PPJ

Ukazi (virtualka)

- **ocaml**

```
opam install utop
utop
#use "test.ml";;
```

ali

```
ocaml
```

- **prolog**

```
swipl
[test].
```

- **lambda**

```
make all
./plzoo/lambda.native -l example.lambda --wrapper rlwrap
```

(naredi `example.lambda` na `Desktop`)

Vsebina

1. **Aritmetični izrazi**
2. **Dokazovanje pravilnosti**
3. **Lambda račun**
4. **Deklarativno programiranje**
   - **podatkovni tipi**
   - **AVL drevesi**
   - **rekurzija**
   - **specifikacija in abstrakcija**
   - **objektno programiranje**
5. **Koinduktivni tipi in izpeljava tipov**
6. **Logično programiranje**

ARITMETIČNI IZRAZI

Drevo

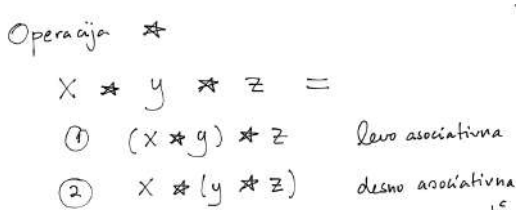
- koren: glavna operacija
- leva asociativnost: začetek izraza se začne v levem listu drevesa

Gramatike (kako tvorimo izraze/drevesa)
Trikotniki - entitete
Prioriteta pada od zgoraj navzdol.

Konkretna (niz znakov) --> **abstraktna sintaksa** (drevo)
desni: v korenu

Abstraktna --> konkretna sintaksa

- asociativnost



asociativnost	prioriteta	operator
/	0	spremenljivke in konstante
desno	1	potenca
desno	2	unarni minus (-)
levo	3	*, /
levo	4	+, -
levo	5	and, or, ;

desna asociativnost: gnezdeno desno od njega (argument na desni strani), na levi strani nima nič

leva asociativnost: najprej pomnožimo vse kar je na levi strani operatorja in nato dodamo nek elementarni element na desni (in ga pomnožimo)

Kalkulator

Kalkulator razume jezik, opisan z naslednjo gramatiko v notaciji BNF:

```
⟨izraz⟩ ::= ⟨aditivni⟩ EOF
⟨aditivni⟩ ::= ⟨multiplikativni⟩ | ⟨aditivni⟩ + ⟨multiplikativni⟩
⟨multiplikativni⟩ ::= ⟨nasprotni⟩ | ⟨multiplikativni⟩ * ⟨nasprotni⟩
⟨nasprotni⟩ ::= - ⟨nasprotni⟩ | ⟨osnovni⟩
⟨osnovni⟩ ::= ( ⟨aditivni⟩ ) | ⟨spremenljivka⟩ | ⟨konstanta⟩
⟨spremenljivka⟩ ::= [a-zA-Z]+
⟨konstanta⟩ ::= ⟨float⟩
```

Pravilo za ⟨float⟩ je enako kot zapis nenegativnih konstant tipa float v Javi, na primer 3.14.

Ko imamo gramatiko podano v BNF obliki lahko poleg strukture aritmetičnih izrazov razberemo še njihovo **prioriteto** in **asociativnost**.

Prva vrstica enote predstavlja nek izraz, ki predstavlja aditivni izraz.

Pravilo za `aditivni`
Nekaj kar je `multiplikativni` ali nekaj kar je `aditivni` + nekaj kar je `multiplikativni` ...
gre za vsoto pri čemer je lahko vsota samo en element ali imamo poljubno število zaporednih prištevanj - seštevanje je levo asociativno (prištevamo z desne, vsoto seštejemo na levi strani), neskončna vsota je veljaven aritmetični izraz v dani slovnici.

Pravilo za `multiplikativni`
Imamo produkt, ki ima samo en element (ni znaka za množenje) in ponavljanje `multiplikativni`.
Operator za množenje je levo asociativen.

Pravilo za `nasprotni`
Lahko imamo `osnovni` ali – `(nasprotni)`, lahko imamo poljubno število minusov, med njimi je poljubno število presledkov.
Na desni strani imamo lahko prazen niz, na levi strani pa se rekurzivno sklicujemo na nasprotni.
Če želimo razrešiti izraz, moramo razrešiti najprej levo stran. Prej: če smo želeli razrešiti nek produkt, smo mogli najprej razrešiti desno stran.
Asociativnost: desna (izraz imamo na desni strani)
Če ne vemo, katera asociativnost, lahko poskušamo z oklepaji.

Pravilo za `osnovni`: konstanta, spremenljivka (neprazno zaporedje črk, ki predstavlja ime spremenljivke), aditivni izraz, ki je v oklepajih (sklicujemo nazaj)

Prioriteta od zgoraj navdol pada (gnezdenost) - višja prioriteta ima nižjo številko, pove nam v kakšnem vrstnem redu moramo izračunati.

desna asociativnost: gnezdeno desno od njega (argument na desni strani), na levi strani nima nič

leva asociativnost: najprej pomnožimo vse kar je na levi strani operatorja in nato dodamo nek elementarni element na desni (in ga pomnožimo)

Če bi bilo odštevanje desno asociativno: a - (b - c)

a ** (-b) ... potenca ima višjo prioriteto od unarnega minusa

Dodajanje gradnikov k gramatiki BNF

Deljenje

Enaka asociativnost in prioriteta pri množenju in deljenje => dodajanje k istemu gradniku (multiplikativni).

⟨multiplikativni⟩ ::= ⟨nasprotni⟩ | ⟨multiplikativni⟩ * ⟨nasprotni⟩ | ⟨multiplikativni⟩ / ⟨nasprotni⟩

Odštevanje

Enaka asociativnost in prioriteta pri seštevanju in odštevanju => dodajanje k istemu gradniku (aditivni).

⟨aditivni⟩ ::= ⟨multiplikativni⟩ | ⟨aditivni⟩ + ⟨multiplikativni⟩ | ⟨aditivni⟩ - ⟨multiplikativni⟩

Potenca

Potenca nima iste asociativnosti in prioritete kot drugi elementi => dodajanje novega gradnika. Posledično je potrebno popraviti tudi gradnik nasprotni, saj lahko zdaj sprejme tudi gradnik potenco, ne samo osnovni.

Potenca je lahko `osnovni` (potence nimamo), na levi lahko imamo: `osnovni`, `osnovni`, `osnovni` .... ampak iz tega ne bi bila razvidna asociativnost. Kako dosežemo desno asociativnost?
Potenca na desni strani je gradnik potenca, kar pomeni, da lahko vsebuje gnezdene potence (najprej poračunamo gnezdene potence in šele na koncu uporabimo ta rezultat pri končni vrednosti potence).

⟨potenca⟩ ::= ⟨osnovni⟩ | ⟨osnovni⟩ ^ ⟨potenca⟩

⟨nasprotni⟩ ::= - ⟨nasprotni⟩ | ⟨potenca⟩

ALI

⟨potenca⟩ ::= ⟨nasprotni⟩ | ⟨potenca⟩ ^ ⟨nasprotni⟩

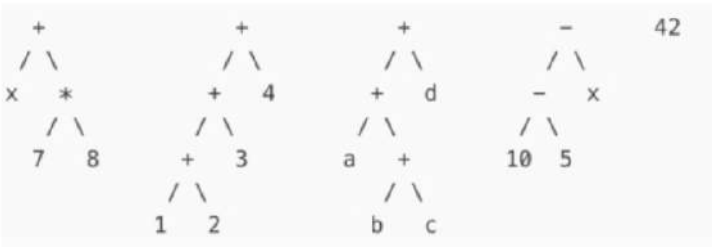
⟨nasprotni⟩ ::= - ⟨nasprotni⟩ | ⟨osnovni⟩

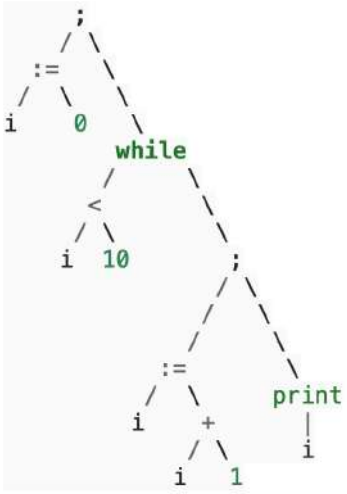
Iz konkretne v abstraktno sintakso

Dane izraze, zapisane v konkretni sintaksi, predstavi z abstraktno sintakso kot drevo. Pri tem veljajo običajna pravila za prioriteto in asociativnost aritmetičnih operacij.

1. x + 7 * 8
2. 1 + 2 + 3 + 4
3. a + (b + c) + d
4. 10 - 5 - x
5. 42
6. y + (5 + 7 * x) * 8
7.

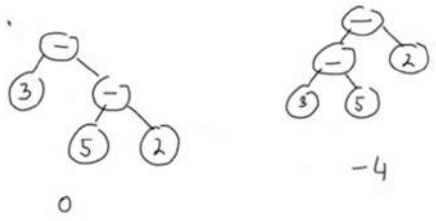
```
i := 0;
while i<10 do
   i := i+1 ;
   print i
done
```



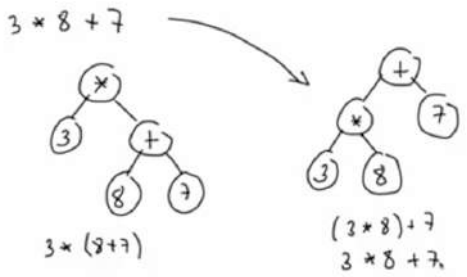


Napiši na dva načina (dokaži dvoumnost).

3 - 5 - 2

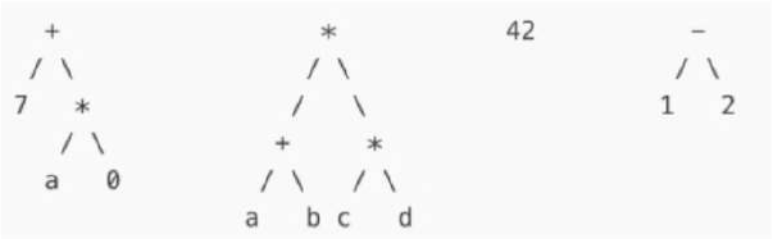


3 * 8 + 7



Iz abstraktne v konkretno sintakso

Izraze, predstavljene z abstraktno sintakso kot drevo, pretvori v konkretno sintakso. Upoštevaj običajna pravila za prioriteto in asociativnost aritmetičnih operacij.



Rešitev

7 + a * 0

(a + b) * (c * d)

42

1 - 2

Kvadriranje

Dana je abstraktna sintaksa aritmetičnih izrazov:

⟨izraz⟩ ::= ⟨aditivni-izraz⟩ EOF

⟨aditivni-izraz⟩ ::= ⟨multiplikativni-izraz⟩ | ⟨aditivni-izraz⟩ + ⟨multiplikativni-izraz⟩

⟨multiplikativni-izraz⟩ ::= ⟨osnovni-izraz⟩ | ⟨multiplikativni-izraz⟩ * ⟨osnovni-izraz⟩

⟨osnovni-izraz⟩ ::= ⟨spremenljivka⟩ | ⟨številka⟩ | (⟨izraz⟩)

⟨spremenljivka⟩ ::= [a-zA-z]+

⟨številka⟩ ::= -? [0-9]+

Jezik želimo razširiti s funkcijo sqr, ki sprejme argument in izračuna njegov kvadrat. Na primer, sqr(5) se izračuna v 25.

Popravite abstraktno sintakso tako, da bo jezik omogočal tudi izraze oblike sqr(e), kjer je e aritmetični izraz. Dana je semantika velikih korakov za aritmetične izraze:

$\eta \mid n \hookrightarrow n$

$\eta(x) = n$

$\eta \mid x \hookrightarrow n$

$\eta \mid e_1 \hookrightarrow n_1 \quad \eta \mid e_2 \hookrightarrow n_2$

$\eta \mid e_1 + e_2 \hookrightarrow n_1 + n_2$

$\eta \mid e_1 \hookrightarrow n_1 \quad \eta \mid e_2 \hookrightarrow n_2$

$\eta \mid e_1 - e_2 \hookrightarrow n_1 - n_2$

$\eta \mid e_1 \hookrightarrow n_1 \quad \eta \mid e_2 \hookrightarrow n_2$

$\eta \mid e_1 * e_2 \hookrightarrow n_1 \cdot n_2$

Semantiko popravite tako, da se bo vsebovala tudi ustrezno pravilo za sqr.

Med pravili za `multiplikativni-izraz` in `osnovni-izraz` dodamo pravilo za `kvadratni-izraz`. Popraviti moramo tudi pravilo za `multiplikativni-izraz`, tako da zamenjamo sklice na `osnovni-izraz`.

```
⟨izraz⟩ ::= ⟨aditivni-izraz⟩ EOF
⟨aditivni-izraz⟩ ::= ⟨multiplikativni-izraz⟩ | ⟨aditivni-izraz⟩ +
⟨multiplikativni-izraz⟩
⟨multiplikativni-izraz⟩ ::= ⟨kvadratni-izraz⟩ | ⟨multiplikativni-
izraz⟩ * ⟨kvadratni- izraz⟩
⟨kvadratni-izraz⟩ ::= sqr ( ⟨aditivni-izraz⟩ ) | ⟨osnovni-izraz⟩
⟨osnovni-izraz⟩ ::= ⟨spremenljivka⟩ | ⟨številka⟩ | ( ⟨izraz⟩ )
⟨spremenljivka⟩ ::= [a-zA-z]+
⟨številka⟩ ::= -? [0-9]+
```

Semantiko dopolnimo s pravilom za `kvadratni-izraz`:

$$\frac{\eta \mid e \hookrightarrow n}{\eta \mid \mathrm{sqr}(e) \hookrightarrow n^2}$$

$\eta = [x \mapsto 7,\ y \mapsto 2]$

$$\dfrac{\dfrac{\dfrac{}{\eta \mid 2 \hookrightarrow 2} \quad \dfrac{\eta(x)=7}{\eta \mid x \hookrightarrow 7}}{\eta \mid 2 * x \hookrightarrow 14}\ 2 \cdot 7 = 14 \quad \dfrac{}{\eta \mid 3 \hookrightarrow 3}}{\eta \mid 2 * x + 3 \hookrightarrow 17}\ 14 + 3 = 17$$

Izpiti

Sintaksa za zapis aritmetičnih izrazov:

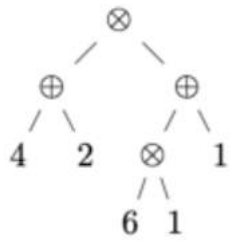
⟨izraz⟩ ::= ⟨številka⟩ | ⊖ ⟨izraz⟩ | ⊕ ⟨izraz⟩ ⟨izraz⟩ | ⊗ ⟨izraz⟩ ⟨izraz⟩
⟨številka⟩ ::= [0–9]+

Simboli ⊖, ⊕ in ⊗ označujejo nasprotno vrednost, seštevanje in množenje. Na primer, izraz

⊗ ⊕ 1 2 ⊕ ⊖ 6 20

ima vrednost 42.

Zapiši izraz, ki predstavlja sintaktično drevo:



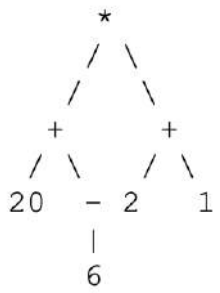
Izraz je lahko številka, - izraz, + izraz izraz ... Številka je neko zaporedje števk 0 do 9.
Operatorji so v prefiksni obliki.

V sintaksi se ne uporabljajo oklepaji.
Najboljše je, da gremo reševati od vrha navzdol po sintaktičnem drevesu.
Imamo ta primer:
Prvi operator je krat ... x
Na levi in na desni strani je izraz.
Na levi strani je plus, ki je spet izraz ... + 42 Na drugi strani imamo spet +.

Rekurzivno pogledamo in razpišemo.

Rešitev: (x)(+)42(+)(x)611

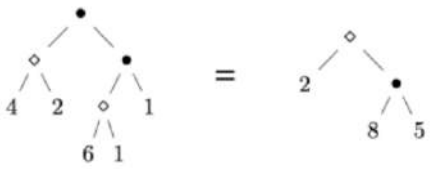
Zapiši sintaktično drevo, ki predstavlja izraz: $20\ 6\ \ominus\ \oplus\ 2\ 1\ \oplus\ \otimes$



Izraz, predstavljen z abstraktno sintakso kot drevo, pretvori v konkretno sintakso. Upoštevaj običajna pravila za prioriteto in asociativnost aritmetičnih operacij.

$7 + (a * 0)$

Kakšna je vrednost izraza predstavljenega z drevesom?



\+ ... štirikotnik * ... krog

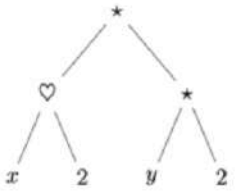
(4 + 2) * ((6 + 1) * 1) = 2 + (8 * 5)

42 = 42

Podana je sintaksa aritmetičnih izrazov.

⟨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⟩ | (⟨srčni-izraz⟩)
⟨spremenljivka⟩ ::= [a − zA − z]+
⟨številka⟩ ::= [0 − 9]+

Simbol † ima prednost pred ⋆, ki ima prednost pred ♡. Simbol ♡ je levo asociativen in ⋆ desno asociativen. Drevo



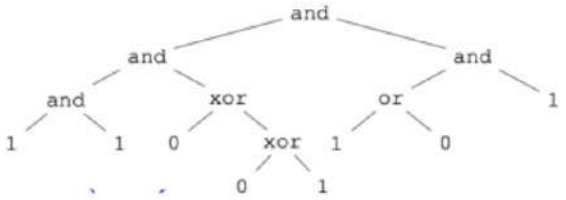
Zapišo drevo v konkretni sintaksi s čim manjšim številom oklepajev.

srce je zamenjano z +
(x + 2) * y * 2

Podana je sintaksa logični izrazov z uporabo and, or in xor. Zapiši izraz, ki je predstavljen z drevesom.

⟨expression⟩ ::= ⟨exclusive⟩ | ⟨exclusive⟩ or ⟨expression⟩
⟨exclusive⟩ ::= ⟨conjunctive⟩ | ⟨conjunctive⟩ xor ⟨exclusive⟩
⟨conjunctive⟩ ::= ⟨basic⟩ | ⟨basic⟩ and ⟨conjunctive⟩
⟨basic⟩ ::= (⟨expression⟩) | 0 | 1

right assoc.
right assoc.
right assoc.

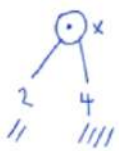


((1 and 1) and (0 xor 0 xor 1)) and (1 or 0) and 1

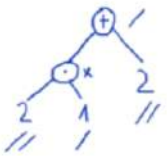
Elbonski direktorat za standarde je uvedel novo sintakso aritmetičnih izrazov. Vsa števila zapisujejo s poševnicami v eniškem sistemu, na primer ///// je število pet (nihče ni pomislil na število nič). Ljudstvo je bilo navadušeno, saj je v Elboniji poševnica znak za srečo. Direktorat je zato spremenil tudi zapis seštevanja in razglasil, da se namesto znaka + odslej za seštevanje uporabi poševnica /. Množenje so pisali s ×. Njihova nova sintaksa je torej naslednja:

<izraz> ::= <multiplikativni> | <izraz> / <multiplikativni>
<multiplikativni> ::= <število> | <multiplikativni> × <število>
<število> ::= /+
V državi sedaj vlada zmeda, zato so vas poklicali na pomoč. Direktorju direktorata morate pojasniti, da je možno nekatere izraze razčleniti na več načinov. V ta namen mu predočite izraz // × ////
Narišite dve različni drevesni predstavitvi zgornjega izraza, s katerima boste direktorju prikazali dvoumnost nove sintakse.

Prva različica:



Druga različica:



V Elbonji varčujejo črnilo, zato znaka za množenje ne pišejo, ampak namesto njega pustijo presledek. Operacijo seštevanja označijo s piko ·.
V ta namen uporabljajo naslednjo sintakso za zapis aritmetičnih izrazov, kjer označuje presledek:
⟨**izraz**⟩ **::=** ⟨**multiplikativni**⟩ | ⟨**izraz**⟩ · ⟨**multiplikativni**⟩
⟨**multiplikativni**⟩ **::=** ⟨**osnovni**⟩ | ⟨**osnovni**⟩ _ ⟨**multiplikativni**⟩
⟨**osnovni**⟩ **::= (**⟨**izraz**⟩**)** | ⟨**število**⟩
⟨**število**⟩ **::= [0–9]+**

Narišite sintaktično drevo za izraz 20 (4 · 2) 1 · 3 · (19 20).

Ugotoviti moramo, kje so presledki smiselni in kje ne, kaj se bo izračunalo prej, kje je leva in kje desna asociativnost.

Ne moremo imeti presledka kot operator in · eden zraven drugega, ker nikjer ne piše, da je za multiplikativni izraz lahko samo presledek. Kje presledki predstavljajo množenje?

$$20\_(4 \bullet 2)\_1 \bullet 3 \bullet (19\_20).$$

Asociativnost nam pove, katero množenje ima prednost. Množenje s presledkom ima prednost pred množenjem s piko.
Če imamo neke števke, ki se držijo skupaj predstavljajo neko število. Množenje: desno

* Desno drevo visi v levo.

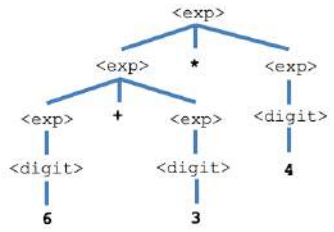
Poddrevesa gledamo kot 1 element.

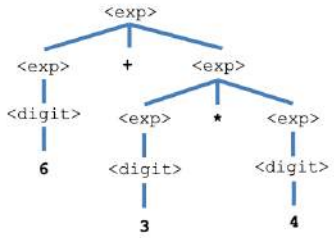
Primeri dvoumne gramatike

Dvoumna gramatika

```
<exp>    ::= <exp> + <exp> | <exp> * <exp> | ( <exp> ) | <digit>
<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

6 + 3 * 4





(6 + 3) * 4 6 + (3 * 4)

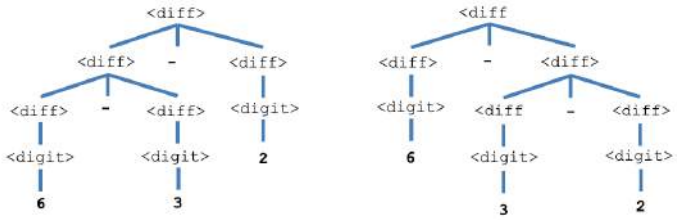
Problem je, da iz gramatike ne moremo razbrati prioritete operatorjev.

Nedvoumna gramatika

```
<exp>    ::= <exp> + <exp> | <term>
<term>   ::= <term> * <term> | <factor>
<factor> ::= ( <exp> ) | <digit>
<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Dvoumna gramatika

```
<diff>  ::= <diff> - <diff> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```




```
<diff>  ::= <diff> - <digit> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

DOKAZOVANJE PRAVILNOSTI PROGRAMOV

Kdaj podani program v ukaznem jeziku ustreza danim specifikacijam, ki jih podamo kot predpogoj in končni pogoj? Za dokazovanje uporabljamo Hoarovo logiko in pripadajoča pravila.

Implikacija

A	B	$A \Rightarrow B$
p	p	p
p	n	n
n	p	p
n	n	p

Kako dokažemo, da program ne ustreza dani specifikaciji?
Poiščemo protiprimer.

Delna pravilnost: ignoriramo ustavitev programa (neskončna zanka je ok), { P } c { Q }
Če imamo {, } in program ne ustreza specifikaciji in se ne ustavi, potem bo dana specifikacija za delno pravilnost še vedno veljala.
Če nimamo zanke: z delno pravilnostjo dokažemo tudi popolno

Popolna pravilnost: če velja P, se c ustavi in velja Q (ali se koda tudi ustavi), [P] c [Q]

Splošni napotki

Dokazujemo, da iz izjave `A` sledi izjava `D`. Pri tem si pomagamo z vmesnimi izjavami `B` in `C`.
Zapisali smo `A⟹B⟹C⟹D`. V tem zapisu ne mislimo na `A⟹(B⟹(C⟹D))` ampak na
`A⟹B ∧ B⟹C ∧ C⟹D`.

Velja:

- `x < y⟹ x ≤ y`
- $x < y \implies x + 1 \leq y$

Po stavku `x := N` vedno velja `{ x = N }`.

```
{ N = N }
   x := N
{ x = N }
```

Pravilo za prireditveni stavek je

```
{ P[x ↦ e] }
   x := e
   { P }
```

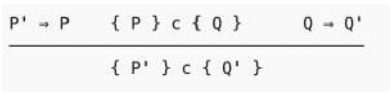
- iz vrha navzdol (problem pri prirejanju)

  Pri izpeljevanju najprej vse `x` v predpogoju izrazimo z `e`, nato pa `e` zamenjamo z `x`. Potrebno v eksplicitno izraz spremeniti, da so izrazi `e` izstopami (da lahko izraz `e` zamenjamo z `x`).

