

Rekurzija

prejšnjič...

Spoznali smo osnovne konstrukcije na množicah

$A \times B$

$A + B$

$A \rightarrow B$

Spoznali smo programski jezik OCaml

```
let zamakni_znak zamik znak =
  if 'A' <= znak && znak <= 'Z' then
    let mesto_znaka = Char.code znak - Char.code 'A' in
    let novo_mesto = (mesto_znaka + zamik) mod 26 in
    Char.chr (Char.code 'A' + novo_mesto)
  else
    znak

let sifriraj_niz zamik =
  String.map (zamakni_znak zamik)
```



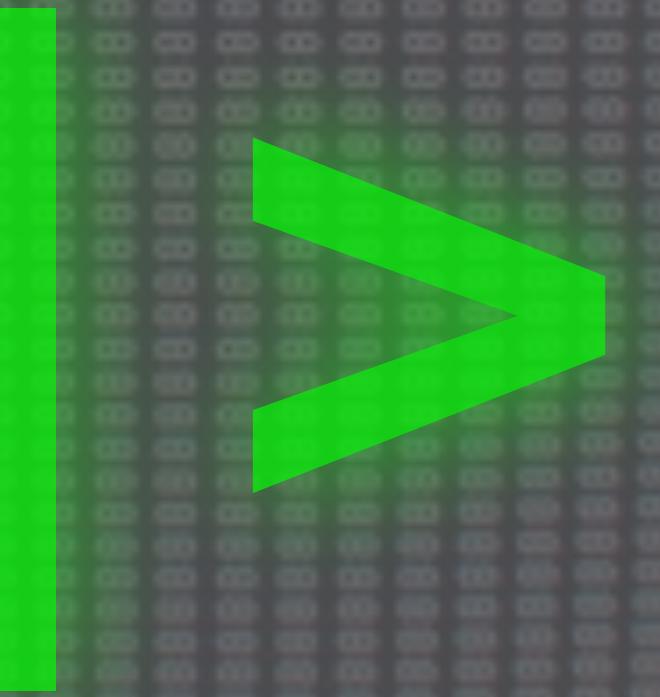
OCaml

Naštevne tipe smo razstavljali z ujemanjem vzorcev

```
type dostava =
| OsebniPrevzem
| PoPosti of naslov
| HitraDostava of naslov * telefon

let cena_dostave =
function
| OsebniPrevzem -> 0.
| PoPosti naslov -> 2.5 +. dodatek naslov
| HitraDostava (naslov, _) -> 4. +. dodatek naslov
```

operacija veriženja



Splošna oblika rekurzije

Kakšno je matematično ozadje rekurzivnih definicij?

```
public static int fact(int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fact(n - 1);  
    }  
}
```

```
let rec fact n =  
    if n = 0 then 1 else n * fact (n - 1)
```

