujemanje vzorcev (namesto uporabe `hd`, `tl`, `null`)

- funkcije `hd`, `tl` in `null` znamo sedaj sprogramirati sami!

Opcija kot rekurzivni podatkovni tip



- tudi pri opcijah lahko sedaj uporabimo ujemanje vzorcev
- funkcije `valOf` in `isSome` znamo sedaj sprogramirati sami!

Polimorfizem podatkovnih tipov

- novi podatkovni tip lahko uporablja poljuben drugi (vgnezdeni) podatkovni tip
- zahtevamo konsistentno rabo vgnezdenega tipa (pri vseh pojavitvah predstavlja `'a` isti tip; enako velja za `'b`, `'c` itd.)

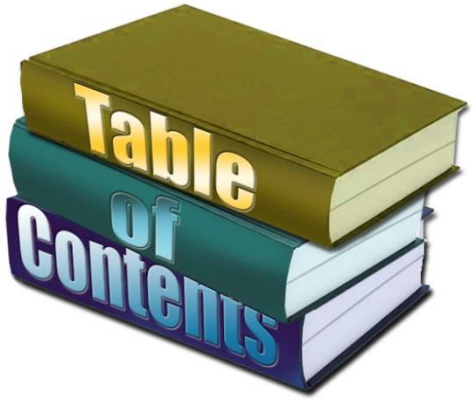
```
datatype 'a list = nil
                 | :: of 'a * 'a list
```

```
datatype 'a option =  NONE
                   | SOME of 'a
```

- primer: izdelajmo lasten polimorfen podatkovni tip: seznam, ki hrani dva različna tipa podatkov:

```
datatype ('a, 'b) seznam =
        Elementa of ('a * ('a, 'b) seznam)
      | Elementb of ('b * ('a, 'b) seznam)
      | konec
```

Pregled



- sestavljeni podatkovni tipi
  - terke (angl. tuples)
  - seznami (angl. lists)
  - zapisi (angl. records)
- vezave v lokalnem okolju
- podatkovni tip „opcija“ (angl. *option*)
- sinonimi za podatkovne tipe
- izdelava lastnih podatkovnih tipov
- ujemanje vzorcev s stavkom *case*
- definicija seznama in opcije
- polimorfizem podatkovnih tipov
- ujemanje vzorcev pri deklaracijah
- rekurzivno ujemanje vzorcev
- sklepanje na podatkovni tip
- izjeme

Resnica o deklaracijah

- deklaracije spremenljivk in funkcij dejansko uporabljajo **ujemanje vzorcev** na mestu, kjer smo navajali ime spremenljivke:

```
val vzorec = e
fun ime vzorec = e
```