- od spodaj navzgor

  Če v končnemu pogoju `P` zamenjamo vse spremenljivke z imenom `x` z izrazom `e`, dobimo predpogoj. Iz tega pravila vidimo, da je bolj smiselno dokazovanje od spodaj navzgor.

Splošno pravilo



Če imamo 2 predpogoja in če iz prvega predpogoja `P'` sledi drugi predpogoj `P`, potem program še vedno ustreza tej specifikaciji. Podobno velja za končni predpogoj.

Primer

Ali velja implikacija `y < x => y ≤ x ? (y ≤ x) <=>`

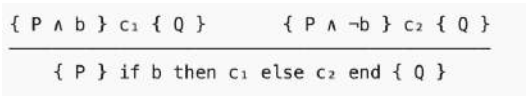
`y < x => {y < x || x = y}`

Ali se lahko zgodi, da je `true (y < x) => false (y ≤ x)?` Ne. Zato implikacija velja!

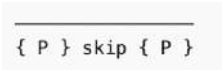
`P'` (strožji) => `P`

Vse rešitve levega pogoja so zajete v desnem pogoju (če je levi pogoj `true`, potem bo zagotovo tudi desni). V obratni smeri ne velja.

Pravilo za pogojni stavek



Pravilo za skip



Pravilo za zanko while

$$\frac{\{ P \wedge b \}\ c\ \{ P \}}{\{ P \}\ \texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}\ \{ \neg b \wedge P \}}$$

V **zančni invarianti** se zančni pogoj in lokalne spremenljivke ne pojavljajo, saj mora invarianta veljati tudi po izstopu iz zanke. Ko invarianto »peljemo« čez telo zanke, se nam v njej lahko pojavijo lokalne spremenljivke. Teh se želimo znebiti, ko dosežemo konec zanke.
Kdaj mora izraz, ki je zančna invarianta veljati?

- pred zanko
- ko pridemo v zanko
- na koncu telesa zanke
- po zanki

  lahko invarianta **prešibka** => ne moremo izpeljati končnega pogoja => v invarianto dodamo še kak pogoj in ga ločeno peljemo čez zanko

  če velja $A \implies C$, potem velja tudi $A \wedge B \implies C$

Kako poiskati invarianto (z njo želimo dokazati končni pogoj Q):

- **najmočnejša**

  Naredimo tabelo, da vidimo, kako se spreminjajo oz. ne spreminjajo spremenljivke. Potrebujemo še nek pogoj, ki pove še nekaj o velikosti spremenljivk (while pogoj, zgornja meja).

- **najšibkejša**

  `true` => s tako invarianto si težko pomagamo

Ko dokazujemo, uvajamo nove predpogoje. Dokaze razčlenimo na več vmesih pogojev in jih združujemo z pravilom za zaporedje ukazov.
Če iz originalnega predpogoja lahko sklepamo na naš predpogoj, končamo z dokazovanjem.

Popolna pravilnost
Spremeniti bo potrebno samo pravilo za while zanko.

$$\frac{[\ P \wedge b \wedge e = z\ ]\ c\ [P \wedge e < z\ ] \qquad z \notin FV(c)}{[\ P\ ]\ \texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}\ [\ \neg b \wedge P\ ]}$$

Spremeni se znotraj zanke - poiskati je potrebno količino e, ki se ob vsaki iteraciji zmanjša (ne sme se v nedogled). To količino poiskusimo pridobiti iz invariante (da bo navzdol omejena).
Ne iščemo spremenljivke, ampak poljuben aritmetični izraz.

Naloga 1

`{false} c {false}` // vedno velja (ker iz `false` sledi karkoli)

`{true} c {false}` // program `c` se ne ustavi ... če velja `true` in če se bo `c` ustavil, potem bo veljal `false` (vendar `false` nikoli ne velja)

`{false} c {true}` // vedno velja (ker iz `false` sledi karkoli)

`{true} c {true}` // vedno velja (ker iz `true` sledi iz česarkoli)

`[false] c [false]` // vedno velja (ker iz `false` sledi iz česarkoli)

`[true] c [false]` // nikoli ne velja ... `false` ne more veljati, če se `c` ustavi

`[false] c [true]` // vedno velja (ker iz `false` sledi iz česarkoli)

`[true] c [true]` // program `c` se ustavi ... `c` se bo ustavil in velja `true` ; če velja `true`, potem se `c` ustavi in velja `true`

Naloga 2

Dokaži pravilnost programa:

```
{  x ≤ 7 }
x := x + 3
{ x ≤ 10 }
```

1. možnost: od spodaj navzgor

```
{  x ≤ 7 } //  druga vrstica sledi iz prve (logično sklepanje)
{x + 3 ≤ 10 } //  P(e)
x := x + 3 //  e je izraz x + 3
{ x ≤ 10 } //  P(x) je izjava x ≤ 10
```

Ali lahko iz `{x ≤ 7}` pridemo do `{x + 3 ≤ 10}`? Logično sklepanje.

2. možnost: od zgoraj navzdol

Gremo po programu in `x` zamenjamo s tistim, kar bi moral biti in razmišljamo, kako bi to popravili.
Ko vidimo `x := x + 3`, poskrbimo, da se v prejšnji vrstici ponavlja samo desna stran `x + 3` => desno stran nato zamenjamo z `x` in dobimo pravo stvar.

```
{  x ≤ 7 } => logični sklep
{ ( x + 3) - 3 ≤ 7 }
 x := x + 3
{x - 3 ≤ 7 } => logični sklep
{ x ≤ 10 }
```

Naloga 3

Dokaži pravilnost programa:

{ x ≤ y }

s := (x + y) / 2

{ x ≤ s ≤ y }

{ x ≤ y }

s := (x + y) / 2

{ x ≤ y ∧ s = (x + y) / 2}

{ x ≤ (x + y) / 2 ≤ y ∧ s = (x + y) / 2} => logični sklep

{ x ≤ s ≤ y }

Naloga 4

{ x = m ∧ y = n }

x := x + y;

y := x - y;

x := x - y

{ x = n ∧ y = m }

1. možnost: od spodaj navzgor

{ x = m ∧ y = n } => *2

{x + y - (x + y - y) = n ∧ x + y - y = m } <=> {y = n ∧ x = m}

x := x + y;

{x - (x - y) = n ∧ x - y = m }

y := x - y;

{x - y = n ∧ y = m }*1

x := x - y

{ x = n ∧ y = m }

*1 Spremenimo predpogoj { x = n ∧ y = m }, da ustreza pravilu za prirejanje? (uvedenemo nov predpogoj)

{ x = n ∧ y = m } <=> { P[x ↦ e] } <=> {x - y = n ∧ y = m }

x := x - y

{ x = n ∧ y = m } <=> { P }

V izjavi oz. končnem predpogoju P moramo zamenjati vse x z e, da dobimo veljavno Hoarovo trojico.

e = x - y

*2 Iz prve vrstice oz. predpogoja { x = m ∧ y = n } skušamo sklepati na drugo vrstico, ki smo jo dobili z dokazovanjem od spodaj navzgor. Dobimo celo ekvivalenco.

Če iz originalnega predpogoja lahko sklepamo na naš predpogoj, končamo z dokazovanjem.

{y = n ∧ x = m} Q_3

{x - (x - y) = n ∧ x - y = m } Q_2

{x - y = n ∧ y = m } Q_1

Dokaze razčlenimo na več vmesnih pogojev Q. Na koncu skušamo sklepati iz originalnega predpogoja P na naš predpogoj Q_3. Na pamet je težko ugibati, kaj so te vmesni predpogoji. Zato je potrebno izbrati strategijo.

2. možnost: od zgoraj navzdol

{ x = m ∧ y = n } <==> P

{(x + y) - y = m, y = n} P_1

x := x + y;

{x - y = m, y = n}

{x - y = m, x - y = n} P_2

y := x - y;

{y = m, x - y = n} P_3

x := x - y

{y = m, x = n} P_4

=> logični sklep

{ x = n ∧ y = m } <==> Q

Naloga 5

Dokažite parcialno in popolno pravilnost programa glede na dano specifikacijo.

```
{ }
if y < x then
  z := x;
  x := y;
  y := z
else
  skip
end
{ x ≤ y }
```

Parcialna pravilnost

Prazen predpogoj `{}` je vedno `true`.

Uporabimo strategijo od spodaj navzgor.

1. korak: postavitev pogojev: `{true}`, `{y < x}` in `{y ≥ x}`
2. korak: dokazovanje

```
{ true }
if y < x then
 { y < x } => *2
 { y ≤ x }
  z := x;
 { y ≤ z }
  x := y;
 { x ≤ z }
  y := z
 { x ≤ y }
else
  {y ≥ x} => *1
  { x ≤ y }
  skip
{ x ≤ y }
end
{ x ≤ y }
```

`*1`

Dokazovali od spodaj navzgor in prišli do pogoja `{y ≥ x}`.

Ali velja implikacija `{ ¬(y < x) } == {y ≥ x} => { x ≤ y }`?

Da. Imamo celo ekvivalenco.

Nadaljujemo z dokazovanje od psodaj navzgor.

`*2`

Ali velja implikacija `{ y < x } => { y ≤ x }`

`{ y < x } => {y < x || x = y}`

Levi pogoj je strožji od desnega => vse rešitve levega pogoja so zajete v desnem pogoju (če je levi pogoj true, potem bo zagotovo tudi desni). V obratni smeri ne velja!

Protiprimer za: `{ y ≤ x } => { y < x }`

`y = x: T => F ... F`

<u>Popolna pravilnost</u>

Sledi iz parcialne, ker ni nobene `while` zanke.

Naloga 6

Sestavite program c, ki zadošča specifikaciji

[n ≥ 0]

c

[s = 1 + 2 + ... + n]

```
{ n ≥ 0 }
s := 0 ;
i := 0 ;
while i <= n do
      s := s + i ;
      i := i + 1
done
{ s = 1 + 2 + ... + n }
```

Invarianta: P = {s = 1 + 2 + ... + (i - 1) ∧ i ≤ n + 1}

Popolna pravilnost: n - i = z

Delna pravilnost

Ko dokazujemo program, ki ima zanko `while`, je pomembno da izberemo primerno zančno invarianto P. To je izraz, ki velja (se ne sme spremeniti): 1) pred zanko, 2) ko pridemo v zanko, 3) na koncu telesa zanke in 4) po zanki.

1. korak: zapišemo vse 4 pogoje, dokazujemo do začetka zanke

```
{ n ≥ 0 }
s := 0 ;
{ n ≥ 0, s = 0 }
i := 0 ;
{ P, n ≥ 0, s = 0, i = 0 }
while i <= n do
{ P, i <= n }
      s := s + i ;
      i := i + 1
      { P }
done
{ P, i > n } => *1
{ s = 1 + 2 + ... + n }
```

2. korak: izberemo primerno zančno invarianto

*1

Z { P, i > n } želimo dokazati Q = { s = 1 + 2 + ... + n }, kar pomeni da mora izbrati tak P, da bo med njima implikacija oz. da bomo sklepali na Q.

Pomagamo si lahko s tem, kje je umeščena zanka while: med {n ≥ 0 ∧ s = 0 ∧ i = 0} in { s = 1 + 2 + ... + n }.

Ali je v redu invarianta P = {i = n + 1}? Ne, ker ne velja za vsako iteracijo!

Ali je v redu invarianta P = { s = 1 + 2 + ... + i }?

i je v tem primeru minimalno n + 1 => Q ne bo veljal (ni v redu invarianta)

Rešitev: namesto i zapišemo (i - 1)

Z {i > n ∧ s = 1 + 2 + ... + (i - 1) ∧ i ≤ n + 1} lahko sklepamo na končni pogoj.

Združimo pogoj i > n in i ≤ n + 1 => i = n + 1

{s = 1 + 2 + ... + (i - 1) ∧ i = n + 1}

Na začetku je dobro izbrati tako invarianto, ki velja na začetku zanke in iz katere lahko sklepamo na končni pogoj.

Izbrana invarianta: P = {s = 1 + 2 + ... + (i - 1) ∧ i ≤ n + 1}

3. korak: Nadaljujemo z dokazovanjem, pri tem upoštevamo invarianto.

```
{ n ≥ 0 }
s := 0 ;
{ n ≥ 0, s = 0 }
i := 0 ;
{ n ≥ 0, s = 0, i = 0 } =>
I: { s = 1 + 2 + ... + (i - 1), i ≤ n + 1}
while i <= n do
      I: { s = 1 + 2 + ... + (i - 1), i ≤ n + 1, i ≤ n} => *2
      { s = 1 + 2 + ... + (i - 1) , i ≤ n } <=>
      { s + i = 1 + 2 + ... + i , i ≤ n }
      s := s + i ;
      { s = 1 + 2 + ... + i , i ≤ n } <=>
      { s = 1 + 2 + ... + (i + 1 - 1), i + 1 ≤ n + 1 }
      i := i + 1
      I: { s = 1 + 2 + ... + (i - 1), i ≤ n + 1 }
done
{ i < n, s = 1 + 2 + ... + (i - 1), i ≤ n + 1} <=>
I: { s = 1 + 2 + ... + (i - 1), i = n + 1} =>
{ s = 1 + 2 + ... + n }
```

*2

Pogoj `i <= n` je bolj strog, zato je pogoj `i ≤ n + 1` brez pomena.

Popolna pravilnost

$$\frac{[\ P \wedge b \wedge e = z\ ]\ c\ [P \wedge e < z\ ] \qquad z \notin FV(c)}{[\ P\ ]\ \texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}\ [\ \neg b \wedge P\ ]}$$

Poiskati je potrebno količino `e`, ki se ob vsaki iteraciji zmanjša (ne sme se v nedogled).
To količino poiskusimo pridobiti iz invariante (da bo navzdol omejena).
Ne iščemo spremenljivke, ampak poljuben aritmetični izraz.

Za količino `e` izberemo: `n - i`.
Na začetku je ta količina enaka `z`-ju (ghost variable): `n - i = z`
Ali je količina navzdol omejena? Da, `i ≤ n + 1`. Omejena z - 1.
Dokazati moramo, da bo na koncu veljalo `e < z`.

```
[ s = 1 + 2 + ... + (i - 1) ∧ i ≤ n + 1 ∧ n - i = z ]
while i <= n do
      [ s = 1 + 2 + ... + (i - 1), i ≤ n + 1, i ≤ n, n - i = z ]
      [ s = 1 + 2 + ... + (i - 1), i ≤ n, n - i = z ]
      [ s  + i = 1 + 2 + ... + i, i ≤ n, n - i = z ]
       s := s + i ;
      [ s = 1 + 2 + ... + i, i ≤ n, n - i = z ]
      [ s = 1 + 2 + ... + i + 1 - 1, i + 1 - 1 ≤ n, n - (i + 1 - 1) = z ]
     i := i + 1
      [ s = 1 + 2 + ... + i - 1, i - 1 ≤ n, n - (i - 1) = z ]
       [ s = 1 + 2 + ... + i - 1, i ≤ n + 1, n - (i - 1) = z ]
      [ s = 1 + 2 + ... + i - 1, i ≤ n + 1, n - i = z - 1 < z ] =>
       [ s = 1 + 2 + ... + i - 1, i ≤ n + 1, n - i < z ]
```

Dokazali smo, da se `n - i` zmanjša na vsakem koraku. Na začetku `n - 1 = z`, po sami izvedbi telesa while zanke pa dobimo `n - i < z`.

Naloga 7

```
{ x ≥ 0 }
y := 0;
z := x;
while 1 < z - y do
  s := (y + z)/2;
  if s * s < x then
    y := s
  else
    z := s
  end
done
{ y² ≤ x ≤ (y + 1)² }
```

Invarianta: $y^2 \leq x \leq z^2$

Popolna: $z - y = e,\ z - y > 0$

Delna pravilnost

Kaj je invarianta?

Iz $\{1 \geq z - y,\ P\}$ moramo sklepati na $\{ y^2 \leq x \leq (y + 1)^2 \}$.

$\{P,\ 1 \geq z - y\} <=> \{P,\ 1 + y \geq z\} <=?=> \{ y^2 \leq x \leq (y + 1)^2 \}$

Najlažje je, da naredimo tabelo lokalnih spremenljivk.

X = 15	Y	Z	Z-y (pogoj)	s
15	0	15	15	NA
15	0	7	7	7
15	3	7	4	3
15	3	5	2	5
15	3	4	1	4

Iz tabele je razvidno, da y in z predstavljata spodnjo in zgornjo mejo `floor(sqrt(x))`. Spremenljivka s pa predstavlja sredino.

Invarianta: $y^2 \leq x \leq z^2$

x ne more biti neposredno v invarianti.

V vsakem koraku dokazovanja moramo dokazati, da velja originalna invarianta (glej odebeljeno). Vzamemo samo tiste pogoje, ki so potrebni. Ko na nekem mestu dokažemo invarianto, ni potrebno nesti vseh pogojev naprej.

Dokazovanje od spodaj navzgor

```
{ x ≥ 0 }
y := 0;
{ x ≥ 0, y = 0 }
z := x;
{ x ≥ 0, y = 0, z = x} =>
{ y² ≤ x ≤ z² }
while 1 < z - y do
{ y² ≤ x ≤ z², 1 < z - y } <=>
{ y² ≤ x ≤ z², 1 < z - y, (y + z) / 2 = (y + z) / 2 }
  s := (y + z)/2;
  { y² ≤ x ≤ z², 1 < z - y, s = (y + z)/2 }
  if s * s < x then
   { y² ≤ x ≤ z², 1 < z - y, s = (y + z)/2, s * s < x } =>
   { x² ≤ z², s² < x } => (če s² < x potem velja tudi s² ≤ x}
   { s² ≤ x ≤ z² }
   y := s
  { y² ≤ x ≤ z² }
  else
  { y² ≤ x ≤ z², 1 < z - y, s = (y + z)/2, s * s ≥ x } =>
  { y² ≤ x, s * s ≥ x } =>
  { y² ≤ x ≤ s²} =>
    z := s
  { y² ≤ x ≤ z² }
  end
{ y² ≤ x ≤ z² }
done
{ y² ≤ x ≤ z², 1 ≥ z - y } <=> { y² ≤ x ≤ z², z ≤ y + 1 } =>
{ y² ≤ x ≤ (y + 1)² }
```

Popolna pravilnost

Opomba

Pri dokazovanju lahko spustimo druge pogoje, ki se ne nanašajo na `[ z - y = e, z - y > 0 ]`. Dokazati moramo, da se količina `e` zmanjšuje.

```
[ z - y = e, z - y > 0 ]
s := (y + z)/2;
[ z - y = e, z - y > 0, s = (y + z)/2 ]
if s * s < x then
      [ z - y = e, z - y > 0, s = (y + z)/2] # y = 2s - z
      [ z - (2s - z) = e, z - (2s - z) > 0 ]
      y := s
      [ z - (2y - z) = e, z - (2y - z) > 0 ]
      [ 2(z - y) = e, 2(z - y) > 0 ]
      [ z - y = e/2, z - y > 0 ] // dokazali
      else
      [ z - y = e, z - y > 0, s = (y + z)/2] # z = 2s - y
      [ 2s - y - y = e, 2s - y - y > 0 ]
       z := s
      [ 2z - y - y = e, 2z - y - y > 0 ]
      [ 2(z - y) = e, 2(z - y) > 0 ]
      [ z - y = e/2, z - y > 0 ]
      end
[ z - y = e/2, z - y > 0 ]
```

Ker je `e/2 < e`, se količina `z - y` res zmanjšuje.

Naloga 8

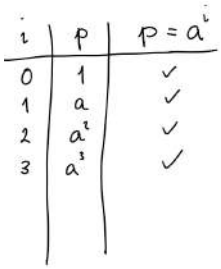
```
{ b ≥ 0 }
i := 0 ;
p := 1 ;
while i < b do
      p := p * a;
      i := i + 1
done
{p = a^b}
```

Invarianta: `p = a ^ i, i ≤ b`

Popolna pravilnost: `b - i ≥ 0`

Delna pravilnost

Invarianta



Katera količina povezuje spremenljivke, ki se spreminjajo? Količina, ki se ne spreminja: `p = a ^ i`

Izjava, ki pove nekaj o velikosti spremenljivk (zgornje meje): `i ≤ b`

```
{ b ≥ 0 }
i := 0 ;
{ i = 0, b ≥ 0}
p := 1 ;
{ i = 0, b ≥ 0, p = 1} =>
{ p = a ^ i, i ≤ b}
while i < b do
      { p = a ^ i, i ≤ b, i < b } <=> { p = a ^ i, i < b } => ?* =>
      { p * a = a ^ (i + 1), i + 1 ≤ b} <=> {p = a ^ i, i + 1 ≤ b}
      p := p * a;
      { p = a ^ i + 1, i + 1 ≤ b}
      i := i + 1
      { p = a ^ i, i ≤ b}
done
{ p = a ^ i, i ≤ b, i ≥ b } <=> { p = a ^ i, i = b} =>
{p = a^b}

*
i < b <=> i + 1 < b + 1 => i + 1 ≤ b
```

Popolna pravilnost

Dokazati moramo, da se zmajšuje količina `e = b - i ≥ 0`.

```
[ p = a ^ i, i ≤ b, e = b - i ]
while i < b do
      [ b - i = e ]
      p := p * a;
      [ b - i = e ]
       [ b - (i + 1) + 1 = e ]
      i := i + 1
      [ b - i + 1 = e ]
      [ b - i = e - 1 < e ]
```

Začnemo z `b - i = e`, na koncu pa imamo `b - i < e`.

Naloga 9

```
[x = m ∧ y = n]
if y < x then
   x := x + y ;
   y := x - y ;
   x := x - y
else
   skip
end
[ x = min(m, n) ∧ y = max(m, n) ]
```

Delna pravilnost

```
{ x = m ∧ y = n }
if y < x then
 { y < x, x = m, y = n }
 { n < m, x = m, y = n }
 { y = n = min(m, n), x = m = max(m, n)}
 { x + y - (x + y - y) = min(m, n), x + y - y = max(m, n) }
   x := x + y ;
  { x - (x - y) = min(m, n), x - y = max(m, n) }
   y := x - y ;
  { x - y = min(m, n), y = max(m, n) }
   x := x - y
  { x = min(m, n), y = max(m, n) }
else
   { y ≥ x, x = m, y = n }
   { n ≥ m, x = m, y = n }
   skip
     { n ≥ m, x = m, y = n } =>
      { x = min(m, n), y = max(m, n) }
end
{ x = min(m, n), y = max(m, n) }
```

Popolna pravilnost

Ker ni zanke, popolna pravilnost sledi iz delne pravilnosti. Zamenjamo le { } z [ ].

Naloga 10

Pojasni, kaj o programu P pove izjava

`[ true ]`

`P`

`[ false ]`

Program `P` se ustavi, po koncu izvajanja pa velja `false`. Tak program ne obstaja.