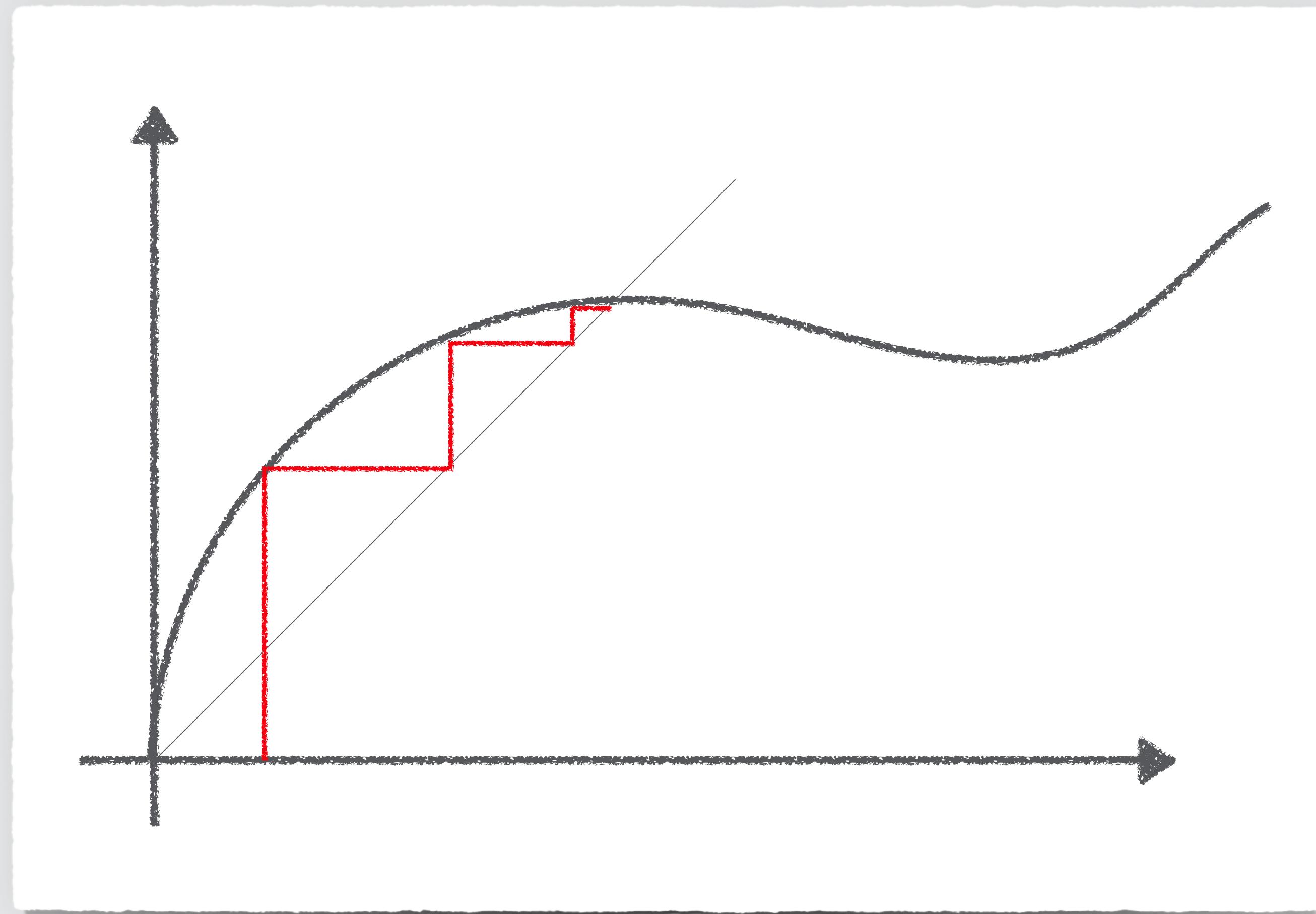


Negibne točke poznamo iz matematike

$$f(x) = x$$

Negibno točko lahko izračunamo z iteracijo

$x_0, f(x_0), f(f(x_0)), f(f(f(x_0))), \dots$



negibná točka preslikave

$$x \mapsto \frac{1}{1+x}$$

Kaj bi bila “negibna točka” funkcije med ukazi?

$c' \mapsto \text{if } b \text{ then } (c; c') \text{ else skip}$

$\text{while } b \text{ do } c \text{ done} =$
 $\text{if } b \text{ then } (c; \text{while } b \text{ do } c \text{ done}) \text{ else skip}$

Vsako rekurzivna definicija je **negibna točka**

$$f = \lambda n . \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f(n - 1)$$

$$\Phi = \lambda g . (\lambda n . \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot g(n - 1))$$

$$f = \Phi f$$

Tudi rekurzivne funkcije dobimo z **iteracijo**

$$\Phi = \lambda g . \lambda n . \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot g(n - 1)$$

$$f_0$$

$$\Phi(f_0)$$

$$\Phi(\Phi(f_0))$$

$$\Phi(\Phi(\Phi(f_0)))$$

negibna točka preslikave

$$g \mapsto \lambda n. \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{else} \\ n \cdot g(n - 1) & \end{cases}$$