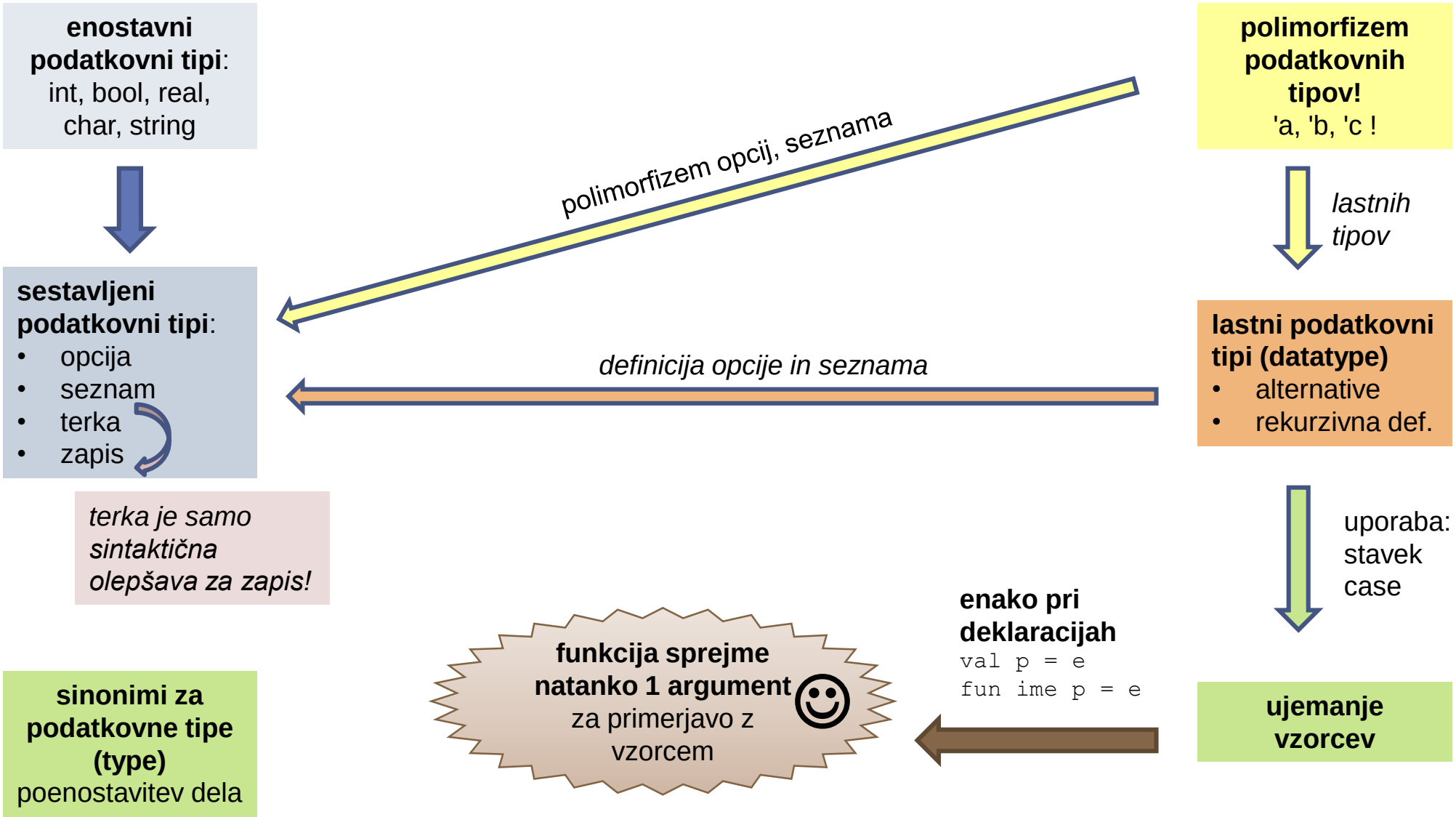
- zgornje pomeni, da vsaka **funkcija sprejema natanko en argument**, ki ga primerja z vzorcem
- ekvivalentna zapisa:

```
fun sestej1 (trojcek: int*int*int) =
    let val (a,b,c) = trojcek
    in a+b+c
    end
```

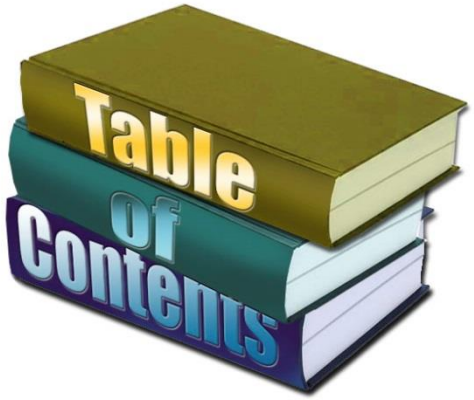
```
fun sestej2 (a,b,c) =    (* vzorec *)
    a + b + c
```

je kakšna razlika med zapisom z vzorcem in zapisom "funkcije s tremi argumenti"?

Kaj smo se danes naučili?