Naloga 11

Dokaži delno pravilnost programa:

```
{x > 2}
y := x + 1
if x > y then
    z := x * x - y * y
else
    z := y * y - x * x
end
{1 ≤ z}
```

```
{ x > 2 }
y := x + 1
{ x > 2, y = x + 1 }
if x > y then
   { x > 2, y = x + 1, x > y }
    z := x * x - y * y
   { z = x² - y², x > 2, y = x + 1, x > y } #A
else
   { x > 2, y = x + 1, x ≤ y }
    z := y * y - x * x
   { z = x² - y², x > 2, y = x + 1, x ≤ y } <=>
   { z = x² - (x + 1)², x > 2 } <=>
   { z = x² - x² + 2x + 1, x > 2 } <=>
   { z = 2x + 1, x > 2 } <=>
   { z > 5, x > 2 } #B
end
```

// veljati morata #A in #B

```
{A => z ≥ 1}: false
{B => z ≥ 1}: true
{1 ≤ z}
```

Naloga 12

Dokaži delno pravilnost programa:

```
{ 0 ≤ x ∧ 0 ≤ y ∧ 0 ≤ z}
if x < y then
      if x < z then
            m := x
      else
            m := z
      end
else
      if y <= z then
            m := y
      else
            m := z
      end
end
{ 0 ≤ m ≤ x}
```

```
{ 0 ≤ x ∧ 0 ≤ y ∧ 0 ≤ z }
if x < y then
      { 0 ≤ x, 0 ≤ y, 0 ≤ z, x < y }
      if x < z then
            { 0 ≤ x, 0 ≤ y, 0 ≤ z, x < y, x < z }
            m := x
            { 0 ≤ x, 0 ≤ y, 0 ≤ z, x < y, x < z, m = x} =>
            { 0 ≤ m ≤ x }
      else
            { 0 ≤ x, 0 ≤ y, 0 ≤ z, x < y, x < z, x ≥ z }
            m := z
            { 0 ≤ x, 0 ≤ y, 0 ≤ z, x < y, x < z, x ≥ m } =>
            { 0 ≤ m ≤ x }
      end
      { 0 ≤ m ≤ x }
else
      { 0 ≤ x, 0 ≤ y, 0 ≤ z, x ≥ y }
      if y <= z then
      { 0 ≤ x, 0 ≤ y, 0 ≤ z, x ≥ y, y ≤ z }
            m := y
            { 0 ≤ x, 0 ≤ m, 0 ≤ z, x ≥ m, m ≤ z } =>
            { 0 ≤ m ≤ x }
      else
      { 0 ≤ x, 0 ≤ y, 0 ≤ z, x ≥ y, y > z }
            m := z
            { 0 ≤ x, 0 ≤ y, 0 ≤ m, x ≥ y, y > m } =>
            { 0 ≤ m ≤ x }
      end
{ 0 ≤ m ≤ x }
end
{ 0 ≤ m ≤ x }
```

Naloga 13

Dokažite delno pravilnost programa:

```
{0 < a ∧ 0 < b}
k := 0
while not (a * k = b) do
      k := k + 1
done
{b = a * k}
```

Ali velja tudi popolna pravilnost? (Se pravi, da `{ ... }` nadomestimo z `[ ... ]`.) Odgovor utemeljite.

Delna pravilnost

```
{0 < a ∧ 0 < b}
k := 0
while not (a * k = b) do
      k := k + 1
done
{b = a * k}
```

Negacija zančnega pogoja je enaka končnemu pogoju: če se zanka izteče, mora veljati končni pogoj.

Popolna pravilnost

Popolna pravilnost ne velja; program se zacikla, če `a` ne deli `b`.

Naloga 14

```
[0 < a ∧ 0 < b]
k := 0
while a * k < b do
      k := k + 1
done
[ a * (k - 1) < b ≤ a * k ]
```

Zančna invarianta:

Popolna pravilnost:

Delna pravilnost

```
{ 0 < a ∧ 0 < b }
k := 0
{P
while a * k < b do
      k := k + 1
done
{ a * (k - 1) < b ≤ a * k }
```

Popolna pravilnost

Popolna pravilnost ne velja; program se zacikla, če `a` ne deli `b`.

Naloga 15

Dokažite popolno pravilnost programa:

```
[a < b]
x := a ;
y := b ;
while x < y do
      x := x + 1 ;
     y := y - 1
done
[ a + b ≤ 2x ≤ a + b + 1 ]
```

Invarianta: $\{ x + y = a + b,\ x \leq y + 1\}$

Popolna pravilnost: $e = y + 1 - x,\ y + 1 - x < 0$

Delna pravilnost

```
[ a < b ]
{ a < b }
x := a ;
{ a < b, x = a}
y := b ;
{ a < b, x = a, y = b }
{ x + y = a + b, x ≤ y + 1} => Q
while x < y do
      { x + y = a + b, x ≤ y + 1, x < y} =>
      { x + y = a + b, x < y}
      { x + 1 + y - 1 = a + b, x + 1 < y + 1}
      x := x + 1 ;
      { x + y - 1 = a + b, x < y + 1}
      { x + y - 1 = a + b, x < y - 1 + 2}
     y := y - 1
      { x + y = a + b, x < y + 2}
      { x + y = a + b, x ≤ y + 1} => Q
done
{ x + y = a + b, x ≤ y + 1, y ≤ x }
{ x + y = a + b, y ≤ x ≤ y + 1 }
{ x + y = a + b, y + x ≤ x + x ≤ x + y + 1 } =>
[ a + b ≤ 2x ≤ a + b + 1 ] Q
```

Popolna pravilnost

Invarianta: { x + y = a + b, y + x ≤ x + x ≤ x + y + 1 }

```
e = y + 1 - x
y + 1 - x < 0

{ x + y = a + b, x ≤ y + 1 }
{ y + 1 - x < 0, e = y + 1 - x }
while x < y do
{ y + 1 - x < 0, e = y + 1 - x, x < y }
{ -x < - y - 1, e = y + 1 - x, x < y }
{ x < y + 1, e = y + 1 - x, x < y }
{ e = y + 1 - x, x < y }
{ e = y + 1 - (x + 1) + 1, x + 1 - 1 < y }
x := x + 1 ;
{ e = y + 2 - x, x - 1 < y }
{ e = y + 2 - x, x < y + 1 }
{ e = (y - 1) + 2 + 1 - x , x < (y - 1) + 2 }
 y := y - 1
{ e = y + 3 - x , x < y + 2 }
{ y + 1 - x = e - 2, 0 < y + 2 - x }
{ y + 1 - x < e, 0 ≤ y + 1 - x }
done
```

Naloga 16

Dokažite delno pravilnost programa:

```
{n > 0}
s := 0 ;
k := 0 ;
a := 1 ;
while k <= n do
      s := s + a ;
      a := a * n ;
      k := k + 1
done
{ (n – 1)· s = a – 1 }
```

Dokažite še popolno pravilnost, se pravi, utemeljite, da se zanka while pri danih predpostavkah vedno zaključi.

Invarianta: $(n - 1) \cdot s = a - 1$

Popolna pravilnost: $n - k = e,\ k - 1 \leq n$

Delna pravilnost

```
{ n > 0 }
s := 0 ;
k := 0 ;
a := 1 ;
{ n ≥ 0, s = 0, k = 0, a = 1 } =>
{ (n – 1)· s = a – 1 }
while k <= n do
      { (n – 1)· s = a – 1 }
      { (n – 1)· (s + a - a) = a – 1 }
      s := s + a ;
       { (n – 1)· (s - a) = a – 1 }
      { (n – 1)· s - (n – 1)· a = a – 1 }
      { (n – 1)· s - na + a = a – 1 }
      { (n – 1)· s = n · a – 1 }
      a := a * n ;
      { (n – 1)· s = a – 1 }
      k := k + 1
      { (n – 1)· s = a – 1 }
done
{ (n – 1)· s = a – 1 }
```

Popolna pravilnost

```
k ≤ n ... k - 1 ≤ n

{ n - k = e, k - 1 ≤ n }
while k <= n do
{ n - k = e, k ≤ n, n - k ≥ 1 }
      s := s + a ;
      a := a * n ;
      { n - (k + 1 - 1) = e, (k + 1 - 1) ≤ n, n - (k + 1) ≥ 1 }
      k := k + 1
      { n - (k - 1) = e, k - 1 ≤ n, n - k ≥ 1 }
      { n - k = e - 1, n - k ≥ 1 }
```

Količina `n - k` se je zmanjšala in je omejena navzdol.

Naloga 17

Dokažite delno pravilnost programa:

```
{x ≤ y}
c := y ;
if z <= x then
      a := z ;
      b := x ;
else
      a := x ;
      if z <= y then
            b := z
      else
            b := y ;
            c := z
      end
end
{a ≤ b ∧ b ≤ c}
```

```
{x ≤ y}
c := y ;
{ x ≤ y, c = y }
if z <= x then
      { x ≤ y, c = y, z ≤ x }
      a := z ;
      { x ≤ y, c = y, a ≤ x }
      b := x ;
      { b ≤ c, a ≤ b }
else
{ x ≤ y, c = y, z > x }
      a := x ;
      { a ≤ y, c = y, z > a}
      if z <= y then
      { a ≤ y, c = y, z > a, z ≤ y}
            b := z
            { a ≤ y, c = y, b > a, b ≤ y}
            { a ≤ y, b > a, b ≤ c}
            { a ≤ b, b ≤ c}
      else
      { a ≤ y, c = y, z > a, z > y}
            b := y ;
            { a ≤ b, c = b, z > a, z > b}
            { a ≤ b, z > a, z > b}
            c := z
            { a ≤ b, z > a, z > b, c = z}
            { a ≤ b, c > a, c > b}
            { a ≤ b, b ≤ c}
      end
      { a ≤ b, b ≤ c}
end
{a ≤ b ∧ b ≤ c}
```

Naloga 18

Poženemo 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 => 0, d => 3, k => 42]`. Kakšno je končno okolje, ko se program konča:

42 mod 2 = 0, d = 4

21 mod 2 = 1,

10 mod 2 = 0, d = 5

5 mod 2 = 1

4 mod 2 = 0, d = 6

2 mod 2 = 0

Rešitev: [a = > 0, d => 6, k => 0]

Naloga 19

Sestavimo program P:

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

Označi vse specifikacije, ki jim zadošča program.

a) `[n = 3] P [true]`

Pomeni, da se pri danem predpogoju bo ukaz `P` končal.

To je res, saj: 3 => 10 => 5 => 16 => 8 => 4 => 2 => 0.

b) `{true} P {n = 1}`

Pomeni, da če se bo ukaz `P` končal, potem bo veljalo `{ n = 1 }`.

Za vse predpogoje pri katerih se ukaz `P` konča, bo veljal končni pogoj `{ n = 1 }`.

To ni res, protiprimer je `{ n < 1 }`, npr. `[n ↦ -7]`.

Protiprimer mora biti podan kot začetno stanje, iz katerega je razvidno, da specifikacija ne velja.

c) `{n = 0} P {n = 1}`

Če velja `{ n = 0 }` in se bo ukaz `P` končal, potem bo veljalo `{ n = 1 }`. To ne drži, ker se bo program P končal (zanka se ne bo izvedla) in vrednost `n`-ja bo 0.

Naloga 20

Dokaži delno pravilnost programa

```
{b > 1}
i := 2;
j := 0;
while j < b do
      i := i + i + i - 2;
      j := j + 1;
end
{i = 3^b + 1}
```

Invarianta: $i = 3^j + 1,\ j \leq b$

Spreminja se i in j; za j moramo postaviti mejo.

$\{b > 1\}$

$i := 2;$

$j := 0;$

$\{\ b > 1,\ i = 2,\ j = 0\ \} \Rightarrow$

$\{\ i = 3^j + 1,\ j \leq b\ \}$

while $j < b$ do

$\{\ j < b,\ i = 3^j + 1\ \}$

$\{\ j < b,\ 3i = 3 * 3^j + 3\}$

$\{\ j < b,\ i + i + i - 2 = 3^{j+1} + 1\ \}$

$i := i + i + i - 2;$

$\{\ j < b,\ i = 3^{j+1} + 1\ \}$

$\{\ j + 1 \leq b,\ i = 3^{j+1} + 1\ \}$

$j := j + 1;$

$\{\ j \leq b,\ i = 3^j + 1,\ j \geq b\}$

$\{i = 3^j + 1,\ j = b\} \Rightarrow$

$\{i = 3^b + 1\}$

end

$\{i = 3^b + 1\}$

Program se ustavi, ker $b - j$.

Naloga 21

Dokažite delno pravilnost programa. Iz vaše rešitve naj bo jasno razvidna invarianta zanke while.

```
{true}
x := a;
i := 0;
while i < 100 do
      x := x * x * a;
      i := i + 1;
end
{x = a^(2^101 - 1)}
```

Dokažite še polno pravilnost programa. Iz vaše rešitve naj bo jasno razvidno, katera količina zagotavlja zaustavitev zanke while.

Invarianta:

$\{ x = a^{2^{i+1}-1}, i \leq 100\}$

Na začetku mora biti $x = a$

če je $x = a^{2^i} \Rightarrow i = 0 \Rightarrow x = 1$

$i \leq 100$, da velja tudi po zanki

Popolna pravilnost:

$e = 101 - i, 101 - i > 0$

Delna pravilnost

```
{ true }
x := a;
{ x = a }
i := 0;
{ x = a, i = 0 } =>
{ x = a^(2i + 1) - 1 , i ≤ 100 }
while i < 100 do
      { i < 100, i ≤ 100, x = a^(2i + 1) - 1 } =>
      { i < 100, x = a^(2i + 1) - 1 }
      { i < 100, x * x * a = a^(2i + 2) - 1 }
      x := x * x * a;
      { i < 100, x = a^(2i + 2) - 1 }
      { i + 1 ≤ 100, x = a^(2i + 1 + 1) - 1 }
      i := i + 1;
      { i ≤ 100, x = a^(2i + 1) - 1 }
end
{ i ≤ 100, i ≥ 100, x = a^(2i + 1) - 1 } =>
{ i = 100, x = a^(2i + 1) - 1 } =>
{ x = a^(2101) - 1 }
{ x = a^(2101) - 1 }
```

$x^2 a = (a^{i+1} - 1)^2 a =$
$= (a^{2^{i+2}} - 2) a =$
$= a^{2^{i+2}} - 1$ ✓

Popolna pravilnost

```
e = 101 - i, 101 - i > 0
while i < 100 do
      { e = 101 - i, 101 - i > 0 }
      x := x * x * a;
      { e = 101 - i, 101 - i > 0 }
      { e = 101 - (i + 1 - 1), 101 - (i + 1 - 1) > 0 }
      i := i + 1;
      { e = 101 - (i - 1), 101 - (i - 1) > 0 }
      { e = 102 - i, 102 - i > 0, i ≤ 100 }
      { e - 1 = 101 - i, 102 - i > 0, i ≤ 100 }
      { e ≥ 101 - i, 101 - i > 0}
end
{x = a^(2^101 - 1)}
```

LAMBDA RAČUN

- http://plzoo.andrej.com/language/lambda.html

  namesto λ je treba pisati ^ ali \; ukaze ločimo s ;
- https://projectultimatum.org/cgi-bin/lambda?t=(λx.λy.y)%20((λx.x%20x)%20(λx.x%20x))

Teorija

V Lambda računu je vse funkcija: argumenti, ki jih sprejme funkcija so funkcije, prav tako rezultati

Gramatika

t ::=		*terms:*
	x	*variable*
	λx.t	*abstraction*
	t t	*application*

Osnovni Lambda račun pozna samo 2 operaciji - **aplikacija** in **redukcija**.

- **abstrakcija**: naredimo funkcijski predpis (extend as far to the right)

  Nek izračun napišemo bolj generično.

  Izraz `x + y` izračuna vsoto spremenljivk `x` in `y`.

  Običajna matematična notacija za izraz je `f(x) = x + y` oz. `f : (x |-> x + 1)`

  Izraz zapišemo kot funkcijo spremenljivke `x` z uporabo lambda abstrakcije `λx.(x + y).`

  Razlika med zapisoma: lambda notacija funkcije nima imena, v matematični notaciji pa običajno zapišemo tudi ime funkcije `f`.

  Iz izraza `x + 2y + x` naredimo funkcijo dveh spremenljivk `x` in `z` z lambda izrazom:

  `λx.λz.(x + 2*y + x)`

  Pred `.` pridejo argumenti funkcije, za `.` pa telo funkcije oz. aplikacija (uporaba funkcije).

  Lambda račun nima funkcij z večimi argumenti vendar lahko isti učinek dosežemo s funkcijami višjega reda.

  Transformacija, ki funkcijo z večimi argumenti prevede v funkcijo višjega reda, imenujemo Cur

  Primer 1: `true := λxy.x;`

  `:= λx.λy.x;`

  1. argument sprejmemo kot argument funkcije
  2. argument sprejmemo tako, da prva funkcija vrne funkcijo, ki sprejme 2. argument in potem telo 2. argumenta odloči, kako bo te dva argumenta uporabil

Primer 2

Naj bo `M` izraz s prostima spremenljivkama `x` in `y`. Želimo napisati funkcijo `F`, ki za vsak par `(N, L)` zamenja `x` z `N` in `y` z `L` v izrazu `M`.

V osnovnem lambda računi ne moremo napisati `F = λ(x, y).M`.

Rešitev: `F = λx.λy.M`

`F` je funkcija, ki ob danem argumentu `N` vrne funkcijo, ki ob dani vrednosti za `y` vrne željeni rezultat.

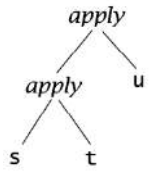
- **aplikacija**: aplikacija neke funkcije (aplikacije nad funkcijskimi predpisi) (levo asocirana)

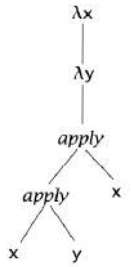
  Aplikacija lambda izraza na vrednost, ki je podana kot parameter se izvrši podobno kot pri matematičnih funkcijah. Spremenljivko povezano z lamda zamenjamo za vrednost.

  `(λx.(λz.(x + 2 * y + z))) 5 4 = 4 + 2 * y + 5`

  povezovanje: `λx.f(fx) = λx.(f(f(x)))`

  `stu = (st)u`





  `λx. λy. x y x = λx.(λy.((xy)(x))`

  `f ~ (x -> f x)`

  `sin ~ (x -> sin(x)) ~ (x -> sinx)`

  sprejme `x` in aplicira sinus nad `x`

- **redukcija (substitucija)**: evaluacija lambda izrazov

  Zamenjamo vse spremenljivke v danem izrazu z nekim izrazom, ki je bodisi sestavljen izraz ali vrednost. Preimenovanje vezanih spremenljivk!

Naloge

Povezovanje

λx.λy.x z y = λx.(λy.((x z) y))

(λx.λz.x z) y = (λx.(λz.(x z)))y

λx.λy.λz.(x z)(y z) = λx.(λy.(λz.((x z)(y z))))

λm.λn.λz.λs.m (n z s) s = λm.(λn.(λz.(λs.((m ((n z) s)) s))))

Evulacijske strategije

(λxyz.xyz)(λx.xx)(λx.x)x

// abc ... (ab)c ... ni pomemben vrstni red računanja

= (((λxyz.xyz)(λx.xx))(λx.x)x)

= (λxyz.xyz)(λx.xx)

// spreminjanje imen vezanih spremenljivk

= (λxyz.xyz)(λx'.x'x')

= (λyz.(λx'.x'x')yz)

= (λyz.((λx'.x'x')y)z)

= (λyz.(yy)z) // normalna oblika

= ((λyz.(yy)z)(λx.x))x)

= ((λyz.(yy)z)(λx.x))

= ((λz.((λx.x)(λx.x))z)

= (λz.(λx.x)z)

= (λz.z)

= (λz.z)x

= x

(λ f x . f (f x)) (λ z . z z)

= (λ f . λ x . f (f x)) (λ z . z z)

= (λ x . λ z . z z) (λ z . z z) x))

= (λ x . λ z . z z)

= λ x . x x (x x)

(λ f g x . f (g x)) (λ z . t (t z)) t

= (λ f. λ g. λ x . f (g x)) (λ z . t (t z)) t

= (λ g. λ x . λ z . t (t z) (g x) t

= λ x . λ z . t (t z) (t x)

= λ x . t (t (t x))

((λ x . x) (λ f g x . g)) u v w

= ((λ x . x) (λ f. λ g. λ x . g)) u v w

= (λ f. λ g. λ x . g) u v w

= (λ g. λ x . g) v w

= (λ x . v) w

= v

Izračuaj f f in f g, kjer f := λ x . x x in g := λ y . y y y.

f f

(λ x . x x)(λ x . x x)

= (λ x . x x) (λ x . x x)

= (λ x . x x) (λ x . x x) (λ x . x x) (λ x . x x)

= ...

f g

(λ x . x x) (λ y . y y y)

= (λ y . y y y) (λ y . y y y)

= (λ y . y y y)(λ y . y y y)(λ y . y y y) = ...

Neskončno dolg izraz.

Definiramo λ-izraze

a := λ f x . f(f(f x)) ;

b := λ g y . g(g y) ;

Kateremu izrazu je enak izraz a b c d?

a b c d =

= (λf.λx.f(f(fx))(λg.λy.g(gy))cd

= λx.((λg.λy.g(gy))((λg.λy.g(gy))((λg.λy.g(gy))x)))cd

= (λg.λy.g(gy))((λg.λy.g(gy))((λg.λy.g(gy))c))d

= (λg.λy.g(gy))((λg.λy.g(gy))(λy.c(cy)))d

= (λg.λy.g(gy))(λy.(λy.c(cy))(λy.c(cy)y))d

= (λg.λy.g(gy))(λy.(λy.c(cy))(c(cy)))d

= (λg.λy.g(gy))(λy.c(c(c(cy))))d

= (λy.(λy.c(c(c(cy))))(λy.c(c(c(cy)))y))d

= (λy.(λy.c(c(c(cy))))(c(c(c(cy)))))d

= λy.c(c(c(c(c(c(cy))))))d

= c(c(c(c(c(c(cd))))))))

V λ-računu definiramo izraza `K := λxy.x`, `S := λxyz.(xz)(yz).`

a) S K K

b) S K S

SKK

(λ x . λ y. λ z . (x z) (y z)) (λ x. λ y . x) (λ x. λ y . x)

= (λ y. λ z . ((λ x. λ i . x) z (y z)) (λ x. λ y . x)

= (λ z . ((λ x. λ i . x) z ((λ x. λ i . x) z))

= (λ z . (λ i . z) (λ x. λ i . x) z)

= (λ z . (λ i . z) (λ i . z)

= λ z . z

SKS

(λx.λy.λz.(xz)(yz))(λx.λy.x)(λx.λy.λz.(xz)(yz))

=(λy.λz.xzyz)(λx.λy.λz.xzyz)

=λz.xzyz

Kaj izračuna funkcija h?

S := λxyz.(xz)(yz).

(i) seznam zadnjih elementov vseh nepraznih seznamov danega seznama seznamov

(ii) seznam praznih seznamov danega seznama seznamov

(iii) seznam prvih elementov vseh nepraznih seznamov danega seznama seznamov

(iv) Haskell zavrne definicijo, ker ta vsebuje vsebuje napako

Funkcija h izračuna (i) seznam zadnjih elementov vseh nepraznih seznamov danega seznama seznamov.

V λ-računu evaluiramo izraz

(λfx.f(fx))(λf.ff)(λx.x)

Kateri izraz dobimo?

(λf.λx.f(fx))(λf.ff)(λx.x)

= (λf.λx.f(fx))(λx.x)(λx.x)

= (λf.λx.f(fx))(λx.x)

= (λf.λx.f(fx))(λy.y)

= λx.(λy.y)((λy.y)x))

= λx.(λy.y)(x)

= λx.x

Boolove vrednosti in logični vezniki

`True` in `false`

`true` in `false` sta podatka, ki nam povesta, kako se odločimo med dvema možnostma.

Nanju lahko gledamo kot na funkciji, ki sprejmeta obe možnosti `x` in `y`, nato pa vrneta tisto, ki sta jo izbrala.

```
if true x y = x
      true := λxy.x;
           := λx.λy.x;
           := λx_.x;
      (x |-> (y |-> x))
```

`True` je funkcija, ki sprejme 2 argumenta; 2. argument ignorira in vrne 1.

`X` slikamo v funkcijo, ki sprejme `y` in vrne `x`.

```
if false x y = y
      false := λxy.y;
            := λ_y.y;
```

`False` je funkcija, ki sprejme 2 argumenta; 1. argument ignorira in vrne 2.

```
if := λ p x y . p x y;
```

Funkcija, ki sprejme 3. argumente (pogoj - `true` ali `false`, 1. veja pogojnega stavka (if), 2. veja pogojnega stavka (else)).

Vrne 2. argument, če je 1. argument `true`. Vrne 3. argument, če je 1. argument `false`.

Enostavno aplicira pogoj `p` na obeh možnostih, pogoj `p` pa izbere pravo.

Konstanta

Definiramo konstanti (to sta tudi funkciji!)

```
:constant T;
:constant F;

constT := λx.T;
const := λa.λx.a;
```

Če sprejmemo `a`, bo rezultat funkcija, ki sprejme poljubno vrednost in vrača `a`.

bool

Funkcija, ki vrne konstanto `T`, če za vhod dobi `true` in `false`, če za vhod dobi `F`.

Sprejme samo 1 argument.

```
if true T F;
T
if false T F;
F

bool := λp. if p T F;
```

and

Funkcija, ki sprejme 2 logični vrednosti in se obnaša kot logični veznik `and`.

`And` vrne `false`, takoj ko je eden izmed argumentov `false`.

```
and false false ≡ false
and true false ≡ false
and false true ≡ false
and true true ≡ true

Java: if (p) q; else false;
```

Za 2. argument zapišemo `y`.

`and true y ≡ y` // če je 1. argument `true`, potem 1. argument nima nobenega vpliva

`and false y ≡ false`

Kako s pomočjo `if` stavka preverimo, ali sta oba argumenta `true`?

```
and := λab.if a(if b true false)(false);
```

() ... predstavlja vejo

```
and := λa.λb.if a b false;
```

`And` je funkcija, ki sprejme 2 argumenta. Če je 1. argument `true`, potem vrnemo 2., če pa je 1. argument `false`, vrnemo `false`.