Z eno samo rekurzivno funkcijo lahko izrazimo vse

```
let rec fact n =  
    if n = 0 then 1 else n * fact (n - 1)
```

```
let fact_telo g n =  
    if n = 0 then 1 else n * g (n - 1)
```

```
let rec fact n = fact_telo fact n
```

```
let rec rek telo = fun n -> telo (rek telo) n  
let fact = rek fact_telo
```

rek & rek2
funkciji

Rekurzivno lahko definiramo tudi **druge vrednosti**

```
ghci> enDvaTri = 1 : 2 : 3 : enDvaTri
```

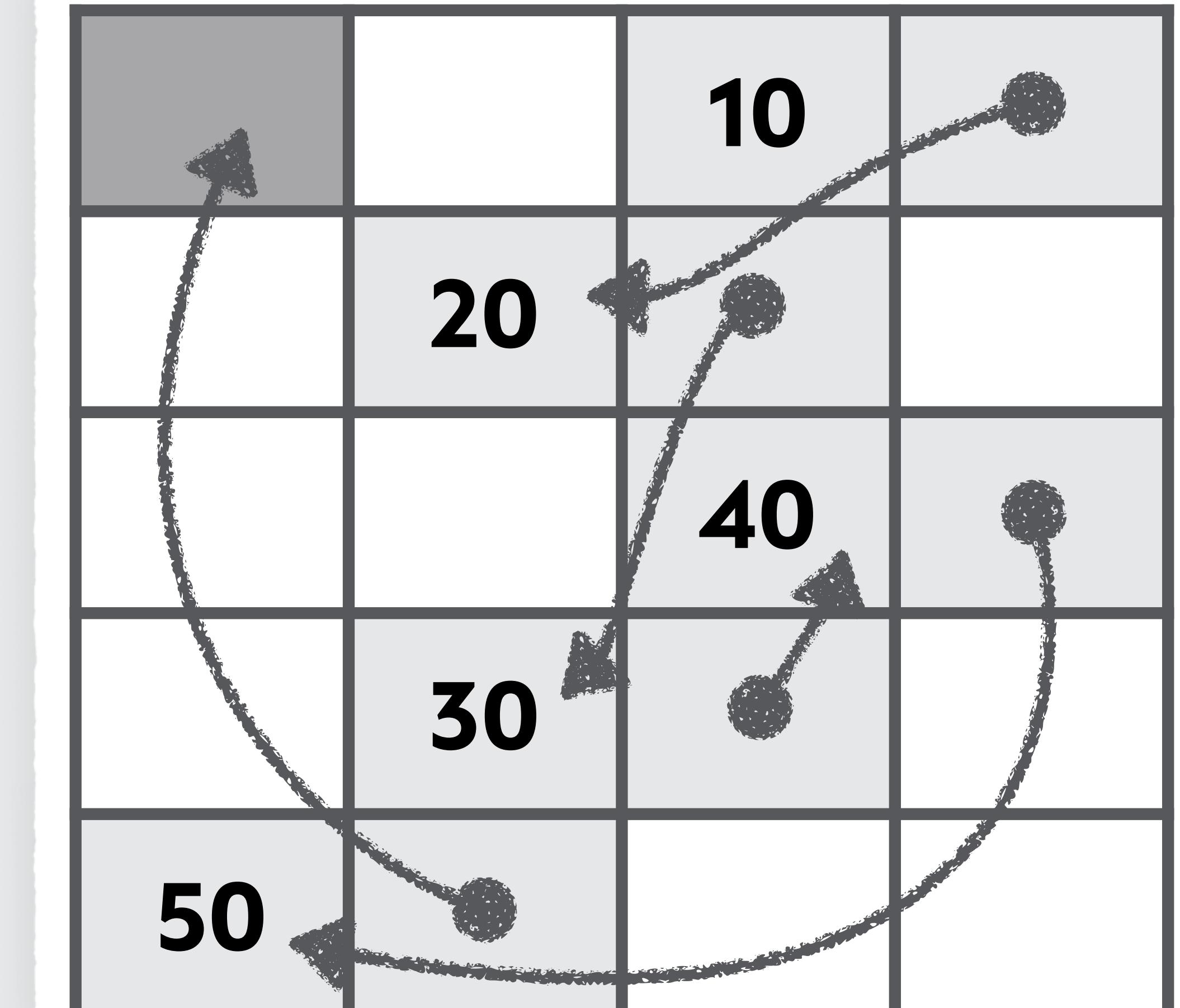
```
ghci> enDvaTri
```

Rekurzivni tipi

Sezname lahko implementiramo na dva načina

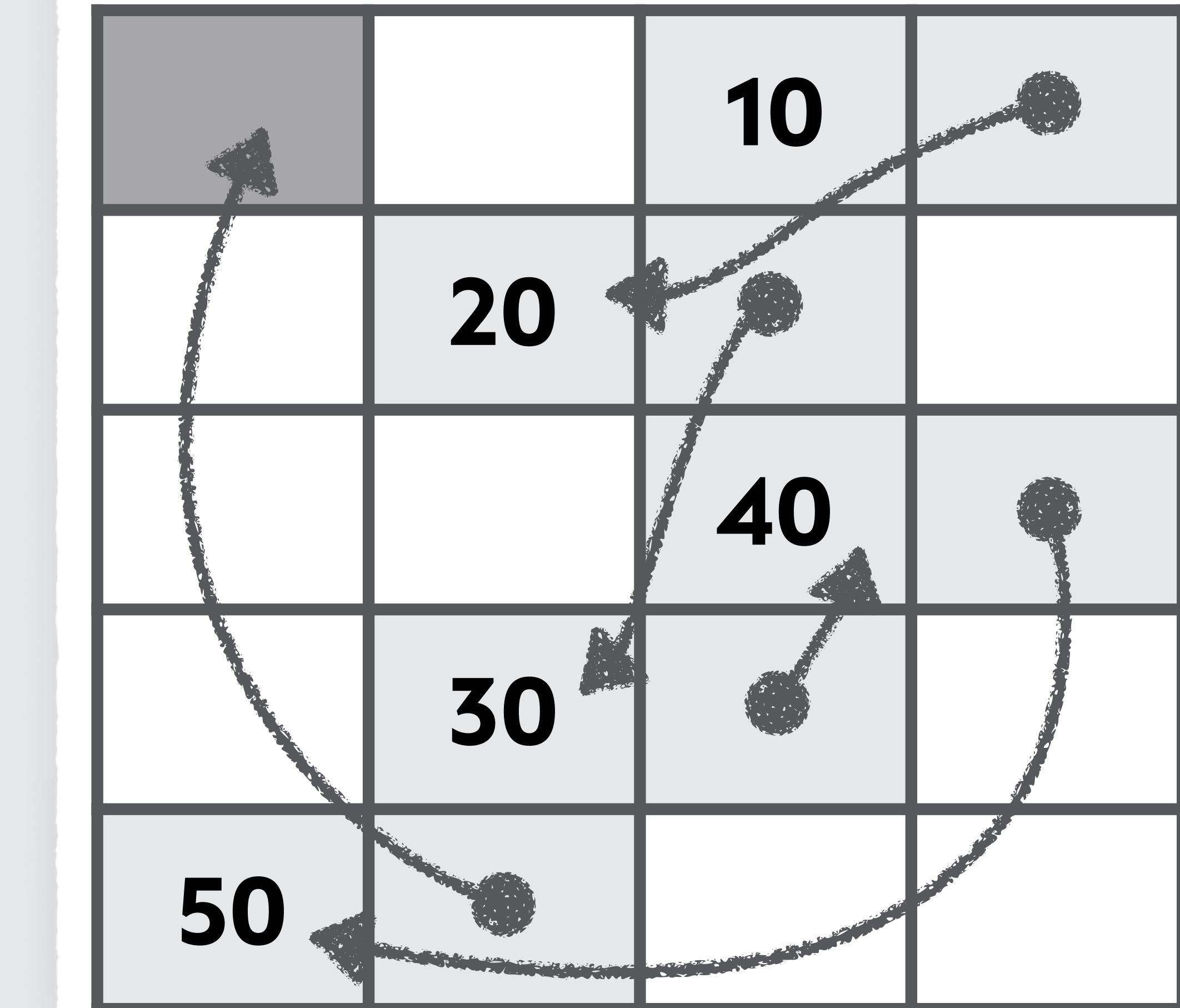
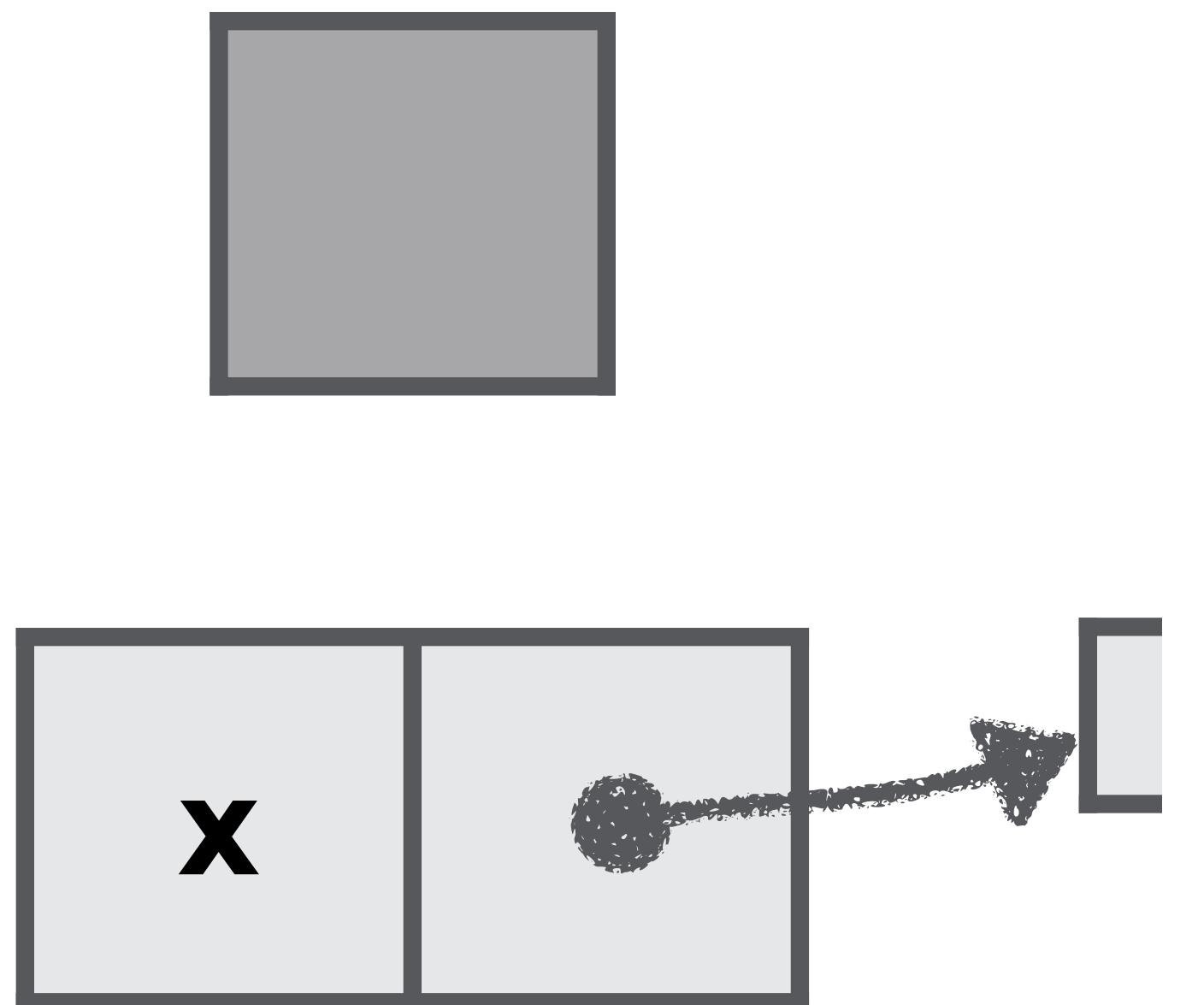
5	10	20	30
40	50		

