

Podatkovni tipi – do sedaj

- enostavni PT

- `int`
- `bool`
- `real`
- `string`
- `char`



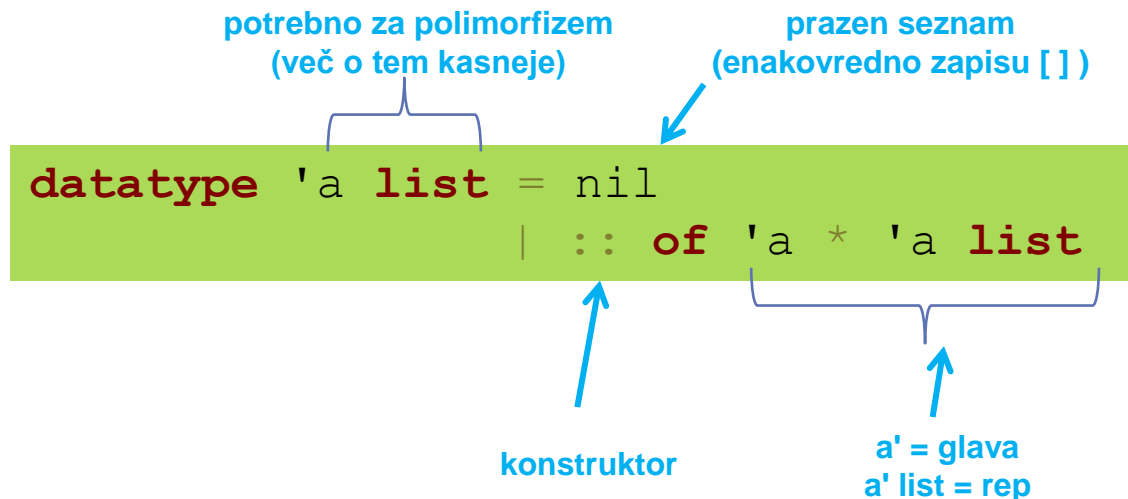
- sestavljeni (kompleksni) podatkovni tipi

- `terke` `(e1, e2, ..., en)` – tip `t1 * t2 * ... * tn`
- `seznam` `[e1, e2, ..., en]` – tip `a' list`
- `opcije` `SOME e, NONE` – tip `a' option`
- `zapisi`

- izdelava lastnih podatkovnih tipov?

} tema za danes

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 sezname uporabljajo konstruktorje, lahko tudi na njih izvajamo ujemanje vzorcev (namesto uporabe `hd`, `tl`, `null`)
- funkcije `hd`, `tl` in `null` znamo sedaj sprogramirati sami!


Rekurzivno ujemanje vzorcev

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

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

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

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



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

Izjeme

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

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

- klic izjeme

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

- obravnava izjeme

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

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
```

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

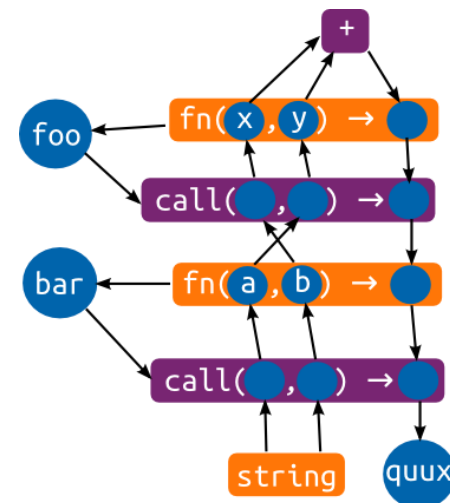
```
- (((vmejah_curry 5) 15) [1,5,3,43,12,3,4]);
```

lahko lepše zapišemo brez oklepajev

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

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


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]; (* po uri 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
```

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

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
```

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

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

KORAK 1:
skrijemo funkcijo za množenje

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

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

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. modul vsebuje vse ne-abstraktne tipe, navedene v podpisu
2. modul vsebuje implementacijo abstraktnih tipov iz podpisa
(`datatype`, `type`)
3. vsaka deklaracija vrednosti (`val`) v podpisu se nahaja v modulu (v modulu je lahko bolj splošnega tipa)
4. vsaka izjema (`exception`) v podpisu se nahaja tudi v modulu