Če upoštevamo definicijo `if`, lahko to še poenostavimo:

```
and := λ a b . a b false
```

Ta definicija neposredno uporablja dejstvo, da je logična vrednost `a` funkcija, ki izbere eno od dveh možnosti. V izrazu `a b false` tako `x` izbere bodisi `b` bodisi `false`. Če je `a ≡ true`, potem izbere `b`. Če pa je `a ≡ false`, potem izbere `false`.

not

```
not := λ x. if x false true;
not := λ x. x false true;
```

Uporabimo `if` stavek. Če je argument `a true`, vrnemo `false`, drugače `true`.

or

```
or false false ≡ false
or true false ≡ true
or false true ≡ true
or true true ≡ true
```

Če je 1. argument funkcije `x` resničen, potem vrnemo `true`. Drugače vrnemo 2. argument.

```
or := λ x y . if x true y;
or := λ x y . x true y;
```

imply

```
imply true true ≡ true
imply true false ≡ false
imply false true ≡ true
imply false false ≡ true
```

```
imply := λ x y . if x y true ;
imply := λ x y . x y true ;
```

Če je 1. argument `true`, vrnemo `y`.

Če je 1. argument `false`, vrnemo `true`.

iff

```
iff true true ≡ true
iff true false ≡ false
iff false true ≡ false
iff false false ≡ true
```

```
iff := ^ x y . and (a imply b) (b imply a) ;
```

Urejeni pari

```
pair := λa.λb.λp. p a b;
```

Želimo

```
fst(pair x y) = x // vrne 1. komponento
snd(pair x y) = y // vrne 2. komponento
```

```
fst := λp. p true
    := λp. p (λx.λy.x)
snd := λp. p false
    := λp. p (λx.λy.y)
```

Churchova števila

Število n je funkcija, ki sprejme funkcijo f in vrne n-kratni kompozitum f:

f ∘ f ∘ ... ∘ f

Sprejme 2 argumenta f in x in aplicira f n-krat na argument x (funkcija, ki naredi nekaj n-krat oz. naredi n-aplikacij nad 2. argumentom).

0' := λ f x . x ; // (kot false)

1' := λ f x . f x ; // funkcijo f nad x aplicira 1x

2' := λ f x . f (f x) ;

3' := λ f x . f (f (f x)) ;

4' := λ f x . f (f (f (f x))) ;

5' := λ f x . f (f (f (f (f x))));

Število izrazimo s prejšnjim

Funkcijo f n-krat uporabimo na argumentu x: n f x

6' := λ f x . f (5' f x) ;

Potrebno narediti še dodatno aplikacijo. Vsako naslednje število je samo dodatna aplikacija te funkcije.

6' := λ f x . (5' f x) ; // ni ok

5' f x dobimo aplikacijo f(f(f(f(fx)))). Predstavlja 5 aplikacij f nad x.

Potenciranje (kompozitum)

op := λ x y . y x;

Sprejme argumenta x in y ter vrne aplikacijo x nad y.

f ∘ f ∘ f ∘ ... = f^n (večkratna aplikacija funkcij)

2' : f |-> f ∘ f

3' : (g |-> g ∘ g ∘ g)

2' 3' (f |-> f ∘ f)(g |-> g ∘ g ∘ g) =

= (g |-> g ∘ g ∘ g) ∘ (g |-> g ∘ g ∘ g) =

g^3

$g^3 \circ g^3 = g^9$

Scott-Churchova števila

Število `n` predstavimo kot `n`-kratno uporabo funkcije `f`, dodatno podamo še število oz. števec, ki predstavlja število opravljenih aplikacij.

2. argument pove, koliko aplikacij `x` smo naredili nad `f`.

```
0 := ^ f x . x ;
1 := ^ f x . f 0 x; // nobene aplikacije
2 := ^ f x . f 1(f 0 x);
3 := ^ f x . f 2 (f 1 (f 0 x)) ;
4 := ^ f x . f 3 (f 2 (f 1 (f 0 x))) ;
5 := ^ f x . f 4 (f 3 (f 2 (f 1 (f 0 x)))) ;
```

Število izrazimo s prejšnjim

Številu pošljemo funkcijo, ki predstavlja prejšnjo število in nad njiimapliciramo z `f` in `x`.

Torej funkciji `f` pošljemo prejšnje število in aplicirano prejšnje število.

```
6 := ^ f x. f 5 (5 f x)
1' := ^ f x . f 0' (0' f x)
```

Naslednik števila `n`

Sprejme število `n` in vrne naslednika od `n`, ki je tudi Scott-Churchovo število in sprejme 2 argumenta `f` in `x`.

Kaj je potrebno narediti? Dodatno aplikacijo `f`-ja nad prejšnjim številom.

Nad prejšnjim številom `n` apliciramo `f` in `x`, da dobimo rezultat.

```
naslednik := λ n f x . f n (n f x)
```

Najprej naredimo `n` aplikacij `f` nad `x`, nato smo še eno aplikacijo.

Primitivna rekurzija

Ko imamo podano definicijo primitivne rekurzije, jo lahko direktno implementiramo v lambda račun z uporaba Scott-Churchovih števil.

Imamo robni pogoj in želimo računati funkcijo `g`.
Primitivna rekurzija omogoča en rekurzivni klic. Rezultat rekurzivnega klica lahko na vsakem koraku spremeni neko funkcijo `f`, ki poleg rezultata rekurzivnega klica sprejme še višino rekurzije (na najnižjem nivoju je 0). Višina rekurzije 0 pomeni, da smo na robnem pogoju.

Implementirali bi radi rekurzivno funkcijo `g`, ki pri

- argumentu 0 vrne vrednost `x`

  `g 0 = x` // g(0) = x
- argumentu `n + 1`

  naredi en rekurzivni klic na predhodniku `n`

  `g (n + 1) = f n (g n)`

Pomožna funkcija `f` sprejme 2 argumenta (`n` in rezultat rekurzivnega klica `g n`) in izračuna končni odgovor.
Prvi argument `n` poveča za 1 (`n + 1`).
Taki rekurziji pravimo **primitivna ali strukturna rekurzija** (poznamo tudi splošno rekurzijo, v kateri lahko g naredi povsem poljubne rekurzivne klice).

Factoriela

Izračunamo s pomočjo primitivne rekurzije (sprejme celo število in izračuna produkt od 1 do `n`)

`n! = 1 · 2 · ... · n`

`fact 0 = 1` // robni pogoj

`fact (n + 1) = (n + 1) · fact n`

Rekurzijo spremenimo v primitivno rekurzijo

`g(n + 1) = f n (g n)`

`fact (n + 1) = f n (fact n)`

Ugotoviti moramo, kaj moramo vstaviti za pomožno funkciji `f`.
Funkcija `f` sprejme 2 argumenta: prvi argument `n` poveča za 1, drugi argument (rezultat rekurzije) `r` pusti pri miru in ga pomnoži s prvim argumentom.

f := λ k r . (k + 1) · r

Church-Scottova števila na vsakem koraku vodijo evidenco na kakšni globini smo.

Primitivno rekurzijo spremenimo v λ-račun

`fact := λ n . n f 1`

Faktoriela sprejme Church-Scottovo število kot argument - nad njim apliciramo funkcijo `f` in robni pogoj (pri 0 vrnemo 1).

Vstavimo pomožno funkcijo `f`

`fact := λ n . n (λ k r . (k + 1) · r) 1`

Scott-Churchova števila v splošnem omogočajo računanje funkcij, ki so definirane s primitivno rekurzijo. Če je `g` definirana kot

`g 0 = x` // robni pogoj

`g (n + 1) = f n (g n)`

jo s Scott-Churchovimi števili definiramo kot

`g := λ n . n f x`

`g := λ n . n (λ m r . ...) x`

`n` ... število

`x` ... baza rekurzije (začetna vrednost)

`f` je `λ m r` ... pomožna funkcija, ki pove, kako iz rezultata rekurzivnega klica `r` pri argumentu `m` izračunamo rezultat pri `m + 1` (števec, ki ga lahko uporabimo)

`r` ... rezultat rekurzije oz. dosedanjih aplikacij

Funkcija `g` sprejme C-S število in nad njim apliciramo funkcijo, ki sprejme dva argumenta `m` in `r`.

Predhodnik

Primitivno rekurzivna definicija:

```
predhodnik 0 = 0 // robni pogoj
predhodnik (n + 1) = n

predhodnik 0 = 0
predhodnik (n + 1) = (λ m r . m) n (predhodnik n)
// g (n + 1) = f n (g n)
```

Za pomožno funkcijo `f` izberemo funkcijo, ki ignorira rekurzivni klic, števec ostane tak kot je. Sprejme dva argumenta `m` in `r`, drugi argument zavrže.

```
// g := λ n . n (λ m r . ...) x
f := λ m r . m // ignorira rezultat aplikacij in vrne dejanski števec
predhodnik = λ n . n (λ m r . m) 0

predhodnik 5
5 := ^ f x . f 4 (f 3 (f 2 (f 1 (f 0 x)))) ;
```

Začetna vrednost je 0. Na vsakem koraku bi funkcija `f` zavrgla prejšnjo aplikacijo in ko bi izvedli zadnjo aplikacijo.

Definiramo funkcijo `show`, ki število `n` spremeni v `n`-kratno uporabo konstante `S` na konstanti `Z`

```
:constant S // naslednik
:constant Z // 0
show 0 = Z
show (n + 1) = S (show n)
```

Kaj je povezovalna funkcija `f`? Sprejme dva argumenta in sicer trenutni števec `n` in `show` od `n`.
Ignorira prvi argument. Nad rezultatom rekurzije aplicira `S`.

```
show(n + 1) = f n (show n)
f := (λ m r . S r)
```

Naredimo primitivno rekurzijo z C-S števili.

```
show := λ n . n (λ m r . S r) Z ;
```

Sprejme C-S število `n` in vrne aplikacijo števila `n` nad `f`, začetna vrednost je `Z`.
Toliko kot je aplikacij `f`-jev, toliko bo dodanih `S`-jev.
Za začetno vrednost vzamemo `Z`. Nad njim izvedemo `f`: trenutni števec zavržemo, nad rekurzivnim klicem pokličemo `S`.
Dobimo nekaj aplikacij `S` nad `Z`.

```
lambda> show (pred 4)

S ((λ _ r . S r) ((λ _ r . S r) 0 Z) 1)
```

Nato še vklopimo neučakano računanje, da bo lambda vse izračunala do konca:

```
lambda> :eager
I will evaluate eagerly.
lambda> show (pred 4)
S (S (S Z))
```

Funkcija naslednik

`naslednik 0 = 1`

`naslednik (n + 1) = (n + 2)` // rekurzivni klic zavržemo

Problem je, da nimamo definicije seštevanja.

Funkcijo naslednik implementiramo neposredno, izhajajoč iz definicije Scott-Churchovih števil:

`naslednik n = λ f x. f n(f(n - 1)(...f 1 (f 0 x)...)`

`naslednik n = λ f x. f n(n f x)`

Funkcija sprejme `n`. Na desni strani je predpis. Na levi strani želimo imeti samo ime. `n` moramo spraviti na drugo stran. Narediti moramo funkcijo, ki sprejme argument in vrača predpis.

To naredimo takole: `naslednik = λn. λ f x . f n (n f x)`

Imamo predpis, ki sprejme `n` in ga slika v nov predpis. Za nek `n` smo vrnili funkcijo, pri čemer `n` zamenjamo z neko vrednostjo.

Pri predhodniku smo imeli funkcijo, ki vrne `m`.

```
predhodnik := λ n . n (λ m r . m) 0 ;
```

Namesto, da `m` zmanjšamo za 1 oz. ga ne spremenimo, moramo povečati za 2; `r` je rezultat rekurzivnega klica, ki ga ignoriramo.

Pokličemo 2 x krat `naslednik`.

```
naslednik := λ n . n (λ m r . m) 1 ;
naslednik := λ n . n (λ m r . successor (successor m)) 1 ;
```

ali pravilnejše

```
naslednik := λ n . λ f x . f n (n f x) ;
```

Seštevanje

```
0 + k = k
naslednik n + k = naslednik (n + k)

plusK (n + 1) = f n (plusK n)

f := λ m r . naslednik r // števec ignoriramo

plus 0 k = k
plus (n + 1) k = naslednik (plus n k)

plus 0 k = k
plus (n + 1) k = (λ m r . naslednik r) n (plus n k)
```

Argument `k` nastopa kot konstanta, 1. argument pa kot spremenljika oz. števec.

Funkcija `plus` sprejme 2 argumenta in pri robnem pogoju vrne `k`.

V splošnem pa za rekurzivni korak vzame kar naslednika od vsote.

V vsakem koraku rekurzije se prišteje 1, trenutni števec pa zavržemo.

```
+ := λ k . λn. n (λ m r . naslednik r) k;
```

Funkcijo `f` lahko definiramo kot naslednika od rekurzivnega klica, `m` pa lahko ignoriramo.

Rezultat rekurzije pa na vsakem koraku (števila izvajajo aplikacijo) prištejemo trenutnemu rezultatu 1.

`n` sprejme funkcijo `f` in rezultat rekurzivnega klica `r`, ki ga vrnemo v robnem primeru - `k` (obnaša kot konstanta in ni del rekurzije)

`k` bo začetna točka in okoli nje bomo kot čebulo dajali sloje čez in število se bo povečevalo, dokler `n` ne bo 0.

Množenje

`0 · k = 0`

`(succ n) · k = k + n·k`

Aksioma moramo zapisati v obliki neke funkcije (primitivne rekurzije), ki pomnoži vhodni argument oz. število z številom `k`.

Uporabimo "recept" kako strukturno rekurzijo pretvorimo v lambda račun s pomočjo Scott-Churchovih števil.

V splošnem

`* := λ n k . n f x`

`x` je rezultat rekurzije oz. rezultat robnega pogoja (izhodiščna točka, ki je 0) - funkcija pri robnem pogoju vrne 0.

`* := λ n k . n f 0`

Kaj bi bila povezovalna funkcija `f` v tem primeru? 2 argumenta.

Plus

2 argumenta: `m` (števec) in `r` (rezultat rekurzije)

`f`: (`λ m r . naslednik r`)

Zakaj sprejme 2 argumenta?

Scott-Churchova števila so funkcije, ki sprejmejo funkcijo in neko začetno izhodišče (v splošnem kar `x`, neka spremenljivka, ki ni definirana). `F` sprejme prejšnje Scott-Churchovo število in aplikacijo nad njim. `m` je višina rekurzije (nad prvim Scott-Churchovim številom v tej rekurziji, ni bilo narejene nobene aplikacije nad njim, lahko gledamo kot konstanto).

```
n f x
4 f 0
4
```

4 je funkcija `f` in `x`. `F` je začetna točka s katero začnemo, nato večkrat apliciramo `f` nad tem `x`-som (večkrat rekurzivno pokličemo `f`). V tem primeru se `f` rekurzivno pokliče 4-krat (4 pove globino rekurzije). `f` je funkcija, ki opiše rekurzijo (rekurzivni klic) in sprejme 2 argumenta - prejšnji rezultat rekurzivnega klica in število na vsakem koraku, ki ga lahko uporabimo. Pri rekurzivnem klicu vemo, na kateri višini smo:

pri 1. koraku je višina 0: `f 0 x`

na 2. koraku: `f 1` in rezultat rekurzije - aplikacija `f` nad 0 in `x`) ... `(f 1 (f 0 x))`

ponavljamo ...

4 pomeni samo število rekurzivnih klicev, `f` pa je kako povežemo rekurzivne klice med seboj.

Če `f` definiramo, se bo ta izraz v nekaj izračunal pri čemer podamo začetno izhodišče.

```
λ f x . f 3 (f 2 (f 1 (f 0 x)))
```

Izračunamo vsoto vseh naravnih števil do 4.

Kako napišemo `f`:

0 -- število, ki ga izberemo za `x` (s katerim začnemo)

```
4 (λ x y. + x y)0
```

Kaj vrne funkcija?

1. rekurziven klic: `f` bo + => `0 + 0 => 0 => f` bo spet seštevanje

Vse f-je zamenjamo s plusom in `x` zamenjamo z 0.

Kaj dobimo? 0, potem bo rezultat vsega 1 (0 prištejemo 1), vse do 4 (rekurziven klic tudi upoštevamo). Na koncu prištejemo 3.

```
* := λ n k . n (λ m r . + k r) 0 ;
```

n ... `n` - krat izvedemo funkcijo, `n` je Scott-Churchovo število, ki sprejme `f` in začetni pogoj (0)

`f` ... sprejme 2 argumenta

Potrebujemo rekurzivni klic, rekurzivnemu klicu prištejemo `k` + rekurzivn klic (`r`).

`m` ... trenutna višina/globina rekurzije (tukaj m-ja ne potrebujemo, ker vedno prištevamo isto; iterator)

`m = 0` ... robni pogoj

Odštevanje

`n - m` je enak `0`, če je `m` večji od `n`.

```
n - 0 = n
n - (k + 1) = predhodnik(n - k)
```

V katerem argumentu bo rekurzija? V 2. argumentu, `n` pa bo konstanta.

Tokrat imamo primitivno rekurzijo na 2. argumentu, saj se v rekurzivnem klicu zmanjša `k`.

Če bo 2. argument večji od 1., vrnemo 0.

Robni pogoj: če odštejemo 0, vrnemo kar `n` (začnemo z `n` in na vsakem koraku odštejemo 1)

Ko zmanjka `k`, je konec (for loop, gremo od 0 do `k` in vedno odštejemo 1).

Ko od 0 odštejemo 1 (predhodnik od 0 je 0), potem vračamo samo 0.

Kaj je število, ki nam predstavlja "zanko"? `k`, `n` je konstanta.

Kaj je robni pogoj? Začnemo z rezultatom robnega pogoja.

Rekurzija: najprej se pokliče rekurzija, napolne se sklad, izvajanje v obratnem vrstnem redu => zanka se izvaja nazaj; začne se pri `n` (začne se od zadaj).

```
- := λ n k . k (λ m r . predhodnik r ) n ;
```

`n` ... robni pogoj

Od česa vzamemo predhodnika? Od rekurzivnega klica. `m` lahko ignoriramo, ker ga ne potrebujemo

`n` ... s čim štartamo (od zadaj)

`k` ... kolikrat se izvede

Ta funkcija ima na vsakem koraku na voljo indeks.

Primerjava z 0

```
iszero 0 = true
iszero (n+1) = false
```

Tudi to je primitivna rekurzija, čeprav rekurzivnega klica ne vidimo (mislimo si, da je bil rekurzivni klic zavržen):

```
iszero 0 = true
iszero (n + 1) = (λ m r . false) n (iszero n)
```

`n` lahko zmanjšamo za 1 in prenesemo delo na rekurzivni klic `iszero` => pogledamo kakšen je rezultat `iszero`

na levi strani imamo `n + 1`, na desni strani pa `n`

kako pridemo iz `iszero n` na `iszero (n + 1)`? funkcija, ki to povezuje je `f`

Za 0 je `true`, za vse drugo, kar je večje od 0 bo `false`.

Če uporabimo Scott-Churchova števila, ki ne ignorirajo funkcije, bo takoj `false`.

0 ignorira funkcijo (ne aplicira funkcije, je v nekem smislu identiteta; sprejme 2 argumenta in 1. ignorira).

Kaj funkcija naredi? Funkcija `f` je konstanta, vedno vrača `false`.

```
iszero := λ n . n (λ m r . false) true ;
```

Ignoriramo `m`, ker ga ne potrebujemo (števec), ne potrebujemo tudi `r`-ja - če se bo funkcija poklicala 1 x krat, smo imeli število večje od 0 - false.

Če je `iszero`, začnemo z `n`-jem.

Za katerikoli `n`, ki ni 0 bo to `false`. Zakaj? Ker se funkcija `f` vsaj enkrat poklicala in ta funkcija vrne `false`.

Izvede se `n`-krat in vsakič bo vrnila `false`, razen takrat ko se ne izvede (`n == 0`).

Relacija ≤

Relacijo `a ≤ b` lahko sestavimo s pomočjo odštevanja in `iszero`, ker velja `a ≤ b ⇔ a - b = 0`

Upoštevali smo dejstvo, da smo definirali `a - b = 0`, če je `a < b`.

Definiramo funkcijo ≤, tako da je `≤ a b` enako `true`, če je `a` manjši ali enak `b`, sicer pa `false`.

Uporabimo že prej definirani funkciji in `iszero`.

```
<= := λ a b . iszero(- a b) ;
```

Če bo 1. večje od 2., potem je to ekvivalentno temu, vrnemo 0.

Če vrne 1. argument, `true`.

Če vrne 2. argument, `false`.

Relaciji < in =

Definiramo funkcijo <, ki računa relacijo "strogo manjše" ter funkcijo ==, ki ugotovi, ali sta števili enaki.

`a < b ⇔ a + 1 ≤ b`

Kdaj bo enako?

Če odštejemo `a` in `b` ter če odštejemo `b` in `a`, bi morali dobiti enako. Oboje hkrati mora biti 0. Uporabimo `and`, ki sprejme 2 argumenta.

```
== := λ a b . and (<= a b) (<= b a);
-- a < b ⇔ a + 1 ≤ b
< := λ a b . and (<= a b) (not (== a b));
<' := λ a b . <= (succ a) b ;
```

Iskanje števila, ki zadošča pogoju

Imamo predikat `p`, ki za vsako število vrne `true` ali `false`. Med števili 1, 2, ..., `n-1` želimo poiskati največje, ki zadošča `p`. Če takega števila ni, vrnemo 0. To lahko izrazimo z rekurzivno funkcijo `find`:

`find p 0` je enako 0, ker ni števila med 1 in -1

`find p (n+1)` je enak `n`, če velja `p n` enako `true`

`find p (n+1)` je enak `find p n k`, če ne velja `p n`

Definiramo funkcijo `find`, ki sprejme predikat `p` in število `n`.

Funkcija vrne največje izmed števil 1, 2, ..., `n - 1`, ki zadošča pogoju `p`. Če takega števila ni, vrne 0.

Funkcijo preizkusimo na naslednjih primerih:

- med števili od 1 do 5 poišči največje število, ki je manjše od 3:

```
lambda> show (find (λ k . < k 3) 5)
S (S Z)
```

- med števili od 1 do 5 poišči (največje) število, katerega kvadrat je enak devet:

```
lambda> show (find (λ k . == (* k k) (+ 4 5)) 5) S (S (S Z))
```

```
find := λ predikat n . n (λ m r . if (predikat m) m r) 0 ;
```

Deljenje

Definiramo funkcijo `/`, ki deli naravna števila. Če se deljenje ne izide, funkcija vrne 0.

```
/ := λ n m . find (λ k . == (* m k) n) (naslednik n) ;
```

Urejene trojice

V λ-računu definiramo funkcije za delo z urejenimi trojicami:

- `triple` zgradi novo trojico
- funkcije `fst`, `snd` in `trd` pa vrnejo prvi, drugi oziroma tretji element dane trojice

Primer:

```
# :constant U
# :constant V
# :constant W
# snd (triple U V W)
U
```

```
triple := λ a b c . λ p. p a b c
fst := λ p . p (λ x y z . x)
snd := λ p . p (λ x y z . y)
trd := λ p . p (λ x y z . z)
```

Dvakratnik Scott-Church števila

V Scott-Churchevi notaciji smo naravno število `n` predstavili z `n`-kratno aplikacijo funkcije na argumentu:

```
0 := λ x f . x
1 := λ x f . f 0 x
2 := λ x f . f 1 (f 0 x)
3 := λ x f . f 2 (f 1 (f 0 x))
```

V λ-računu definiramo funkcijo `double`, ki vrne dvakratnik danega Scott-Churchevega števila.

```
double := λ n . λ x f . n (n x f) f ;
```

Naloga

V λ-računu definiramo izraza

```
O := λxy.x
I := λxy.y
```

Predstavljamo si, da je O bit nič in I bit ena. Definirajte λ-izraz X, ki izračuna bitni XOR, se pravi, da zadošča enačbam

```
XOO = O
XIO = I
XOI = I
XII = O
not := λx.xIO
xor := λxy.xy(not y)
```

Naloga

V λ-računu denifirajte izraz A tako, da bo veljalo (λx.Ax)(λx.Ax) = y

(λx.Ax)(λx.Ax)= A(λx.Ax)= y

A = λx.y

Naloga

V λ-računu predstavimo števila s Churchovimi numerali, na primer,

0 := λf x . x,

1 := λf x.fx,

2 := λf x.f(fx),

3 := λf x . f(f(fx)).

Katero število je λg y . 3 2 g y?

OCAML

Teorija

S programom izrazimo/deklariramo strukturo podatka, ki ga želimo imeti, ne pa nujno kako se izračuna.