tabela



verižni seznam

Verižni seznami so primer **rekurzivnega tipa**





seznam

tip

Najenostavnejši primer rekurzivnega tipa so **naravna števila**

0 n +

**tip
naravníh
štěvít.**

Z induktivnimi tipi predstavljamо gnezdene podatke

```
type json =
| String of string
| Number of int
| Object of (string * json) list
| Array of json array
| True
| False
| Null
```

Z induktivnimi tipi predstavljamo **abstraktno sintakso**

$$e ::= n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \mid - e$$

```
type expression =
| Numeral of int
| Plus of expression * expression
| Minus of expression * expression
| Times of expression * expression
| Divide of expression * expression
| Negate of expression
```

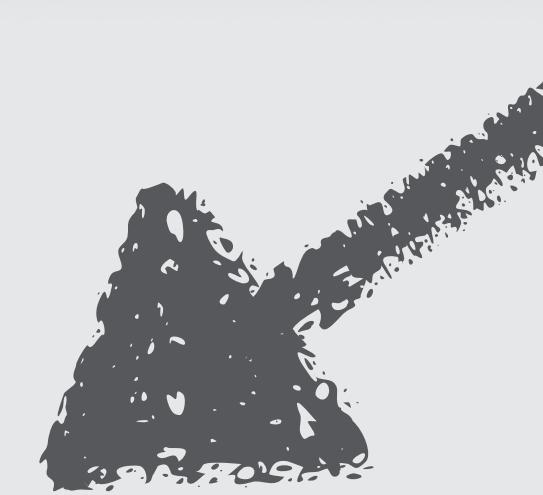
tip
dvodjetških
dreves

Induktivni in koinduktivni tipi

Ali so elementi rekurzivnih tipov lahko neskončni?

```
type seznam = Nil | Cons of int * seznam
```

```
Cons (1, Cons (2, Cons (3, Cons (4, ...))))
```



ne



da

induktivni tipi

koinduktivni tipi

Za delo s koinduktivnimi tipi si bomo ogledali **Haskell**





tip
tokov

The logo consists of the word "tip" in a smaller, bold, black sans-serif font above the word "tokov" in a larger, bold, black sans-serif font. Both words are centered on a white rectangular background. This central rectangle is set against a dark gray circular background, which is itself centered on a white square background.

V OCamlu lahko koinduktivne tipe simuliramo z **zakasnitvami**

```
type 'a stream =  
  Cons of 'a * (unit -> 'a stream)
```



prihodnjič...

Predavanja odpadejo, na vajah boste reševali stare izpite

1. naloga (30 točk)

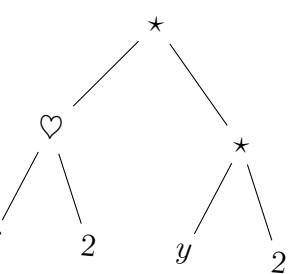
a) (6 točk) V Elboniji uporabljajo za aritmetične izraze drugačne simbole kot njeni. Sintaksa je podana s pravili:

```

<aritmetični-izraz> ::= <srčni-izraz>
  <srčni-izraz> ::= <zvezdni-izraz> | <srčni-izraz> ○ <zvezdni-izraz>
  <zvezdni-izraz> ::= <zaboden-izraz> | <zvezdni-izraz> * <zaboden-izraz>
  <zaboden-izraz> ::= <spremenljivka> | <številka> | †<zaboden-izraz> | ( <zaboden-izraz> )
  <spremenljivka> ::= [a - zA - z] +
  <številka> ::= [0 - 9] +