Pregled



- sestavljeni podatkovni tipi
  - terke (angl. tuples)
  - seznami (angl. lists)
  - zapisi (angl. records)
- vezave v lokalnem okolju
- podatkovni tip „opcija“ (angl. *option*)
- sinonimi za podatkovne tipe
- izdelava lastnih podatkovnih tipov
- ujemanje vzorcev s stavkom *case*
- definicija seznama in opcije
- polimorfizem podatkovnih tipov
- ujemanje vzorcev pri deklaracijah
- rekurzivno ujemanje vzorcev
- sklepanje na podatkovni tip
- izjeme

Rekurzivno ujemanje vzorcev

- namesto vgnezdenih 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
```

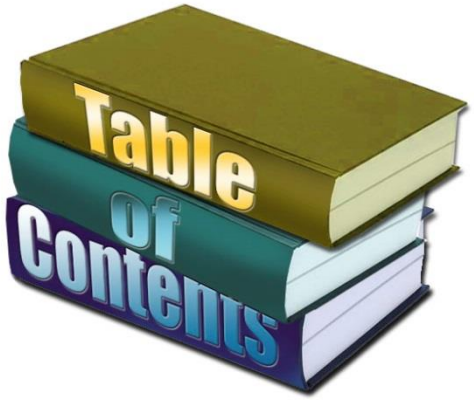
2. Podan je seznam, ki predstavlja zaporedje, izračunano po Fibonaccijevem zakonu. Preveri, ali je seznam veljavno takšno zaporedje.

```
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



- sestavljeni podatkovni tipi
  - terke (angl. tuples)
  - seznami (angl. lists)
  - zapisi (angl. records)
- vezave v lokalnem okolju
- podatkovni tip „opcija“ (angl. *option*)
- sinonimi za podatkovne tipe
- izdelava lastnih podatkovnih tipov
- ujemanje vzorcev s stavkom *case*
- definicija seznama in opcije
- polimorfizem podatkovnih tipov
- ujemanje vzorcev pri deklaracijah
- rekurzivno ujemanje vzorcev
- sklepanje na podatkovni tip
- izjeme

Sklepanje na podatkovni tip

- SML ima vgrajen sistem za sklepanje na podatkovni tip funkcije, tudi če ročno ne navajamo vhodnega in izhodnega tipa

- pogoj 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`

Primerjalni podatkovni tipi

- naslednja funkcija:

```
fun f1 (a,b,c,d) =
  if a=b
  then c
  else d
```

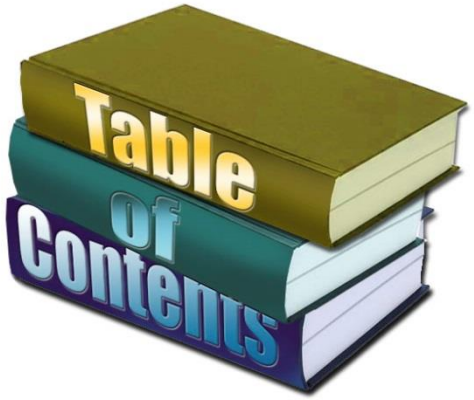
je polimorfnega tipa:

```
val f1 = fn : ''a * ''a * 'b * 'b -> 'b
```

dva apostrofa označujeta primerjalni podatkovni tip

- **primerjalni podatkovni tip** (angl. **eqtype**):
  - Je tudi polimorfni podatkovni tip.
  - Zanj mora veljati sposobnost primerjanja enakosti z drugim tipom (posledica "if a=b") v funkciji.
  - Ker 'a pomeni "*poljuben tip*", ''a pa "*poljuben primerjalni tip*", je **'a bolj splošna oznaka kot ''a**.
  - Zapis tipa (''a) torej predstavlja dodatno omejitev, na katero opozarja programerja.

Pregled



- sestavljeni podatkovni tipi
  - terke (angl. tuples)
  - seznami (angl. lists)
  - zapisi (angl. records)
- vezave v lokalnem okolju
- podatkovni tip „opcija“ (angl. *option*)
- sinonimi za podatkovne tipe
- izdelava lastnih podatkovnih tipov
- ujemanje vzorcev s stavkom *case*
- definicija seznama in opcije
- polimorfizem podatkovnih tipov
- ujemanje vzorcev pri deklaracijah
- rekurzivno ujemanje vzorcev
- sklepanje na podatkovni tip
- izjeme

Izjeme

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

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

- proženje 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 => ~9999999
```

Deklariranje tipov, ujemanje vzorcev