- $\rightarrow$: desno asociativna
- R x R $\rightarrow$ R

  funkcija dveh spremenljivk; funkcija ene spremenljivke, ki je urejeni par
- R $\rightarrow$ (R $\rightarrow$ R)

  funkcija, ki sprejme realno število in vrne funkcijo, ki sprejme realno število in vrne realno število

Deklaracija vrednosti (spremenljivke, funkcije)

```
e : T
```

Pri vezavi spremenljivk in deklaraciji funkcij uporabljamo `let` frazo:

- `let <ime spremenljivke> = <izraz>`
- `let <ime spremenljivke> : <tip spremenljivke> = <izraz>`
- `let <ime funkcije> <argument 1> <argument 2> ... = <izraz>`
- `let <ime funkcije> (<arg 1> : <tip arg 1>) (<arg 2> : <tip arg 2>) ... : <tip vrednosti, ki jo funkcija vrača> = <izraz>`

```
let odgovor = min 8 7 * 6;;
```

Pri funkcijskih aplikacijah oklepajev nimamo (enako kot v lambda računu).
... razen, če je argument izraz.
Opomba: aplikacije so levo asociativne in imajo najvišjo prioriteto.

```
let fakulteta_5 = fakulteta 5 ;;
let ff_3_plus_1 = fakulteta (fakulteta 3) + 1 ;; (* 6! + 1 *)
```

Ocaml načeloma ne loči med funkcijami in spremenljivkami.

Deklaracija poveže ime z vrednostjo.
Imamo dve vrsti deklaracij:

- **globalne**

  `let name = expr ;;`
- **lokalne**

  V OCamlu nimamo normalnih "procedur", ker ni iterativnega programiranja. Če želimo izvajati nek podoben koncept, moramo uporabiti lokalna okolja.

  `let name = expr1 in expr2 ;;`

  Izraz `name` je poznano samo med evaluacijo izraza `expr2`. Lokalna deklaracija poveže `name` z vrednostjo `expr1`.

```
      let x1 = 3 in x1 * x1
```

Lokalna deklaracija zakrije vse predhodne deklaracije z istim imenom.

```
      let obseg_zemlje =
            let pi = 3.14159265 in
            let polmer_zemlje = 6371.0 in
      2.0 *. pi *. polmer_zemlje
;;
```

Funkcijski tipi

Povezava z lambda računom: direktna preslikava v Ocaml, osnovna oblika lambda računa dostopna v Ocaml, lambda abstrakcija definira funkcijo, aplikacija funkcije na argumentu, parameter je poljuben tip.

Funkcijski izraz je sestavljen iz parametra (spremenljivka) in telesa (izraz) funkcije. Parameter realizira lambda abstrakcijo.

Funkcijski tip `a -> b` je tip funkcij, ki sprejmejo argument tipa `a` in vrnejo rezultat tipa `b`.

`a → b → c  ≡  a → (b → c)`

Vsaka funkcija lahko sprejme natanko en argument!
V primeru, ko argumenta ne rabimo, uporabimo argument tipa `unit`.
Obstaja samo ena vrednost tega tipa, tj. [()].

Funkcije višjega reda in aplikacije

Aplikacije levo asociativne in evaluira telo funkcije tako, da se formalni parameter zamenja z dejanskim.
Uporabimo lahko `fun` ali `function`.
Funkcijo, ki vzame kot argument drugo funkcijo ali vrne funkcijo kot rezultat imenujemo funkcija višjega reda.
Aplikacija je levo asociativna, **funkcijski tipi** pa desno.
`(int -> int) -> int -> int` ali `(int -> int) -> (int -> int)`

```
fun x -> (fun y -> 2 * x - y + 3);;
- : int -> int -> int = <fun>
```

Sprejme `int` in vrne `int->int`.
Tip izraza lahko beremo na običajen način kot tip funkcije, ki pričakuje dve števili in vrne celo število. V okviru funkcijskega jezika beremo takšne tipe kot funkcije, ki imajo argument tipa `int` in vrnejo funkcijo, katere argument je `int` in rezultat tipa `int`.

```
fun x -> (fun y -> 2 * x - y + 3) 3;;
- : int -> int = <fun>
```

```
(fun x -> (fun y -> 2 * x - y + 3)) 10 3 ;;
-: int = 20
```

Ko napišemo `f a b`, predpostavljamo implicitno oklepaje, ki so definirani v skladu z levo asociativnim pravilom za povezovanje aplikacije funkcije: `(f a) b`

Poimenovanje

```
let funkcija = fun x -> x + 1;;
val funkcija : int -> int = <fun>

let funkcija x = x + 1;;
val funkcija : int -> int = <fun>

fun x -> x ;;
- : 'a -> 'a = <fun> // poljubna tipa

let funkcija(x,y) = x + 10 * y * y;;
val funkcija : int * int -> int = <fun>

let funkcija x y = x + 10 * y * y;;
val funkcija : int -> int -> int = <fun>

let inc_int : int -> int = fun x -> x + 1;;
```

Funkcije in argumenti

Funkcije, ki "sprejmejo" dva argumenta so še vedno funkcije, ki sprejmejo en argument. Spodje deklaracije so ekvivalentne.

```
let vsota x y = x + y ;; (* tipa int -> int -> int *)
let vsota x = fun y -> x + y ;;
let vsota = fun x -> fun y -> x + y ;;
let vsota = fun x y -> x + y ;;

let vsota (x, y) = x + y ;;
let vsota (x, y) = fun(x, y) -> x + y ;;
```

Zaporedje izrazov (enojno podpičje)

`(<izraz 1> ; <izraz 2> ; ... ; <zadnji izraz>)`

Rezultat takega izraz je izračunan `<zadnji izraz>`. Zadnja vrstica bo "return".

Na podpičje lahko gledamo kot nek operator, ki sprejme 2 argumenta. Prvi argument vrača nekaj kar je tipa `unit`. Drugi argument je lahko kateregakoli tipa. Rezultat prvega argumenta ignorira, rezultat drugega pa vrne.

Preostali izrazi ponavadi imajo stranske učinke.

Primer uporabe (funkcija za razhroščevanje):

```
let izpisi_in_vrni x =
    print_endline (string_of_int x);
    x
;;
```

V OCamlu vsaka funkcija mora vrniti neko vrednost. Ko nimamo ničesar za vrnit (glavna funkcionalnost funkcije je stranski učinek), vrnemo vrednost tipa unit tj. [`()`].

Primer:

Funkcija `print_endline` je tipa `string -> unit`.

Njen stranski učinek pa je izpis na standardni izhod.

Operatorji

Operatorji so funkcije:

```
let vsota = (+) ;; (* int -> int -> int *)

(+) 3 4 ;; (* enako kot 3 + 4 *)

12 + 3 ;; (* celoštevilsko seštevanje *)
12.0 +. 3.0 ;; (* seštevanje v plavajoči vejici *)

12 / 3 ;; (* celoštevilsko deljenje *)
12.0 /. 3.0 ;; (* deljenje v plavajoči vejici *)
5 mod 2 ;; (* ostanek pri deljenju *)



- 3 ;; (* operator za negacijo [-] deluje za cela števila, *)
- 4.5 ;; (* kot tudi za števila v plavajoči vejici *)

~- 3 ;; (* celoštevilska negacija *)
~-. 3.4 ;; (* negacija v plavajoči vejici *)
```

Znaki in nizi

```
let niz = "Pozdravljeni, svet" ;;
```

(* zlaganje nizov z uporab operatorja [(^)] *)

```
let some_str = "Pozdravljeni" ^ ", svet"  ;;
```

```
'a'
```

Funkcije

```
int_of_string, string_of_int, string_of_float, float_of_string
```

```
string_of_int 1987
- : string = "1987"
```

Enotski tip

Tip `unit` opisuje množico, ki vsebuje en sam element, ki ga označimo `()`.

```
# () ;;
- : unit = ()
```

```
();;
let pozdrav () = print_endline "Pozdravljen, svet!" ;;
pozdrav () ;;
```

Deklaracije tipov

Uporabljamo dva glavna konstruktorja tipov: **produkt za n-terice** in **vsota za unije**.

```
type name = typedef
```

Zmnožek tipov (kartežični produkt) (urejeni pari, terke)

Vrednosti različnih tipov lahko združimo v pare oz. n-ternice. Vrednosti, ki sestavljajo n-terice so ločene z vejicami. Konstruktor tipa, ki definira n-terice je *.

```
let par_stevil = 3, 4 ;; (* tipa int * int *)
let trojica = (1, "dva", [3.0; 4.0]) ;; (* tipa int * string * float list *)
[(1, 2)] = [1, 2] ;; (* tipa bool *)

(3, "banana");;
fst (3, "banana");;
     - : 'a * 'b -> 'a = <fun>
snd (3, "banana");;
let (a, b, c) = (1, "banana", false) ;;
let (_, _, z) = (1, "banana", false) ;;

let thd (_, _, z) = z ;;
thd(3, 2, 1);;
```

Zapisi

Zapisi so n-terice, ki imajo imenovane komponente. Zapis vedno ustreza neki deklaraciji novega tipa. Tip zapisa je definiran z deklaracijo imena in tipa vsake posamezne komponente.

Prednosti zapisov v primerjavi z n-tericami: opisni način dostopa do komponent zaradi imen polj, imena polj omogočajo enostaven način definicije ujemanja vzorcev, dostop je omogočen na uniformen način preko imena in vrstni red komponent ni več pomemben.

```
type oseba = { ime : string; priimek : string};;
{ime="Eva"; priimek="Bizilj"};;
oseba.ime;;
```

Vsota tipov (razločevanje primerov, osnovno ujemanje vzorcev)

Za razliko od n-teric in zapisov, ki ustrezajo kartezijskemu produktu, ustreza deklaracija vsot unijam množic. Različni tipi so zbrani v en sam tip.
Različne člane unije identificiramo s pomočjo konstruktorjev.

```
type name =
      | Name name1
      | Name2 of tj ...
```

Ime konstruktorja je izbran identifikator. Imena konstruktorjev se vedno začnejo z veliko črko.

Konstrukcija konstant

Konstruktor, ki ne pričakuje argumenta, lahko uporabljamo v izrazih kot konstanto.

```
type coin = Heads | Tails
```

Konstruktorji z argumenti

Ključna beseda `of` napove tipe konstruktorjevega argumenta. Naštevanje konstruktorjev omogoča združevanje različnih tipov v en sam toš.

```
type suit = Spades | Hearts
type card =
      | King of suit
      | Knight of suit
```

Kreacija vrednosti tipa `card` se izvrši preko aplikacije konstruktorja na vrednosti primernega tipa.

```
King Spades;;
```

Unije in ujemanje vzorcev

Za delo z vrednostmi vsote uporabljamo ujemanje vzorcev.

```
let string_of_suit = fun
      | Spades -> "spades"
```

Vzorec je struktura zgrajena z eno izmed danih podatkovnih struktur programskega jezika, konstant primitivnih tipov, spremenljivk ...
Ujemanje vzorcev se izvaja na vrednostih. Uporablja se za ujemanje strukture vrednosti in vodenje izvajanja programa v skladu z ujemanjem.

```
match expr with
   | p1 -> expr1
   | ...
```

Vrednosti izraza `expr` se prepoznava sekvenčno na vzorcih `p1`, ... Če se vzorec ujema z vrednostjo, se izvede pripadajoča koda. Vzorci v sekvenci so istega tipa. Vertikalna črta pred prvim vzorcem je opcijska.

Vzorci morajo biti nujno linearni - nobena spremenljivka se ne pojavi več kot enkrat v enem vzorcu.

```
(* ujemanje vrednosti *)
let je_nic x =
   match x with
   | 0 -> true
   | _ -> false  (* vzorec [_] predstavlja "wildcard" *)
;;

(* za funckije (ki so v zgornji obliki) obstaja sintaktičen sladkorček *)
let je_nic = function
   | 0 -> true
   | _ -> false
;;

(* uporaba "stražarjev pri ujemanju vzorcev" *)
let abs x =
   match x with
   | x when x < 0 -> -x
   | _ -> x
;;

(* konstruktorji in ujemanje vzorcev *)

type animal = Dog of string | Cat of string ;;
```

```
let say x =
      match x with
      | Dog d -> d ^ " says woof"
      | Cat c -> c ^ " says meow"
;;
say (Cat "Fluffy");;

let mesana_vsota x y =
match x, y with
| Int a, Int b -> Int (a + b)
| Float a, Float b -> Float (a +. b)
| (Float a, Int b | Int b, Float a) ->
   Float (a +. float_of_int b)
;;

type barva = { blue : float; green : float;  red : float }
type izdelek =
      | Cevelj of barva * int
      | Palica of int
      | Posoda of int
Palica 7

let cena1 z =

match z with
      | Cevelj (b, v) -> if v < 25 then 15 else 20
      | Palica x -> 1 + 2 * x
      | Posoda y -> 7
      | Maska -> 2000
;;

let cena2 =
      fun z -> match z with
      | Cevelj (b, v) -> if v < 25 then 15 else 20
      | Palica x -> 1 + 2 * x
;;

let cena3 = function
      | Cevelj (b, v) -> if v < 25 then 15 else 20
      | Palica x -> 1 + 2 * x
```

Rekurzivne podatkovne strukture

Rekurzivni tip je tip, katerega definicija vsebuje eno ali več referenc na definiran tip.

Induktivni tipi

Induktivni tipi ... elementi rekurzivnega tipa končni (sestavljamo s pomočjo konstruktorjev)
Koinduktivni tipi ... imamo lahko neskončne elemente

Seznam

So polimorfni - vsi elementi seznama so istega tipa. Lahko je prazen ali sestavljen iz elementov istega tipa.
Seznam vsebuje glavo in rep, kjer je rep definiran rekurzivno kot seznam.

implementacija v ozadju `type 'a list = (::) of 'a * 'a list`

```
let seznam = [1; 2; 3] ;;
let seznam' = 1 :: 2 :: 3 :: [] ;;
```

Sezname lahko gradimo tudi preko konstruktorja `::` , ki je desno asociativen.

```
(* "dodajanje" na začetek seznama z uporabo konstruktorja [(::)] *)
1 :: [2; 3] ;; (* [1; 2; 3] *)

ℓ = 1 :: 2 :: ℓ
let rec l = 1::2::l
```

Seznam `[1; 2; 3]` je okrajšava za `Cons (1, Cons (2, Cons (3, Nil))).`

```
[1; 2; 3; 4];;
1::2::[3; 4];;
```

Konketenacijo dveh seznamov dosežemo z `@`.

```
[1; 2; 3]@[4; 5; 6];;

type seznam =
      | Nil // prazen
      | Cons of int * seznam

(* drugi element seznama *)
List.nth seznam 1 ;;

(* prvi element seznama *)
List.hd seznam ;;
```

(* "rep" seznama *)
```
List.tl seznam ;;
```

(* Funkcije višjega-reda za delo z seznami: map, filter, fold_left, ... *)
```
List.map (fun x -> x * 2) [1; 2; 3] ;;
```

(* na vsakem elementu pokličemo funkcijo string_of_int *)
```
let nizi = List.map string_of_int stevila
```

(* kvadriramo vsak element seznama s pomočjo anonimne (λ) funkcije *)
```
let nizi = List.map (fun x -> x*x) stevila ;;
```

Lastni podatkovni tipi

Konstruktorji tipa se morajo začeti z veliko začetnico.

```
type searchtree = Empty | Node of int * searchtree * searchtree
```

(* sinonimi za podatkovni tip *)
```
type celoStevilo = int ;;
type parStevil = int * int ;;
```

(* vsota podatkovnih tipov *)
```
type programskiJezik = OCaml | C | Haskell | Java ;;
let jezik = OCaml ;;  (* tipa programskiJezik *)
```

(* konstruktorji lahko imajo argumente *)
```
type stevilo = PlusNeskocno | MinusNeskocno | Realno of float ;;
let r0 = Realno (-3.4) ;; (* tipa stevilo *)
```

(* tip, ki predstavlja mešanico celih in stevil v plavajoči vejici *)
```
type number = Int of int | Float of float ;;
```

(* tip za točko v ravnini *)
```
type tocka = Tocka of float * float;;
let t0 = Tocka (2.0, 3.0) ;;
```

Koinduktivni tipi (tokovi)

V OCamlu lahko simuliramo tokove z uporabo tehnike **zavlačevanja** ("thunk").

Imamo izraz `e` tipa `t`, ki ga še ne želimo izračunati => lahko ga predelamo v funkcijo `fun () -> e.`

Ker je `e` znotraj telesa funkcije, se bo izračunal šele, ko funkcijo uporabimo na `()`.

```
e: int
fun() -> e: unit -> int
```

Prvih `k` elementov iz toka `s` pretvori v seznam

```
let rec to_list k s =
      match (k, s) with
            | (0, _) -> []
            | (n, Cons (x, s)) -> x :: to_list (n-1) (s ())
```

Mešano

```
let node (v, l, r) = Node(v, 1 + max (height l) (height r), l, r)
int * avltree * avltree -> avltree

let node_currying1 = fun n l r -> Node(n, 1 + max (height l) (height
r) ,l, r);;
let node_currying2 = fun n -> fun l -> fun r -> Node(n, 1 + max (height
l) (height r) ,l, r)
value -> avltree -> avltree -> avltree = <fun>
```

Rekurzija in rekurzivni tipi

```
let rec fact n = (if n = 0 then 1 else n * fact (n - 1)) ;;
val fact : int -> int = <fun>
```

Razstavimo na:

- telo rekurzije (samo po sebi ni rekurzivno)
- rekurzivni sklic funkcije same nase

```
let telo funkcija n =
      if n = 0 then 1
      else n * funkcija (n - 1)

let rec funkcija n = telo funkcija n

let rec fakulteta = function
      | 0 -> 1
      | n -> n * fakulteta (n - 1);;

let rec je_sodo = function
      | 0 -> true
      | n -> je_liho(n - 1)
and je_liho = function
      | 0 -> false
      | n -> je_sodo(n - 1);
```

Reference

Gradnika programskega jezika Ocaml, ki ustrezata `&` in `*` označimo z `ref` in `!`.

Operator `ref` skuži kot konstruktor tipa kot tudi kot operator, ki vrne naslov vrednosti ali spremenljivka.

Operator `!` pa ovrednosti kazalec oz. prikaže vrednost, ki se nahaja na naslovu predstavljenim s kazalcem.

Ko v OCamlu definiramo vrednost `x` z `let x = e1 in e2` je `x` **nespremenljiva vrednost**.

Če želimo spremenljivo vrednost, moramo narediti **referenco**:

- `ref v` ... nova referenco z vrednostjo `v`
- `!r` ... trenutna vrednost reference `r`
- `r := v` .. nastavimo vrednost reference `r`

Razlika med referencami in spremenljivkami

Če želimo imeti zapis, v katerem lahko spreminjamo posamezen element, potem uporabimo referenco.

```
let r = ref 5;; (* val r : int ref = {contents = 5} *)
```

Kazalec definiramo na vrednost 3.

```
!r + 5;; (* - : int = 10 *)
```

Izpišemo vrednost kazalca.

```
r + 5 (* Error: This expression has type int ref but an expression was expected of type int *)
r := 8;; (* - : unit = () *)
```

Uporaba operatorja `!` za ovrednotenje kazalca ter infiksna funkcija `:=` s katero spreminjamo vsebino na katero kaže kazalec.

Ocaml ima definiran polimorfični referenčni tip `ref`, ki ga lahko vidimo kot tip kazalcev na poljubno vrednost. Definira je z eno samo spremenljivko komponento.

```
type 'a ref = {mutable contents: 'a}
```

Pogojni stavek

```
if 3 = 4 then 0 else 4
```

Zanke

Realizacija anke ima smisel, če imamo možnost spreminanja vsebine spremenljivk.

V Ocamlu imamo dva iteracijska gradnika: for stavek in while stavek.

Oba stavka vračata vrednost `()` tipa unit.

```
while <pogoj> do
      <izraz> // ne moremo imeti dvojnih dvopičji
done
```

```
for i = <spodnja meja> to <zgornja meja> do
      <izraz>
done
```

Rezultat se skozi iteracije zavrže, vrne se samo zadnji.

Akumulatorji in repna rekurzija

Pravimo, da je rekurzivni klic funkcije **repni klic** ali **klic na repu** (*tail recursive*), če je rezultat rekurzivnega klica hkrati tudi rezultat funkcije. Povedano z drugimi besedami, funkcija se pokliče rekurzivno in nato takoj vrne rezultat rekurzivnega klica.

Na primer, v rekurzivni funkciji

```
let rec f = function
| 0 -> 1
| n ->
    if n mod 2 = 0
    then f (n / 2)
    else 3 * f (n - 1)
```

je prvi rekurzivni klic repni, ker se izvede `f (n / 2)` in nato nič drugega, drugi rekurzivni klic pa ni repni, ker je treba rezultat rekurzivnega klica `f(n - 1)` še množiti s `3`.

Pogosto lahko rekurzivno funkcijo, ki nima repnih klicev, predelamo v tako, ki ima repne klice. Pri tem uporabimo t.i. **tehniko akumulatorjev**, ki imajo v rekurzivni funkciji podobno vlogo kot pomožne spremenljivke v zanki.

```
let vsota_lihih1 n =
    let v = ref 0 in
    let i = ref 0 in
    while !i < n do
        v := !v + (2 * !i + 1) ;
        i := !i + 1
    done ;
    !v
```

Naredimo pomožno rekurzivno funkcijo, ki sprejme več argumentov in enega izmed teh argumentov uporabi kot akumulator.
Akumulator predstavlja trenutno vsoto in ga uvozimo iz rekurzije v rekuzijo. Ko pridemo do končnega pogoja, vrnemo akumulator.

```
let vsota_lihih2 n =
    let rec vsota v i =
        if i < n
        then vsota (v + (2 * i + 1)) (i + 1)
        else v
        in
    vsota 0 0
```

Splošna pretvorba zanke while v rekurzivno funkcijo

Zanko `while` prevedemo v rekurzivno funkcijo z akumulatorjem in repnim klicom. Obravnavajmo zanko `while` oblike

```
s := s0
while p(s) do
  s := f(s)
done ;
return r(s)
```

Tu smo z `s` označili skupno stanje vseh spremenljivk, ki nastopajo v zanki, `s0` je začetno stanje, `p(s)` je pogoj (ki je odvisen od stanja `s`), zanka vsakič trenutno stanje `s` nastavi na novo stanje `f(s)`, na koncu pa vrne rezultat `r(s)` (ki je spet odvisen od `s`).

Na primer, zanko, ki izračuna n-to Fibonaccijevo število bi zapisali takole:

```
(a,b,i) := (0,1,0)
while i < n do
   (a,b,i) = (b,a+b,i+1)
done ;
return a
```

Zgornjo zanko lahko predelamo v rekurzivno funkcijo `zanka`, ki spreme `s0, p, f` in `r` ter izračuna to, kar bi sicer izračunala zanka `while`:

```
let zanka s0 p f r =
    let rec loop s =
        if p s then loop (f s) else r s
    in loop s0
```

Tip funkcije `zanka` je

```
α → (α → bool) → (α → α) → (α → β) → β
```

kar pomeni, da ima začetno stanje `s0` (poljuben) tip α, pogoj `p` je funkcija, ki slika stanje v `bool`, `f` je funkcija, ki stanje predela v novo stanje, in `r` je funkcija, ki stanje predela v rezultat (poljubnega) tipa β.

Specifikacija, implementacija, abstrakcija

Specifikacija S: zahteva, ki opisuje, kakšen izdelek želimo

- **specifikacija**: zahteva/opis, kaj naj koda počne
- **vmesnik/interface**: protokol za uporabo programske kode (opis, kaj ta koda zna)

Implementacija I je izdelek, ki zadošča specifikaciji S, če ustreza zahtevam iz S.

Ocaml

Specifikacija

signatura/podpis

```
module type GROUP =
sig
  (* specifikacija: kaj je grupa *)
  type g
  val mul : g * g -> g (* val ... spremenljivke znotraj modula *)
  val inv : g -> g
  val e : g
end
```

Implementacija

modul/struktura

```
module GROUP
struct
      let
      ...
end
```

// vmes ni podpičij

Abstrakcija

Skrijemo podrobnosti implementacije (skrijemo vsebino modula `M`, razen tistih komponent, ki so naštete v `S`)

`M : S`, modul : signatura (M mora zadoščati S - lahko vsebuje več kot je v specifikaciji)

```
module M : S =
      struct
      end
```

Podpišemo modul `Z3` z `GROUP`

```
module Z3 : GROUP =
struct
  type g = Zero | One | Two
  let e = Zero
  ...
end
```

```
module IntListSet : SET with type element = int =
struct
...
end
```

Generično programiranje

funktor (funkcija, ki sprejme modul in iz njega naredi modul)

Funktor `F`, ki sprejme strukturo `A`, ki zadošča signaturi `S` in vrne strukturo `B`

```
module F(A : S) =
struct
      < definicija strukture B, tu smemo uporabljati A >
end

module type ORDERED =
sig
      type t
      val cmp : t -> t -> order
end
```

Modul `M` opisuje element, ki ga bomo uporabljali v modulu `ListSet`.