```

Simbol † ima prednost pred *, ki ima prednost pred ○. Simbol ○ je levo asociativen. Drevo



predstavlja elbonijski aritmetični izraz. Zapišite ga v konkretni sintaksi s črkoj.

b) (6 točk) Timotej je pognal program

```

while k > 0 do
  if k mod 2 = 0 then
    d := d + 1
  else
    skip
  end ;
  k := k div 2
done

```

v okolju $[a \mapsto 0, d \mapsto 3, k \mapsto 42]$. Kakšno je končno okolje, ko se program konča?

1. $[a \mapsto 1, d \mapsto 6, k \mapsto 1]$
2. $[d \mapsto 6, k \mapsto 0]$
3. $[a \mapsto 0, d \mapsto 6, k \mapsto 0]$
4. $[a \mapsto 0, d \mapsto 0, k \mapsto 0]$

c) (6 točk) Andrej je sestavil program P :

```

while n > 1 do
  if n mod 2 = 0 then
    n := n / 2
  else
    n := 3 * n + 1
done

```

Označite vse specifikacije, ki jim zadošča Andrejev program:

- (a) $[n = 3] P [true]$
- (b) $\{true\} P \{n = 1\}$
- (c) $\{n = 0\} P \{n = 1\}$
- (d) $[n = 0] P [n = 1]$

d) (6 točk) V λ -računu evaluiramo izraz

$$(\lambda fx. f(fx))(\lambda f. ff)(\lambda x. x)$$

Kateri izraz dobimo?

- (a) $\lambda z. z$
- (b) $\lambda f. ff$
- (c) $\lambda x. x(xx)$
- (d) izraza ne moremo evaluirati

e) (6 točk) Dan je parametrični tip

$$((\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \alpha \times \beta \rightarrow \gamma) \text{ list}$$

Označite vse SML izraze, ki imajo ta tip:

- (a) []
- (b) $\text{fn } f \Rightarrow \text{fn } (x, y) \Rightarrow f y x$
- (c) $(\text{fn } f \Rightarrow \text{fn } (x, y) \Rightarrow f y x) :: []$
- (d) $[(\text{fn } f \Rightarrow \text{fn } (x, y) \Rightarrow f x y)]$

d) (6 točk) Andrej je definiral signaturo v SML:

```

signature S =
sig
  type t
  val pi : real
  val f : t -> t -> t
  val g : 'a -> 'a list
end

```

Timotej je implementiral štiri strukture:

```

structure Foo =
struct
  type t = int -> int
  val pi = 3.141592653589793
  fun g x = [x]
  fun r x = [x]
  fun f h k x = k (h x)
end

```

```

structure Bar =
struct
  type t = bool
  type s = int * int
  fun f b c = b
  fun g k = k : g (k + 1)
  val pi = if 17 * 18 < 20 * 15 then 42.0 else 23.0
end

```

```

structure Baz =
struct
  type t = bool
  fun f (h, k) = (fn x => h (k x))
  fun g _ = []
  val pi = 42
end

```

```

structure Qux =
struct
  type t = bool
  fun f (h, k) = (fn x => h (k x))
  fun r x = [x]
  val pi = 3.141592653589793
end

```

Označite tiste strukture, ki zadoščajo signaturi S .

čez dva teda...

Primerjali bomo različne sisteme tipov

```
>>> type((1, 'foo', False))  
<class 'tuple'>
```

```
# (1, "foo", false)  
- : int * string * bool = ...
```

Videli bomo, kako **določiti tip** programa

$\Gamma \vdash e : \tau$

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x. e : \sigma \rightarrow \tau}$$

$$\frac{\Gamma \vdash e_1 : \sigma \rightarrow \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \tau}$$