Ta element bo tipa `ORDERED`.

Množico bomo razširili tako, da bodo njeni elementi tipa `M`.

```
module ListSet (M : ORDERED) : SET with type element = M.t
struct
          type element = M.t
end

module S = ListSet(struct type t = string let cmp = ocaml_cmp end) ;;
```

Podtipi in objekti

Če ima `e` tip `A` in je `A` podtip `B`, potem ima `e` tudi tip `B`.

`A ≤ B`

A je podtip B « ali » B je nadtip A.

$$\frac{e : A \qquad A \leq B}{e : B} \qquad \frac{}{A \leq A} \qquad \frac{A \leq B \quad B \leq C}{A \leq C}$$

`int ≤ float`

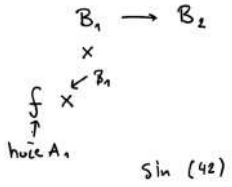
Če na nekem mestu pričakuje float, potem lahko damo tja int.

Strukturna pravila

$$\frac{A_1 \leq B_1 \qquad A_2 \leq B_2}{A_1 \times A_2 \leq B_1 \times B_2}$$

Kaj mora veljati, da se bo urejeni par A1 x A2 lahko obnašal kot B1 x B2?

$$\frac{B_1 \leq A_1 \qquad A_2 \leq B_2}{A_1 \to A_2 \leq B_1 \to B_2}$$



B1 se mora uporabljati kot, da bi bil A1.

A => B je **kontravarianten** (obrača smer) v A in **kovarianten** (ohranja) v B.

Podipi zapisov

(nabori polj s poimenovanimi komponentami)

Na primer, točke v ravnini lahko predstavimo z vrednostmi **tipa zapisov**

```
{ x : float; y : float }
```

Primer vrednosti tega tipa je **zapis**

```
{x = 3.14; y = 2.78}
```

Podtip zapisov po širini

```
A := { x : float; y : float }
```

in

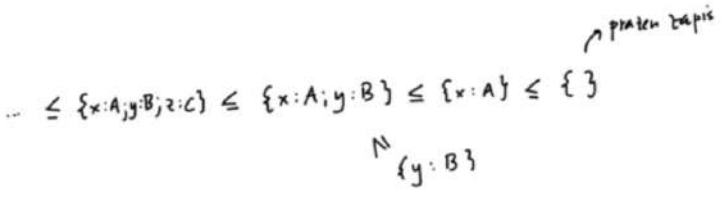
```
B := { x : float; y : float ; z : float }
```

`A` ≤ `B`: ne velja ... Če bi ga lahko uporabljali, kot da ima tip `B`, potem bi smeli pisati `a.z`, kar seveda ne gre.

`B` ≤ `A`: velja ... Lahko ga uporabimo, kot da bi imel tip `A`, saj smemo pisati `b.x` in `b.y`.

Podtip je "širši" (ima več polj) kot njegov podtip.

```
{ z : float; x : float; y : float } ≤ { x : float; y : float }
```



Ista oznaka in isti tip!

Podtip zapisov v globino

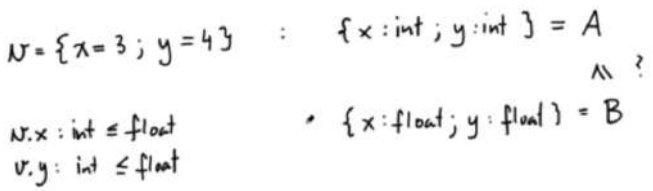
Ne gledamo, če imamo več funkcionalnosti ampak, če je funkcionalnost vsaj tako dobra kot je zahtevana.

Predpostavimo, da velja `int ≤ float`.

Zapis `v = { x = 3; y = 5 }` ima tip `{x : int; y : int}` = `A`.

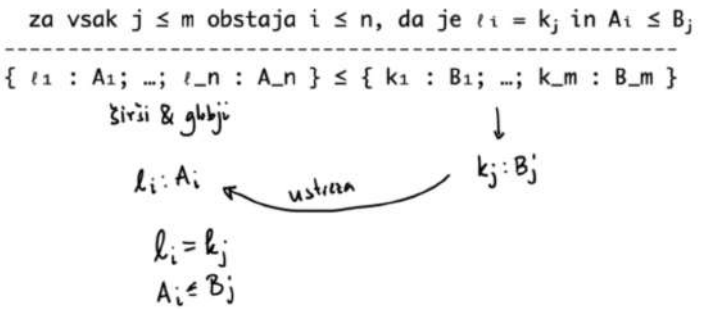
Ali ga lahko uporabljamo, kot da bi imel tudi tip `{x : float; y : float} = B`?
Da, saj lahko njegovi komponenti `v.x` in `v.y` uporabljamo, kot da bi imeli tip `float`, zahvaljujoč `int ≤ float`.

Velja `A ≤ B`?



Kombinirani podtipi (v širino in globino)

Vrstni red ni pomemben!



<u>Zapis s spremenljivimi polji</u>

(**podtipi v širino**, podtipi v globini ne veljajo več)

```
type point = { mutable x : float; y : float }};
p.x <- 100.5 ;;
p
```

Naloge

Dana je funkcija `float -> int`. Ali jo lahko uporabljamo kot `int -> int`?

Da; dobi `int` in pretvori v `float`.

Dana je funkcija `int -> int`. Ali jo lahko uporabljamo kot `int -> float`?

Da; zna pretvoriti v `float`.

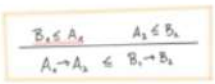
Dana je funkcija `int -> float`. Ali jo lahko uporabljamo kot `int -> int`?

Ne. `Float` ne znamo pretvoriti v `int`.

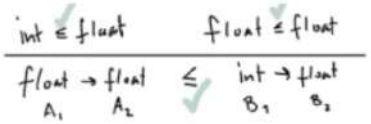
Dana je funkcija `float -> int`. Ali jo lahko uporabljamo kot `float -> float`?

Podamo ji `float`, ki ga lahko spremeni v `int`. Rezultat bo tipa `float`, ki ga ne bo znala pretvoriti v `int`.

Zakaj `float -> float ≤ int -> float`?



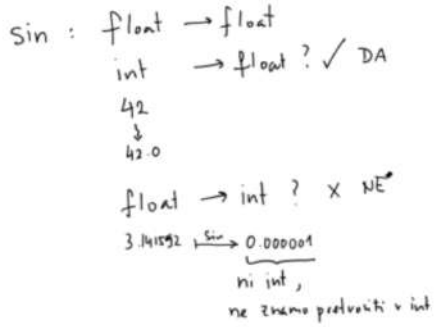
Uporabimo pravilo



`float ≤ float`? refleksivnost

`int ≤ float`? predpostavili zgoraj

Zgornja pogoja sta izpolnjena, zato drži.



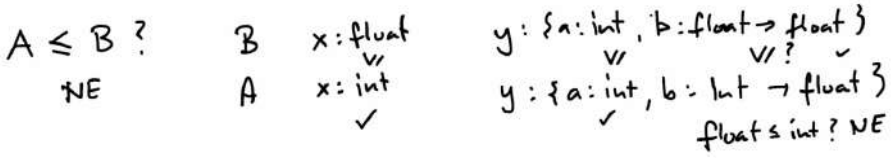
Velja `int ≤ float`.

```
A = {x : int, y : {a : int, b : int -> float}, z : bool}
B = {x : float, y : {a : int, b : float -> float}}
C = {x : int, y : {a : int, b : int -> float}}
```

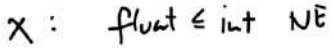
a) $A \leq B$?

Za vsakega v B-ju moramo najti enega v A-ju in pogledati ustreznost tipov.

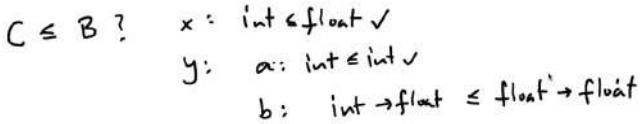
Ko rečemo a.x dobimo int, kar bomo lahko pretvorili v float.



b) $B \leq C$?



c) $C \leq B$?



Ne.

Objektno programiranje

objekt = rekurzivni zapis

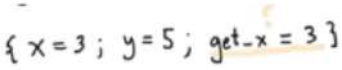
- **atributi** (enkapsuliran stanje): stanje lahko spreminjamo

  polja objekta se lahko sklicujejo na celoten objekt (`this`/`self`)
- **metode** (funkcionalnost)

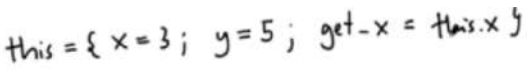
  atributi, metode ... polja se da spreminjati

Atributi in metode so v bistvu samo polja nekega rekurzivnega zapisa.

Zapis



Objekt



Objekt lahko naredimo neposredno (ne potrebujemo razreda).

```
object (this)
      val x = ⋯
      val y = ⋯

      method f = ⋯
      method g = ⋯
end
```

```
p ;;
-: <get_x : int; get_y : int; move : int -> int -> unit > = <obj>
```

Dobili smo samo metode. Zakaj? Stanje je ekapsulirano, atributi niso vidni.

Strukturni podtipi za objekte, ne za zapise.

Dostop do metode

`objekt#metoda` (* dostop do metode v objektu*)

```
let f o = o#get_x +. o#get_y
```

Opisovanje protokolov in standardov s podatkovnimi tipi

V jeziku OCaml definiramo podatkovni tip za opis odgovora strežnika HTTP, ki je določen v RFC 7231.

Iz ukazne vrstice poženemo interpreter z `ocaml`. Pri tem lahko podamo datoteko, ki se bo naložila ob zagonu, npr.

```
ocaml -init http.ml
```

Datoteko naložimo tudi kasneje z direktivo (prvi `#` je ukazni poziv, drugega pa napišemo kot del ukaza `#use`):

```
# #use "http.ml";;
```

Za izhod iz interpreterja pritisnemo `ctrl+D`. Za lažje interaktivno delo lahko uporabimo `rlwrap`, ki poljubnemu interaktivnemu programu doda možnost urejanja ukazne vrstice:

```
rlwrap ocaml
```

Odgovor HTTP

Najprej si oglejmo, kako izgleda glava odgovora strežnika HTTP. V ukazni vrstici poženemo:

```
$ curl -I https://fri.uni-lj.si/sl

HTTP/1.1 200 OK
Date: Sun, 21 Mar 2021 10:18:35 GMT
Server: Apache/2.4.29 (Ubuntu)
X-Content-Type-Options: nosniff
Expires: Sun, 19 Nov 1978 05:00:00 GMT
Cache-Control: no-cache, must-revalidate
X-Content-Type-Options: nosniff
Content-Language: sl
X-Frame-Options: SAMEORIGIN
Access-Control-Allow-Origin: https://hugo.fri.uni-lj.si
Content-Type: text/html; charset=utf-8
```

Vidimo, da je glava sestavljena iz dveh delov: prva vrstica poda različico protokola in status odgovora, preostale vrstice pa določajo vrednosti posameznih polj.

Podatkovni tip `status`

Prva vrstica v odgovoru HTTP nam pove različico protokola in status poizvedbe, npr.

```
HTTP/1.1 200 OK
```

Mi jo bomo predstavili s podatkovnim tipom `status`, ki je zapis z dvema poljema:

In [ ]:
```
type status = {version : string; code : int};;

(* type status = { version : string; code : int; } *)
```

Novo vrednost tega tipa naredimo z:

In [ ]:
```
let myStatus = {version = "HTTP/1.1"; code = 200};;

(* val myStatus : status = {version = "HTTP/1.1"; code = 200} *)
```

Določeno polje iz zapisa dobimo tako:

In [ ]:
```
myStatus.version;;

(* - : string = "HTTP/1.1" *)
```

Definirajmo funkcijo `string_of_status` , ki za dani status vrne ustrezen niz. V OCamlu združujemo nize z operatorjem `^` :

In [ ]:
```
let string_of_status s =
  s.version ^ " " ^
  string_of_int s.code ^ " " ^
  (
    match s.code with
        | 200 -> "OK"
        | _ -> ""
  )

(* val string_of_status : status -> string = <fun> *)
```

Primer uporabe:

In [ ]:
```
string_of_status myStatus ;;

(* - : string = "HTTP/1.1 200 OK" *)
```

Podatkovni tip `response`

Odgovor HTTP je sestavljen iz statusa, glave, ki vsebuje poljubno število polj, in telesa. Telo je lahko niz HTML ali kaj drugega; tu ga bomo obravnavali kot niz. Tip za opis polj (zaenkrat le `Server` in `ContentLength`) bomo predstavili z vsoto

In [ ]:
```
type field =
    | Server of string
    | ContentLength of int

(* type field = Server of string | ContentLength of int *)
```

Tip `response` je zapis z glavo, seznamom polj in telesom:

In [ ]:
```
type response = { status: status; headers: field list; body: string }
```

Zdaj lahko ustvarimo preprosto sporočilo:

In [ ]:
```
let r = {
  status  = {version="HTTP/1.1"; code=200};
  headers =  [
    Server "nginx/1.6.2";
    ContentLength 13;
    ContentType "application/json";
    TransferEncoding [Gzip; Identity];
    Date {dayOfTheWeek="Wed"; dayOfMonth=3; month="Mar"; year=2020;
     hour=14; minute=10; second=30; timezone="GMT"}
  ];
  body="hello world!\n";
  sadje="banane" (* v nobenem podatkovnem tipu nimamo sadja zato vrne erro
  } ;;

(*
val r : response =
  {status = {version = "HTTP/1.1"; code = 200};
   headers = [Server "Apache/2.4.29 (Ubuntu)"; ContentLength 13];
   body = "hello world!\n"}
*)
```

Naloga: izpis polja

Definirajte funkcijo `string_of_field : field -> string` , ki za dano polje vrne ustrezen niz. Primer uporabe:

```
string_of_field (Server "Apache/2.4.29 (Ubuntu)") ;;
- : string = "Server: Apache/2.4.29 (Ubuntu)"

string_of_field (ContentLength 12) ;;
- : string = "Content-Length: 12"
```

In [14]:
```
let string_of_field: field -> string = fun x ->
    match x with
        | Server name -> "Server :" ^ name
        | ContentLength cl -> "Content-Length " ^ string_of_int cl

(* val string_of_field : field -> string = <fun> *)
```

In [ ]:
```
let string_of_field: field -> string = function
    | Server name -> "Server :" ^ name
    | ContentLength cl -> "Content-Length " ^ string_of_int cl

(* val string_of_field : field -> string = <fun> *)
```

In [ ]:
```
let string_of_field x =
    match x with
        | Server name -> "Server :" ^ name
        | ContentLength cl -> "Content-Length " ^ string_of_int cl

(* val string_of_field : field -> string = <fun> *)
```

Aplikacija je močnejša od operacije združevanja.

Preiskusimo delovanje funkcije.

In [ ]:
```
string_of_field (Server "Apache/2.4.29 (Ubuntu)") ;;

(* - : string = "Server :Apache/2.4.29 (Ubuntu)" *)
```

Naloga: izpis odgovora

Definirajte funkcijo `string_of_response : response -> string` , ki za dan odgovor oblikuje in vrne ustrezen niz.

```
# print_string (string_of_response r) ; flush stdout ;; (*
`flush stdout` je zaradi notebook-a` *)
HTTP/1.1 200 OK
Server: Apache/2.4.29 (Ubuntu)
Content-Length: 13

hello world!
```

Pomagate si lahko s funkcijama `String.concat` in `List.map` .

`String.concat` : sprejme seperator in seznam nizov ter jih združi v en niz

`List.map` : preslika posamezen niz

`r.headers` želimo preslikati v niz, pri čemer bodo posamezne preslikave polj imele na koncu novo vrstico.

In [15]:
```
List.map (fun x -> x) r.headers

(* - : field list = [Server "Apache/2.4.29 (Ubuntu)"; ContentLength 13] *)
(* vrne začetni seznam *)

(* seznam nizov *)
List.map string_of_field r.headers

(*- : string list = ["Server :Apache/2.4.29 (Ubuntu)"; "Content-Length 13"

List.map (fun f -> string_of_field f ^ "\n" ) r.headers

(* - : string list = ["Server :Apache/2.4.29 (Ubuntu)\n"; "Content-Length

String.concat "" (List.map (fun f -> string_of_field f ^ "\n") r.headers)

(* - : string = "Server :Apache/2.4.29 (Ubuntu)\nContent-Length 13\n" *)
```

In [ ]:
```
let string_of_response: response -> string = fun r ->
  string_of_status r.status ^ "\n" ^
  String.concat "" (List.map (fun f -> string_of_field f ^ "\n") r.headers
  r.body

(* val string_of_response : response -> string = <fun> *)
```

In [ ]:
```
let string_of_response: response -> string = fun r ->
    string_of_status r.status  ^ "\n" ^
    String.concat "\n" (List.map string_of_field r.headers) ^ "\n\n" ^
    r.body
```

Naloga: dodatna polja

Tip `field` razširite z naslednjimi možnostmi, pri čemer za vrednost polja zaenkrat uporabite kar niz:

- `Content-Type`
- `Transfer-Encoding`
- `Date`
- `Expires`
- `Last-Modified`
- `Location`

Odpravite opozorila, ki jih pri tem začne vračati interpreter.

In [16]:
```
type field =
    | Server of string
    | Content-Length of int
    | Content-Type of string
    | Transfer-Encoding of transferEncoding
    | Date of date
    | Expires of expires
    | Last-Modified of date

type date = {
    dayOfWeek : string;
    dayOfMonth : int;
    month : string;
    year : int;
    hour : int;
    minute : int;
    second : int;
    timeZone : string;
}

type expires =
  | Date of date (* POZOR: ta "Date" ne sme biti poimenovan isto kot v tipu
  | Number of int

type transferEncoding = Chunked | Compress | Deflate | Gzip | Identity
```

Naloga: `Transfer-Encoding`

Popravite tip za polje `Transfer-Encoding` tako, da vrednost namesto niza predstavite z izbiro med kodiranji `chunked`, `compress`, `deflate`, `gzip`, `identity`. Implementiraj funkcijo `string_of_transferEncoding`.

In [17]:
```
type transferEncoding =
  | Chunked
  | Compress
  | Deflate
  | Gzip
  | Identity

let string_of_transferEncoding s =
    match s with
    | Chunked -> "chunked"
    | Compress -> "compress"
    | Deflate -> "deflate"
    | Gzip -> "gzip"
    | Identity -> "identity"

(* val string_of_transferEncoding : transferEncoding -> string = <fun> *)

string_of_transferEncoding Chunked ;;
(* - : string = "chunked" *)
```

Naloga: datumi

Datumi so v glavi HTTP predstavljeni v taki obliki:

```
Sun, 21 Mar 2021 10:18:35 GMT
```

Definirajte podatkovni tip `date` in funkcijo `string_of_date` ter ju uporabite za polja `Date`, `Expires` in `Last-Modified`.

In [ ]:

```
type date = string * int * string;

type date = {
    dayOfWeek : string;
    dayOfMonth : int;
    month : string;
    year : int;
    hour : int;
    minute : int;
    second : int;
    timeZone : string;
}

let string_of_date d =
    d.dayOfWeek ^ "," ^
    string_of_int d.dayOfMonth ^ " " ^
    d.month ^ " " ^
    string_of_int d.year ^ " " ^
    string_of_int d.hour ^ ":" ^
    string_of_int d.minute ^ ":" ^
    string_of_int d.second ^ ": " ^
    d.timeZone
(* val string_of_date : date -> string = <fun> *)
```

Naloga: naslovi URI

Definirajte podatkovni tip `uri` za povezave in funkcijo `string_of_uri` ter popravite polje `Location` tako, da namesto niza uporablja ta tip. Sintaksa naslovov URI je predpisana v RFC 3986. Tip `uri` naj čim bolj natančno opiše te komponente: nekateri deli so opcijski, ime gostitelja je lahko domena ali naslov (IPv4 ali IPv6) itd.

In [ ]:

```
type uri = {
    scheme : string;
    user : string option;
    password : string option;
    host : host option;
    port : int option;
    path : string list;
    query : (string * string) list;
    fragment : string option;
```

```
}

type host =
    | IPV4 of int * int * int * int
    | IPV6 of string * string * string * string * string * string * string
    | HostName of string list

let string_of_host h =
    match h with
        | IPV4 (a, b, c, d) ->
          string_of_int a ^ "." ^ string_of_int b ^ "." ^ string_of_int c
        | IPV6 (a, b, c, d, e, f, g, h) ->
          "[" ^ a ^ ":" ^ b ^ ":" ^ c ^ ":" ^ d ^ ":" ^ e ^ ":" ^ f ^ ":"
        | HostName hn -> String.concat "." hn

let string_of_uri u =
    u.scheme ^ ":" ^
    (
        match u.host with
            | None -> ""
            | Some host -> "//" ^ (
                match u.user with
                    | None -> ""
                    | Some user -> user ^
                    ( match u.password with
                        | None -> ""
                        | Some pass -> ":" ^ pass
                    ) ^ "@"
            ) ^ string_of_host host ^
            (
                match u.port with
                    | None -> ""
                    | Some port -> string_of_int port
            )
    ) ^
    (
        match u.path with
            | [] -> ""
            | path -> "/" ^ string_of_path path
    ) ^
    (
        match u.query with
            | [] -> ""
            | query -> "?" ^ string_of_query query ) ^
            (
                match u.fragment with
                    | None -> ""
                    | Some fragment -> "#" ^ fragment
            )
    )
```

Napiši `response` .

In [ ]:

```
let my_response = {
    status = {
        version = "HTTP/1.1";
        code = 418;
    };
    headers = [
        Server "gws";
        ContentLength 1024;
        ContentType "text/html; charset=UTF-8";
        Date {
            dayOfWeek = "Fri";
            dayOfMonth = 30;
            month = "Mar";
            year = 2018;
            hour = 20;
            minute = 16;
            second = 32;
            timeZone = "CEST"
        };
        (* Expires (Number -1), *)
        Date {
            dayOfWeek = "Thu";
            dayOfMonth = 29;
            month = "Mar";
            year = 2019;
            hour = 16;
            minute = 28;
            second = 26;
            timeZone = "CEST";
        };
        Location {
            scheme = "http"; (* obvezno polje, zato spredaj nima "SOME" *)
            host = Some (IPV6 ("0000", "1234", "abcd", "01ab", "ffff", "99
            (* host = Some (IPV4 (127, 0, 0, 1)); *)
            (* host = Some (HostName ["google"; "si"]); *)
            user = Some "admin";
            password = Some "ultra_safe_password_1A!";
            port = None; (* to polje ni podano *)
            path = ["web"; "page"];
            query = [("a", "b"); ("x", "y")];
            fragment = Some "top"
        };
        TransferEncoding Gzip;
    ];
```

Induktivni podatkovni tipi — AVL drevesa

Implementirali bomo AVL drevesa v jeziku OCaml. AVL drevo je dvojiško iskalno drevo, v katerem se za vsako vozlišče višini levega in desnega poddrevesa razlikujeta za največ 1.

Podatkovni tip AVL drevo

V drevesih hranimo le cela števila (`int`).

AVL drevo je:

- prazno (`Empty`)
- vozlišče (`Node`):
  - vsebine (števila)
  - višine drevesa
  - levega ter desnega AVL poddrevesa

Na AVL drevo lahko gledamo kot na strukturo, ki predstavlja neko množico.

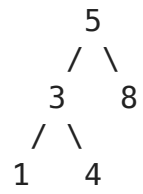
Naloga: definirajte podatkovni tip `avltree` , ki opisuje AVL drevesa.

In [ ]:
```
type avltree =
    | Empty
    | Node of int * int * avltree * avltree

type value = int
type height = int

type avltree =
    | Empty
    | Node of value * height * avltree * avltree
```

Spodnja funkcja `showTreeHorizontally : avltree -> unit` grafično prikaže vhodno drevo `t` po nivojih.

In [ ]:
```
let showTreeHorizontally t =
    let rec strign_of_avltree_level lvl = function
        | Empty -> if lvl = 0 then "nil" else "   "
        | Node (n, h, l, r) ->
              let make_space = String.map (fun _ -> ' ') in
              let sn = string_of_int n in
              let sl = strign_of_avltree_level lvl l in
              let sr = strign_of_avltree_level lvl r in
                  if h = lvl
                  then make_space sl ^ sn ^ make_space sr
                  else sl ^ make_space sn ^ sr
    in
    let rec print_levels lvl =
        if lvl >= 0
        then (print_string (string_of_int lvl ^ ": " ^ strign_of_avltree_le
            print_levels (lvl-1))
        else ()
    in
    let height = function
        | Node (_, y, _, _) -> y
        | Empty -> 0
    in
    print_levels (height t) ; flush stdout
```

Naloga: definirajte AVL drevo `t` :



In [ ]:
```
let t = Node (5, 3,
            Node (3, 2,
                Node(1, 1, Empty, Empty),
                Node(4, 1, Empty, Empty)),
            Node(8, 2, Empty, Empty))
(*

showTreeHorizontally t ;;
3:                5
2:        3           8
1:    1       4
0: nil nil nil nil nil nil

 *)
```

Pametni konstruktor

Naloga: definirajte funkcijo `height : avltree -> int` , ki vrne višino drevesa. Primera:

```
# height Empty ;;
- : int = 0
# height t ;;
- : int = 3
```

In [ ]:
```
let height : avltree -> int = fun x ->
    match x with
        | Empty -> 0
        | Node(v, h, l, r) -> h

let drugi = fun(_, d, _, _) -> d
```

Naloga: definirajte "pametna" konstruktorja `leaf: int -> avltree` in `node: int * avltree * avltree -> avltree` , ki sama poskrbita za višino. Prav vam bo prišla funkcija `max: int -> int -> int` , ki vrne večjega od dveh števil. Primer uporabe:

```
# let six = leaf 6 ;;
val six : avltree = Node (6, 1, Empty, Empty)
# let seven = leaf 7 ;;
val seven : avltree = Node (7, 1, Empty, Empty)
# let answer = node (42, six, seven) ;;
val answer : avltree =
   Node (42, 2, Node (6, 1, Empty, Empty), Node (7, 1, Empty,
Empty))
```

In [ ]:
```
let leaf : int -> avltree = fun x -> Node(x, 1, Empty, Empty)

let leaf x = Node(x, 1, Empty, Empty)

let node : int * avltree * avltree -> avltree = fun(v, l, r) -> Node(v, 1 +

let node (v, l, r) = Node(v, 1 + max (height l) (height r), l, r)
```

Naloga: s pametnimi konstruktorji definirajte AVL drevo, enako drevesu `t` iz prejšnje naloge.

In [ ]:
```
let t = node(5,
             node (3,
                 leaf 1,
                 leaf 4),
             leaf 8) ;;
```

Drevo ⇒ seznam

Naloga: definirajte funkcijo `toList: avltree -> int list` , ki elemente drevesa vrne kot urejen seznam števil. Za združevanje seznamov ima OCaml operator `@` .

Z enim obhodom - `inorder` (ohranja urejenost elementov) se sprehodimo po drevesu in naredimo seznam. Najprej v obliki seznama izpišemo levo poddrevo, nato dodamo element v trenutnem vozlišču ter na koncu dodamo še tisto kar je na desni strani.

Operator `@` sprejme en argument na levi in en argument na desni.

In [ ]:

```
let rec toList t =
    match t with
        | Empty -> []
        | Node(v, _, l, r) -> toList l @ v :: toList r

let rec toList t =
    match t with
        | Empty -> []
        | Node(v, _, l, r) -> toList l @ [v] @ toList r

let rec toList t =
    match t with
        | Empty -> []
        | Node(v, _, l, r) ->
            let ls = toList l in
                let rs = toList r in
                    ls @ [v] @ rs

(*

toList t ;;
- : value list = [1; 3; 4; 5; 8]

*)
```

Iskanje

Algoritem, ki ugotovi, ali je dani `x` v drevesu `t`:

- če je `t` prazno drevo, se `x` ne pojavi;
- če je `t` vozlišče z vsebino `y` in poddrevesoma `l` ter `r`:
  - če je `x = y`, se `x` pojavi;
  - če je `x < y`, iščemo v poddrevesu `l`;
  - če je `x > y`, iščemo v poddrevesu `r`.

Iskanje bomo implementirali s funkcijo `search`, ki naj deluje tako:

```
# search 5 t ;;
- : bool = true
```

Naloga: zapišite *tip* funkcije `search`, ki ugotovi, ali drevo `t` vsebuje vrednost `x`.

In [ ]:
```
search : value -> avltree -> bool
```

Naloga: definirajte funkcijo `search`. Za primerjanje dveh števil uporabite funkcijo `cmp` s predavanj, ki vrne vrednost tipa `order`:

In [ ]:
```
type order = Less | Equal | Greater

let cmp x y =
    match compare x y with
    | 0 -> Equal
    | r when r < 0 -> Less
    | _ -> Greater
```

Ujemanje vzorcev na drevesu.

In [ ]:
```
(* počasna rešitev *)
let search_slow e t = List.mem e (toList t)

let search v t =
    match t with
        | Empty -> false
        | Node(x, _, l, r) ->
            (
                match compare v x with
                    | Equal -> true
                    | Less -> search v l
                    | Greater -> search v r
            )

let rec search v t =
    match t with
        | Empty -> false
        | Node(x, _, l, r) ->
            begin match cmp v x with
                    | Equal -> true
                    | Less -> search v l
                    | Greater -> search v r
            end
```

Naloga: preizkusite, ali `search` deluje.

In [ ]:
```
search 5 t;;
(* - : bool = true *)
```

Vrtenje in uravnoteženje

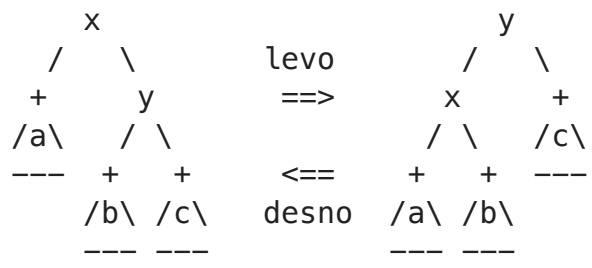
Pri vstavljanju ali brisanju elementov se lahko AVL drevo "pokvari": višina levega in desnega poddrevesa nekega poddrevesa se razlikujeta za več kot 1. To popravimo z vrtenjem (rotacijo) drevesa okrog korena.

Naloga: definirajte funkcijo `imbalance: avltree -> int` , ki vrne stopnjo neuravnoteženosti drevesa, tj. razliko med višinama levega in desnega poddrevesa.

```
# imbalance t ;;
- : int = 1
# imbalance Empty ;;
- : int = 0
```

In [ ]:
```
let imbalance t =
    match t with
        | Empty -> 0
        | Node(_, _, l, r) -> height l - height r

(*
imbalance t ;;
- : int = 1
*)
```

V AVL drevesu je lahko neuravnoteženost kateregakoli vozlišča največ 1. Bolj neuravnotežena poddrevesa lahko popravimo z vrtenjem v levo oziroma desno. Vrtenje v levo oziroma desno ponazorimo z diagramom



Naloga: definirajte funkciji `rotateLeft` in `rotateRight`, ki dano drevo zavrtita na prikazan način. Če to ni mogoče (ker je drevo npr. prazno ali list), naj funkciji vrneta nespremenjeno drevo.

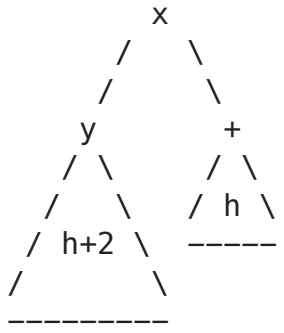
Na desni strani ne more biti konstruktor!

Ni rekurzivna, ker se izvede v enem koraku.

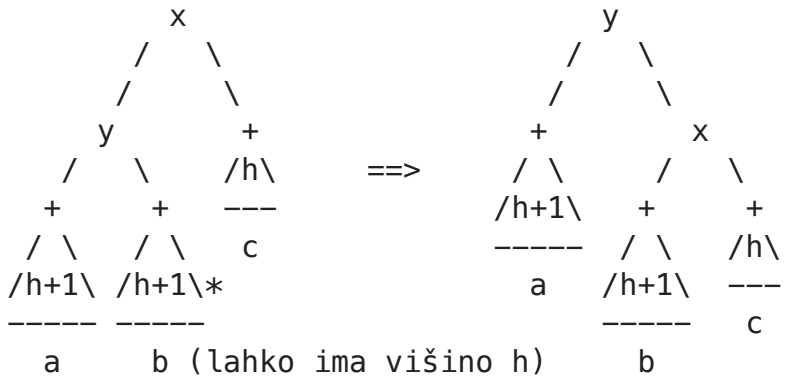
Beseda `function` naredi anonimno funkcijo, argumenti pa začnejo direktno izvajati ujemanje vzorcev.

In [ ]:
```
let rotateLeft : avltree -> avltree = function
    | Node(x, _, a, Node(y, _, b, c)) -> node(y, node(x, a, b), c)
    | tree -> tree

let rotateRight : avltree -> avltree = function
    | Node(y, _, Node(x, _, a, b), c) -> node(x, a, node(y, b, c))
    | tree -> tree
```

Funkciji `rotateLeft` in `rotateRight` uporabimo za uravnoteženje drevesa. Ker bomo drevo uravnotežili po vsakem vstavljanju in brisanju elementa, lahko predpostavimo, da bo neuravnoteženost drevesa kvečjemu 2. Če je neuravnoteženost enaka 0, 1 ali -1, ni treba storiti ničesar. Sicer je neuravnoteženost 2 oziroma -2. V prvem primeru imamo drevo oblike

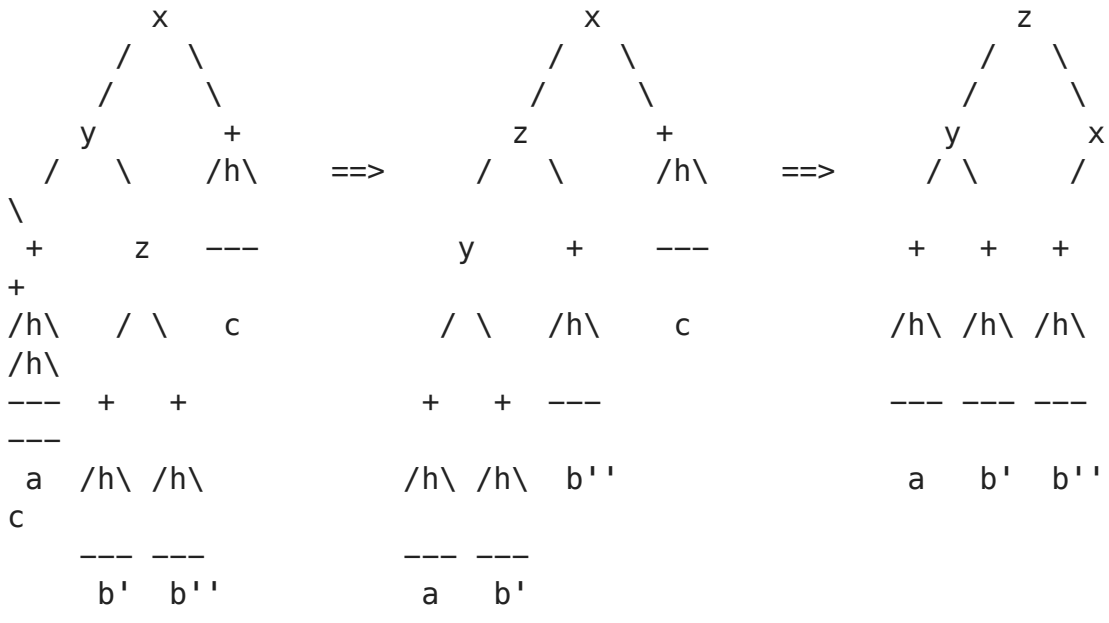


Glede na višini poddreves vozlišča y lahko to drevo uravnotežimo z enim ali dvema vrtenjema. V prvem primeru imamo drevo oblike



ki ga popravimo z enim vrtenjem v desno, kot je prikazano zgoraj.

V drugem primeru imamo drevo oblike



ki ga popravimo tako, da najprej levo poddrevo (s korenom `y`) zavrtimo v levo, nato pa celotno drevo zavrtimo v desno.

Če je neuravnoteženost drevesa enaka -2, postopamo simetrično.

Naloga: definirajte funkcijo `balance: avltree -> avltree`, ki uravnoteži AVL drevo na podlagi opisanih primerov.

In [ ]:
```
let balance t =
    match t with
        | Empty -> Empty
        | Node(v, _, l, r) ->
            (
                match imbalance t with
                | (0 | -1 | 1) -> t
                | -2 -> (* nagnjeno v desno *)
                (
                    match imbalance r with
                    | 1 -> rotateLeft (node(v, l, rotateRight r))
                    | (0 | -1) -> rotateLeft l
                    | _ -> failwith "Tega ne moremo popraviti"
                )
                | 2 ->
                (
                    match imbalance l with
                    | -1 -> rotateRight (node(v, rotateLeft l, r))
                    | (0 | 1) -> rotateRight t
                    | _ -> failwith "Tega ne moremo popraviti"
                )
                | _ -> failwith "Tega ne moremo popraviti"
            )
```

Vstavljanje

Nov element `x` vstavimo v AVL drevo `t` po naslednjem postopku:

- če je drevo prazno, vrni list z vsebino `x`;
- če je `t` vozlišče z vsebino `y` in poddrevesoma `l` ter `r`:
  - če je `x = y`, vrni `t`;
  - če je `x < y`, vstavi `x` v `l` in rezultat uravnoteži;
  - če je `x > y`, vstavi `x` v `r` in rezultat uravnoteži.

Naloga: definirajte funkcijo `add: int -> avltree -> avltree`.

In [ ]:
```
let rec add x t =
    match t with
        | Empty -> leaf x
        | Node(y, _, l, r) ->
            match cmp x y with
                | Equal -> t
                | Less -> balance (node(y, add x l, r))
                | Greater -> balance (node(y, l, add x r))
```

Brisanje

Element `x` iz AVL drevesa izbrišemo tako, da ga zamenjamo z njegovim naslednikom iz poddrevesa v vozlišču `x`. Pri tem bomo uporabili pomožno funkcijo `removeSuccessor`, ki vrne novo drevo in naslednika, deluje pa tako:

- če je `t` prazno drevo, sproži izjemo;
- če je `t` vozlišče z vsebino `x` in poddrevesoma `l` ter `r`:
  - če je `l` prazno drevo, vrni `(r, x)`;
  - sicer izbriši naslednika iz `l`, da dobiš `(l', y)`, nato pa sestavi in uravnoteži novo drevo s korenom `x` ter poddrevesoma `l'` in `r`.

Kaj je `successor`? Nahaja se v desnem poddrevesu in je najmanjši element.

Naloga: definirajte funkcijo `removeSuccessor: avltree -> avltree * int`.

In [ ]:
```
let rec removeSuccessor : avltree -> avltree * int = function
    | Empty -> failwith "cant find successor"
    | Node (x, _, Empty, r) -> (r, x)
    | Node (x, _, l, r) ->
        let (l', y) = removeSuccessor l in
            (balance (node(x, l', r)), y)
```

Element `x` lahko potem izbrišemo iz AVL drevesa `t` po naslednjem postopku:

- če je `t` prazno drevo, vrni `t`;
- če je `t` vozlišče z vsebino `y` in poddrevesoma `l` in `r`:
  - če je `x < y`, izbriši `x` iz levega poddrevesa `l` in rezultat uravnoteži;
  - če je `x > y`, izbriši `x` iz desnega poddrevesa `r` in rezultat uravnoteži;
  - če je `x = y`:
    - če je desno poddrevo prazno, vrni `l`;
    - če je levo poddrevo prazno, vrni `r`;
    - sicer iz `r` izbriši naslednika, da dobiš `(r', y')`, nato pa sestavi in uravnoteži novo drevo s korenom `y'` ter poddrevesoma `l` in `r'`.

Naloga: definirajte funkcijo `remove: int -> avltree -> avltree`.

In [ ]:

```
let rec remove x t =
    match t with
        | Empty -> t
        | Node(y, _, l, r) ->
            match cmp x y with
                | Less -> balance (node(y, remove x l, r))
                | Greater -> balance (node(y, l, remove x r))
                | Equal ->
                    match (l, r) with
                        | (_, Empty) -> l
                        | (Empty, _) -> r
                        | (_, _) ->
                            let (r', y') = removeSuccessor r in
                                balance(node(y', l, r'))
```

In [ ]:

```
(* testi *)
let tr = Empty;;
let _ = showTreeHorizontally tr;;

let tr = add 1 tr;;
let test1 = tr = Node (1, 1, Empty, Empty) ;;
let _ = showTreeHorizontally tr;;


let tr = add 2 tr;;
let test2 = tr = Node (1, 2, Empty, Node (2, 1, Empty, Empty))
let _ = showTreeHorizontally tr;;

let tr = add 3 tr;;
let test3 = tr = Node (2, 2, Node (1, 1, Empty, Empty), Node (3, 1, Empty,
let _ = showTreeHorizontally tr;;

let tr = add 4 tr;;
let test4 = tr = Node (2, 3, Node (1, 1, Empty, Empty),
   Node (3, 2, Empty, Node (4, 1, Empty, Empty)))
let _ = showTreeHorizontally tr;;

let tr = add 5 tr;;
let test5 = tr = Node (2, 3, Node (1, 1, Empty, Empty),
   Node (4, 2, Node (3, 1, Empty, Empty), Node (5, 1, Empty, Empty)))
let _ = showTreeHorizontally tr;;

let tr = add 6 tr;;
let test6 = tr =  Node (4, 3,
   Node (2, 2, Node (1, 1, Empty, Empty), Node (3, 1, Empty, Empty)),
   Node (5, 2, Empty, Node (6, 1, Empty, Empty)))
let _ = showTreeHorizontally tr;;

let tr = add 7 tr;;
let test7 = tr = Node (4, 3,
   Node (2, 2, Node (1, 1, Empty, Empty), Node (3, 1, Empty, Empty)),
   Node (6, 2, Node (5, 1, Empty, Empty), Node (7, 1, Empty, Empty)))
let _ = showTreeHorizontally tr;;
```

```
let tr = remove 1 tr;;
let test_remove1 = tr = Node (4, 3, Node (2, 2, Empty, Node (3, 1, Empty, E
   Node (6, 2, Node (5, 1, Empty, Empty), Node (7, 1, Empty, Empty)))
let _ = showTreeHorizontally tr;;

let tr = remove 2 tr;;
let test_remove1 = tr = Node (4, 3, Node (3, 1, Empty, Empty),
   Node (6, 2, Node (5, 1, Empty, Empty), Node (7, 1, Empty, Empty)))
let _ = showTreeHorizontally tr;;

let tr = remove 3 tr;;
let test_remove1 = tr = Node (6, 3, Node (4, 2, Empty, Node (5, 1, Empty, E
   Node (7, 1, Empty, Empty))
let _ = showTreeHorizontally tr;;

let tr = remove 4 tr;;
let test_remove1 = tr = Node (6, 2, Node (5, 1, Empty, Empty), Node (7, 1,
let _ = showTreeHorizontally tr;;

let tr = remove 5 tr;;
let test_remove1 = tr = Node (6, 2, Empty, Node (7, 1, Empty, Empty))
let _ = showTreeHorizontally tr;;

let tr = remove 6 tr;;
let test_remove1 = tr = Node (7, 1, Empty, Empty)
let _ = showTreeHorizontally tr;;

let tr = remove 7 tr;;
let test_remove1 = tr = Empty
let _ = showTreeHorizontally tr;;
```

Vir.

Rekurzija

Spoznali bomo povezavo med zankami in rekurzijo. Naučili se bomo tudi nekaj tehnik programiranja z rekurzivnimi funkcijami.

Iz zanke v rekurzijo

Reference in zanke v Ocamlu

Program, ki sešteje prvih `42` lihih števil:

In [ ]:
```
let vsota_lihih n =
    let sum = ref 0 in
    let i = ref 0 in
    while !i < n do
        sum := !sum + (2 * !i + 1) ;
        i := !i + 1
    done ;
    !sum
```

Števec `i` in vsota `sum` sta referenci, ker se njuni vrednosti spreminjata. To je običajno, kadar uporabljamo zanke. Rezultat ukazov v telesu `while` je tipa `unit`. Stranki učinek pa je, da imata sedaj `i` in `sum` neke druge vrednosti.

`;` operator, ki ga uporabljamo pri združevanju izrazov.

Naloga 1

Sestavite funkcijo `vsota1 : int -> int`, ki sprejme `n` in vrne vsoto `1 + 2 + ⋯ + n`. Uporabite reference in zanko `while`.

Rešitev

In [ ]:
```
let vsota1 n =
    let sum = ref 0 in
    let i = ref 0 in
    while !i < n do
        sum := !sum + 1 ;
        i := !i + 1
    done ;
    !sum
```

Naloga 2

Sestavite funkcijo `fibonacci1 : int -> int`, ki sprejme `n` in vrne `n`-to Fibonaccijevo število `F(n)`. Nauk: Fibonaccijevo zaporedje je definirano s predpisom:

```
F(0) = 0
F(1) = 1
F(n) = F(n-1) + F(n-2)
```

Uporabite reference in zanko `while`.

Rešitev

In [ ]:

```
let fibonacci1 n =
    let current = ref 1 in
    let prev = ref 1 in
    let i = ref 2 in
    while !i < n do
        let tmp = !current in
            current := !current + !prev;
            prev := tmp;
            i := !i + 1
    done ;
    !current
```

Rekurzivne funkcije

Zanko `while` lahko sistematično pretvorimo v rekurzivno funkcijo. Še prej pa malce ponovimo rekurzivne funkcije.

Naloga 3

Sestavite funkcijo `vsota2 : int -> int`, ki sprejme `n` in vrne vsoto `1 + 2 + ⋯ + n`. Funkcija naj bo rekurzivna in naj ne uporablja zank in referenc.

Rešitev

In [ ]:

```
let rec vsota2 n =
    match n with
        | 0 -> 0
        | n -> n + vsota2(n - 1)

let rec vsota2 : int -> int = fun n ->
    if n < 2 then n
    else n + vsota2(n - 1)
```

Naloga 4

Sestavite funkcijo `fibonacci2 : int -> int` , ki sprejme `n` in vrne `n` -to Fibonaccijevo število `F(n)` . Funkcija naj bo rekurzivna in naj ne uporablja zank in referenc.

Rešitev

In [ ]:
```
let rec fibonacci2 n =
    match n with
        | 0 -> 0
        | 1 -> 1
        | n -> fibonacci2(n - 1) + fibonacci2(n - 2)
```

Akumulatorji in repna rekurzija

Pravimo, da je rekurzivni klic funkcije **repni klic** ali **klic na repu** (angl. *tail recursive*), če je rezultat rekurzivnega klica hkrati tudi rezultat funkcije. Povedano z drugimi besedami, funkcija se pokliče rekurzivno in nato takoj vrne rezultat rekurzivnega klica.

Na primer, v rekurzivni funkciji

In [4]:
```
let rec f = function
| 0 -> 1
| n ->
    if n mod 2 = 0
    then f (n / 2)
    else 3 * f (n - 1)
```

Out[4]:
```
val f : int -> int = <fun>
```

Funkcijo `vsota_lihih1 n` , ki z uporabo zanke izračuna vsoto

$$1 + 3 + 5 + \cdots + (2\,n - 1)$$

pretvorimo v repno rekurzivno funkcijo:

In [ ]:

```
let vsota_lihih1 n =
    let v = ref 0 in
    let i = ref 0 in
    while !i < n do
        v := !v + (2 * !i + 1) ;
        i := !i + 1
    done ;
    !v
(* val vsota_lihih1 : int -> int = <fun> *)

let vsota_lihih2 n =
    let rec vsota v i =
        if i < n
            then vsota (v + (2 * i + 1)) (i + 1)
        else v
    in
    vsota 0 0
```

Naredimo pomožno rekurzivno funkcijo, ki sprejme več argumentov in enega izmed teh argumentov uporabi kot akumulator.

Akumulator predstavlja trenutno vsoto in ga uvozimo iz rekurzije v rekuzijo. Ko pridemo do končnega pogoja, vrnemo akumulator.

Ker zanka uporablja dve spremenljivki `v` in `i`, ima rekurzivna funkcija dva argumenta `v` in `i`. Namesto, da spreminjamo vrednosti `v` in `i` (česar ne moremo, saj `v` in `i` nista več referenci), naredimo repni rekuzivni klic s popravljenima vrednostma `v` in `i`. Akumulator je `v`.

`i` šteje od `0` do `n`. `v` je rezultat trenutnega rekurzivnega klica. `(2 * i + 1)` je rezultat naslednjega rekurzivnega klica. Direktne rekurzivne definicije ne moremo napisati, zato smo definirali `i` in `v`.

Funkcija `vsota_lihih2` ni rekurzvina, ampak vsebuje *pomožno* rekurzivno funkcijo `vsota`, ki igra vlogo zanke. Klic `vsota 0 0` nato izvede `vsota` z ustreznima začetnima vrednostma `v` in `i`.

Naloga 3

Funkcijo `vsota1` spremenimo v funkcijo `vsota3`, ki uporablja akumulatorje in repno rekurzijo.

Rešitev

In [7]:
```
let vsota3 n =
    let rec vsota v i =
        if i <= n
        then vsota (v + i) (i + 1)
        else v
    in
    vsota 0 0

(* val vsota3 : int -> int = <fun> *)
```

Naloga 4

Po zgornjem receptu predelajte funkcijo `fibonacci1` v funkcijo `fibonacci3`, ki uporablja akumulatorje in repno rekurzijo.

Rešitev

In [ ]:
```
let fibonacci3 n =
    let rec fib i curr next =
        if i < n
        then fib (i + 1) (next) (curr + next)
        else
            curr
    in
    fib 0 0 1
```

Splošna pretvorba zanke `while` v rekurzivno funkcijo

Premislimo še, ali lahko zanko `while` v splošnem prevedemo v rekurzivno funkcijo z akumulatorjem in repnim klicom. Obravnavajmo zanko `while` oblike (zapisali smo jo v namišljenem ukaznem programskem jeziku):

```
s := s0
while p(s) do
  s := f(s)
done ;
return r(s)
```

Tu smo z `s` označili skupno stanje vseh spremenljivk, ki nastopajo v zanki, s_0 je začetno stanje, `p(s)` je pogoj (ki je odvisen od stanja `s`), zanka vsakič trenutno stanje `s` nastavi na novo stanje `f(s)`, na koncu pa vrne rezultat `r(s)` (ki je spet odvisen od `s`). Na primer, zanko, ki izračuna `n`-to Fibonaccijevo število bi zapisali takole:

```
(a,b,i) := (0,1,0)
while i < n do
   (a,b,i) = (b,a+b,i+1)
done ;
return a
```

Zgornjo zanko lahko predelamo v rekurzivno funkcijo `zanka`, ki spreme s_0, `p`, `f` in `r` ter izračuna to, kar bi sicer izračunala zanka `while`:

In [8]:
```
let zanka s0 p f r =
    let rec loop s =
        if p s then loop (f s) else r s
    in loop s0
```

Out[8]:
```
val zanka : 'a -> ('a -> bool) -> ('a -> 'a) -> ('a -> 'b) -> 'b = <fun>
```

Tip funkcije `zanka` je

$$\alpha \to (\alpha \to \text{bool}) \to (\alpha \to \alpha) \to (\alpha \to \beta) \to \beta$$

kar pomeni, da ima začetno stanje s_0 (poljuben) tip α, pogoj `p` je funkcija, ki slika stanje v `bool`, `f` je funkcija, ki stanje predela v novo stanje, in `r` je funkcija, ki stanje predela v rezultat (poljubnega) tipa β.

Naloga 5

Sestavite funkcijo `vsota4`, ki izračuna vsoto števil `1 + 2 + ⋯ + n`, tako da uporabite funkcijo `zanka`. Torej, vaša rešitev mora biti oblike

```
let vsota4 n = zanka ⋯ ⋯ ⋯ ⋯
```

kjer ⋯ nadomestite z ustreznimi vrednostmi s_0, `p`, `f` in `r`.

Rešitev

In [ ]:
```
let vsota4 n = zanka
    (0, 0) (* so, začetno stanje za i in v *)
    (fun(i, v) -> i <= n) (* p, pogoj *)
    (fun(i, v) -> (i + 1, v + i)) (* f, stanje predela v novo stanje *)
    (fun(i, v) -> v) (* r, stanje predela v rezultat *)
```

Naloga 6

Sestavite funkcijo `fibonacci4`, ki izračuna `n`-to Fibonaccijevo, tako da uporabite funkcijo `zanka`. Torej, vaša rešitev mora biti oblike

```
let fibonacci4 n = zanka ⋯ ⋯ ⋯ ⋯
```

kjer ⋯ nadomestite z ustreznimi vrednostmi s_0, `p`, `f` in `r`.

Rešitev

In [ ]:

```
let fibonacci3 n =
    let rec fib i curr next =
        if i < n
        then fib (i + 1) (next) (curr + next)
        else
            curr
    in
    fib 0 0 1

let fibonacci3 n = zanka
    (0, 0, 1)
    (fun(i, curr, next) -> i < n)
    (fun(i, curr, next) -> (i + 1, next, curr + next))
    (fun(i, curr, next) -> curr)
```

Naloga 7

Sestavite rekurzivno funkcijo `forzanka`, ki izračuna to, kar izračuna spodnja koda v namišljenem ukaznem programskem jeziku:

```
s := s0
for i = a to b do
  s := f(i, s)
done ;
return r(s)
```

Enako kot prej tukaj `s` označuje skupno stanje vseh spremenljivk, ki nastopajo v zanki, s_0 pa začetno stanje. Zanka za vsak `i` med vključno `a` in `b` pokliče funkcijo `f`, ki glede `i` in trenutno stanje `s` izračuna novo stanje, na koncu pa vrne rezultat `r(s)`. Na primer, zanko `for`, ki izračuna vsoto prvih `n` naravnih števil, bi zapisali takole:

```
v := 0
for i = 1 to n do
    v := v+i
done ;
return v
```

Funkcija `forzanka` naj prejme začetno stanje s_0, spodnjo in zgornjo mejo `a` oziroma `b` ter funkciji `f` in `r`. Njen tip bo torej

```
α → int → int → (int → α → α) → (α → β) → β
```

Rešitev

In [ ]:
```
let forzanka s0 a b f r =
    let rec loop s i =
        if i > b then
            r s
        else
            loop (f i s) (i + 1)
    in
    loop s0 a
```

Naloga 8

Sestavite funkcijo `fibonacci5`, ki izračuna `n`-to Fibonaccijevo, tako da uporabite funkcijo `forzanka`. Torej, vaša rešitev mora biti oblike

```
let fibonacci5 n = forzanka ... ... ... ... ...
```

kjer `...` nadomestite z ustreznimi vrednostmi s_0, `a`, `b`, `f` in `r`.

Rešitev

In [ ]:
```
let fibonacci5 n =
    forzanka
    (0, 1)
    1
    n
    (fun (a ,b) -> (b, a + b))
    (fun (a, b) -> a)
```

Specifikacija, implementacija, abstrakcija

Specifikacija, implementacija

In [ ]:

```
(* Definiramo signaturo (specifikacijo). *)

module type GROUP =
sig
    type g
    val mul : g * g -> g
    val inv : g -> g
    val e : g
end

module type DIRECTED_GRAPH =
sig
    type v
    type e
    val src : e -> v
    val trg : e -> v
end

(* Signaturo implementiramo z modulom ali strukturo.
Dana signatura ima lahko več implementacij. *)
module Z3 : GROUP =
struct

  type g = Zero | One | Two

  let e = Zero

  let plus = function
    | (Zero, y) -> y
    | (x, Zero) -> x
    | (One, One) -> Two
    | (One, Two) -> Zero
    | (Two, One) -> Zero
    | (Two, Two) -> One

  let mul = plus

  let inv = function
    | Zero -> Zero
    | One -> Two
    | Two -> One
end

module C5 = Cycle(struct let n = 5 end)
```

Generično programiranje

In [ ]:
```
(* preslikava, ki vsebuje celo število v directed_graph*)
module Cycle (S : sig val n : int end) : DIRECTED_GRAPH =
  struct
    type v = int (* uporabimo 0, 1, 2, ..., S.n-1 *)
    type e = int (* uporabimo 0, 1, 2, ..., S.n-1 *)
    let src k = k
    let trg k = (k + 1) mod S.n end

module C5 = Cycle(struct let n = 5 end)
module Ckrneki = Cycle(struct let n = 42 end)
```

Objekti

In [ ]:
```
module type POINT =
  sig
    val x : float
    val y : float
  end

module type COLOR_POINT =
  sig
    val x : float
    val y : float
    val c : color (* dodatna funkcionalnost *)
  end

(* Color_point je podtip od point. *)

module F (P : POINT) =
  struct
    let z = P.x +. P.y
  end
```

Prioritetna vrsta (naloga)

Prioritetna vrsta je podatkovna struktura, v katero dodajamo elemente, ven pa jih jemljemo glede na njihovo prioriteto.

Signatura:

- podatkovni tip element
- operacija priority : element → int
- podatkovni tip queue
- konstanta empty : queue
- operacija put : element → queue → queue
- operacija get : queue → element option * queue

Zahteve:

- element je tip elementov, ki jih hranimo v vrsti
- priority x vrne prioriteto elementa x, ki je celo število. Manjše število pomeni »prej na vrsti«
- queue je tip prioritetnih vrst
- empty je prazna prioritetna vrsta, ki ne vsebuje elementov
- put x q vstavi element x v vrsto q glede na njegovo prioriteto in vrne tako dobljeno vrsto
- get q vrne (Some x, q') kjer je x element iz q z najnižjo prioriteto in q' vrsta q brez x Operacija get vrne (None, q), če je q prazna vrsta.

In [ ]:

```
(* specifikacija *)

module type PRIORITY_QUEUE =
  sig
    type element
    val priority : element -> int
    type queue
    val empty : queue
    val put : element -> queue -> queue
    val get : queue -> element option * queue
  end

(* implementacija - abstrakcija *)
(* to kar vemo o MyFirstQueue je to, da je PRIORITY_QUEUE *)
(* Zapisati moramo, katerega tipa je element! *)
module MyFirstQueue : PRIORITY_QUEUE with type element = int * int =
  struct
    type element = int * int (* urejeni pari števil*)

    let priority (a, b) = a

    type queue = element list (* seznam elementov *)
```

```
    let empty = []

    let rec put x = function
      | [] -> [x] (* vrsta prazna --> vrsta, ki vsebuje x*)
      (* vrsta: na začetku y, sledijo y --> primerjamo priotiteto x in y *
      | y :: ys ->
         if priority x <= priority y then
           x :: y :: ys
         else
           y :: put x ys

    let get = function
      (* prazna vrsta --> vrnemo None in isto vrsto *)
      | [] -> (None, [])
      (* vrsta ima na začetlu x in preostanek x --> prvi element je x, pote
      | x :: xs -> (Some x, xs)
  end

(* Vmesnik za elemente, ki jim računa prioriteto. *)

module type PRIORITY =
sig
  type t (* element t*)
  val priority : t -> int (* vsakemu elementu priredi število *)
end


(*
  Implementacija prioritetne vrste s seznami.
  To je funktor, ki sprejme tip elemetov in prioritetno funkcijo.
  Podobno kot prej le, da povsod zamenjamo stvari z M.
*)

(* Modul M - dobimo neke elemente, ki zadoščajo vmesniku PRIORITY *)
(* : ... kaj bomo programirali - PRIORITY_QUEUE in dodatno povemo, da bodo
(* Če ne povemo tipa alemeneta --> nanpaka: abstract interface *)
module ListQueue (M : PRIORITY) : PRIORITY_QUEUE with type element = M.t =
  struct
    type element = M.t

    let priority = M.priority

    type queue = element list

    let fortytwo = 42
    (* se ne bidi, ker jih bo abstraktni vmesnik prekril - vidi se samo tis

    let empty = []

    let rec put x = function
      | [] -> [x]
      | y :: ys ->
         if priority x <= priority y then
           x :: y :: ys
         else
           y :: put x ys
```

```
    let get = function
      | [] -> (None, [])
      | x :: xs -> (Some x, xs)
  end

(* Naredimo prioritetno vrsto nizov, prioriteta je dolžina niza. *)
(* Naredimo strukturo in damo notri podatke. *)
module A =
  ListQueue(
      struct
        type t = string
        let priority = String.length
      end)

(* Preizkus. *)
let example1 =
  A.get (A.put "kiwi" (A.put "jabolko" (A.put "banana" A.empty)))

(* Naredimo prioritetno vrsto nizov, prioriteta je dolžina niza. *)
module A' =
  ListQueue(
      struct
        type t = string
        let priority s = - String.length s
      end)

(* Preizkus. *)
let example1' =
  A'.get (A'.put "kiwi" (A'.put "jabolko" (A'.put "banana" A'.empty)))

(* Naredimo prioritetno vrsto parov števil. *)
module B =
  ListQueue(
      struct
        type t = int * int
        let priority (a,b) = a
      end
    )

module IntQueue =
  ListQueue(
    struct
      type t = int
      let priority k = k
    end
  )
```

Množice

In [ ]:

```
(* Pomožni tip, funkcija in signatura za lepše primerjanje. *)
type order = Less | Equal | Greater

let ocaml_cmp x y =
  let c = Stdlib.compare x y in
  if c < 0 then Less
  else if c > 0 then Greater
  else Equal

module type ORDERED =
  sig
    type t
    val cmp : t -> t -> order
  end

(* Specifikacija podatkovnega tipa množica. *)
module type SET =
  sig
    (* znotraj podpisa mora biti definiran podatkovni tip element, na zuna
    type element
    val cmp : element -> element -> order
    type set
    val empty : set
    val member : element -> set -> bool
    val add : element -> set -> set
    val remove : element -> set -> set
    val to_list : set -> element list
  end
```

V množici se vsak element lahko pojavi samo enkrat. Tip elementov, ki jih vsebuje množica, mora biti primerljiv – elemente primerjamo s funkcijo `cmp`, ki vrne `Less`, `Equal` ali `Greater`. Množica podpira naslednje operacije:

- `member e s` vrne `true`, če množica `s` vsebuje element `e`,
- `add e s` doda element `e` v množico `s`,
- `remove e s` izbriše element `e` iz množice `s`
- `to_list s` vrne seznam elementov v množici `s` (v poljubnem vrstnem redu).

Naloga 1: Implementacija s seznamom

Definirajte strukturo `IntListSet`, ki elemente množice hrani v seznamu. Tip elementov naj bo celo število (`int`), za primerjavo pa lahko uporabite kar funkcijo `ocaml_cmp`:

In [ ]:

```
module IntListSet : SET with type element = int =
  struct
    (* type element *)
    type element = int (* lahko ker smo napisali type element = int *)

    (* val cmp : element -> element -> order *)
    let cmp = ocaml_cmp

    (* ni viden zunaj katerega tipa je, ker ni specifikaciji *)
    (* type set *)
    type set = element list (* lahko tudi int list *)

    (* val empty : set *)
    let empty = []

    (* val member : element -> set -> bool *)
    (* let member = List.mem *)
    let eq x y = cmp x y = Equal
    let rec member e = function
    | [] -> false
    | x :: xs -> eq e x || member e xs

    (* val add : element -> set -> set *)
    let add e s =
      if member e s then s
      else e :: s

    (* val remove : element -> set -> set *)
    val remove e = List.filter(fun x -> not (eq e x))

    let rec remove e = function
    | [] -> []
    | x :: xs -> if eq x e then xs else x :: remove e xs

    (* val to_list : set -> element list *)
    let to_list s = s
  end

  (*

  module IntListSet :
  sig
    type element = int
    val cmp : element -> element -> order
    type set
    val empty : set
    val member : element -> set -> bool
    val add : element -> set -> set
    val remove : element -> set -> set
    val to_list : set -> element list
  end

  *)
```

Primer uporabe:

```
# let s = IntListSet.add 2 (IntListSet.add 4
IntListSet.empty) ;;
val s : IntListSet.set = <abstr>
# IntListSet.member 4 s ;;
- : bool = true
```

Naloga 2: Generični `ListSet`

Definicijo `IntListSet` spremenite v funktor `ListSet`, ki sprejme strukturo, ki zadošča signaturi

In [ ]:
```
module type ORDERED =
    sig
        type t
        val cmp : t -> t -> order
    end
```

Primer uporabe:

```
# module S = ListSet(struct type t = string  let cmp =
ocaml_cmp end) ;;
module S :
  sig
    type element = string
    val cmp : element -> element -> order
    type set
    val empty : set
    val member : element -> set -> bool
    val add : element -> set -> set
    val remove : element -> set -> set
    val to_list : set -> element list
    val fold : ('a -> element -> 'a) -> 'a -> set -> 'a
  end
# let s = S.add "foo" (S.add "bar" S.empty) ;;
val s : S.set = <abstr>
# S.member "foo" s ;;
- : bool = true
```

In [ ]:

```
module ListSet(M : ORDERED) : SET with type element = M.t =
struct
  (* type element *)
  type element = M.t

  (* val cmp : element -> element -> order *)
  let cmp = M.cmp

  (* ni viden zunaj katerega tipa je, ker ni specifikaciji *)
  (* type set *)
  type set = element list

  (* val empty : set *)
  let empty = []

  (* val member : element -> set -> bool *)
  (* let member = List.mem *)
  let eq x y = cmp x y = Equal
  let rec member e = function
  | [] -> false
  | x :: xs -> eq e x || member e xs

  (* val add : element -> set -> set *)
  let add e s =
    if member e s then s
    else e :: s

  (* val remove : element -> set -> set *)
  val remove e = List.filter(fun x -> not (eq e x))

  let rec remove e = function
  | [] -> []
  | x :: xs -> if eq x e then xs else x :: remove e xs

  (* val to_list : set -> element list *)
  let to_list s = s
end

module S = ListSet(struct type t = string  let cmp = ocaml_cmp end) ;;
let s = S.add "foo" (S.add "bar" S.empty) ;;
S.member "foo" s ;;
(* val s : S.set = <abstr> *)
```

Naloga 3: Primerjava učinkovitosti `ListSet` in `AVLSet`

Modul `AVLSet` implementira funktor, ki tako kot `ListSet` sprejme strukturo s signaturo `ORDERED` in vrne implementacijo množic z AVL drevesi.

In [ ]:

```
(* implementacija avlset *)
module AVLSet(M : ORDERED) : SET with type element = M.t =
  struct
    type element = M.t
    let cmp = M.cmp
```

```
type set = Empty | Node of element * int * set * set

let empty = Empty

let height = function
  | Empty -> 0
  | Node (_, h, _, _) -> h

let leaf v = Node (v, 1, Empty, Empty)

let node (v, l, r) = Node (v, 1 + max (height l) (height r), l, r)

let rec member x = function
  | Empty -> false
  | Node (y, _, l, r) ->
     begin
       match cmp x y with
       | Equal -> true
       | Less -> member x l
       | Greater -> member x r
     end

let rec to_list = function
  | Empty -> []
  | Node (x, _, l, r) -> to_list l @ [x] @ to_list r

let rotateLeft = function
  | Node (x, _, a, Node (y, _, b, c)) -> node (y, node (x, a, b), c)
  | t -> t

(* alternativni zapis s case *)
let rotateRight = function
  | Node (y, _, Node (x, _, a, b), c) -> node (x, a, node (y, b, c))
  | t -> t

let imbalance = function
  | Empty -> 0
  | Node (_, _, l, r) -> height l - height r

let balance = function
  | Empty -> Empty
  | Node (x, _, l, r) as t ->
     begin
       match imbalance t with
       | 2 ->
          (match imbalance l with
           | -1 -> rotateRight (node (x, rotateLeft l, r))
           | _ -> rotateRight t)
       | -2 ->
          (match imbalance r with
           | 1 -> rotateLeft (node (x, l, rotateRight r))
           | _ -> rotateLeft t)
       | _ -> t
     end

let rec add x = function
```

```
    | Empty -> leaf x
    | Node (y, _, l, r) as t ->
       begin
         match cmp x y with
         | Equal -> t
         | Less -> balance (node (y, add x l, r))
         | Greater -> balance (node (y, l, add x r))
       end

  let rec remove x = function
    | Empty -> Empty
    | Node (y, _, l, r) ->
       let rec removeSuccessor = function
         | Empty -> assert false
         | Node (x, _, Empty, r) -> (r, x)
         | Node (x, _, l, r) ->
            let (l', y) = removeSuccessor l in
            (balance (node (x, l', r)), y)
       in
       match cmp x y with
       | Less -> balance (node (y, remove x l, r))
       | Greater -> balance (node (y, l, remove x r))
       | Equal ->
          begin match (l, r) with
          | (_, Empty) -> l
          | (Empty, _) -> r
          | _ ->
             let (r', y') = removeSuccessor r in
             balance (node (y', l, r'))
          end

  let rec fold f x = function
    | Empty -> x
    | Node (y, _, l, r) -> fold f (f (fold f x l) y) r

end
```

Preverimo, da je `AVLSet` res učinkovitejši od `ListSet`. Najprej definiramo dve implementaciji množic celih števil, eno s seznami in eno z AVL drevesi:

In [ ]:

```
module SL = ListSet (struct type t = int let cmp = ocaml_cmp end)
module SA = AVLSet (struct type t = int let cmp = ocaml_cmp end)
```

Naloga 4: Množica `{1, 2, ..., n}`

Definirajte funkcijo `add_list n`, ki vrne množico (implementirano s seznami) števil od `1` do `n`. Primer uporabe:

```
# SL.to_list (add_list 10) ;;
- : SL.element list = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]
```

In [ ]:
```
let rec add_list = function
| 0 -> SL.empty
| n -> SL.add n (add_list (n - 1));;

let add_list(n : int) : SL.set =
match n with
| 0 -> SL.empty
| _ -> SL.add n (add_list (n - 1))

SL.to_list (add_list 10)
```

Na enak način definirajte še funkcijo `add_avl`, ki množico implementira z AVL drevesi:

```
# SA.to_list (add_avl 10) ;;
- : SA.element list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

In [ ]:
```
(* let rec add_avl = … *)
SA.to_list (add_avl 10)
```

Naloga 5: Čas izvajanja

Podana je funkcija `time`, ki izmeri čas izvajanja dane funkcije `f`:

In [ ]:
```
let time f =
  let start = Sys.time () in
  f () ;
  let stop = Sys.time () in
  stop -. start
```

Funkcija `f` je »thunk«, torej sprejme le argument tipa `unit`. Čas, ki ga OCaml potrebuje, da sešteje vsa števila od 1 do 10000000:

```
# time (fun () -> let s = ref 0 in for i = 1 to 10000000 do s
:= !s + i done; !s) ;;
- : float = 0.17635
```

Izmerite, koliko časa traja vstavljanje velikega števila elementov v množico. Primerjajte implementaciji `SL` in `SA`.

In [ ]:
```
time (fun () -> add_list 2000) ;;
time (fun () -> add_avl 2000) ;;
```

Naloga 6: Fold

Funkcija `fold` sprejme

- funkcijo `f` dveh argumentov
- začetno vrednost `z`
- množico `m` = $\{x_1, x_2, \ldots, x_i\}$

ter izračuna rezultat tako, da s pomočjo `f` elemente enega za drugim kombinira s trenutno vrednostjo:

$$f\ (\ldots\ (f\ (f\ z\ x_1)\ x_2)\ \ldots)\ x_{i_n}$$

Primer, kjer je funkcija `f` kar sešetvane, začetna vrednost `z` je `0` in množica `m` je `{1,2,3,4}`:

```
# SL.fold ( + ) 0 (add_list 4) ;;
- : int = 10
```

Signaturi `SET` dodajte funkcijo `fold`:

```
val fold : ('a -> element -> 'a) -> 'a -> set -> 'a
```

in jo implementirajte v funktorjih `ListSet` in `AVLSet`.

In [ ]:

```
(* let fold = List.fold_left *)

(* Sprejme neko funkcijo f, začetno vrednost r in množico s. Množico s lahk

let rec fold f z = function
| [] -> z (* aplicira se 0-krat*)
| x :: xs -> fold f (f z x) xs (* funkcija f, funkcijo f pokličemo nad zače

let rec fold f init s =
  match s with
  | [] -> init
  | x :: xs -> fold f (f init x) xs

let rec fold f x = function
  | Empty -> x
  | Node (y, _, l, r) -> fold f (f (fold f x l) y) r
```

Naloga 7: Unija, filter in presek

Definirajte funktor `SetOps`, ki sprejme strukturo s signaturo `SET` in implementira funkcije `union`, `filter` ter `intersection`.

Unijo in presek poznamo, funkcija `filter` pa sprejme predikat (funkcijo, ki vrača vrednosti tipa `bool`) in množico, ter novo množico tistih elementov, ki zadoščajo predikatu. Pri definiciji funkcij si pomagajte s funkcijo `fold`.

Primer:

```
# module SA_ops = SetOps(SA) ;;
module SA_ops :
  sig
    val union : SA.set -> SA.set -> SA.set
    val filter : (SA.element -> bool) -> SA.set -> SA.set
    val intersection : SA.set -> SA.set -> SA.set
  end
# SA.to_list (SA_ops.filter (fun x -> x > 10) (add_avl 20))
;;
- : SA.element list = [11; 12; 13; 14; 15; 16; 17; 18; 19;
20]
```

In [ ]:

```
(* nepodpisan funktor, ustreza podpisu SET*)
module SetOps(S : SET) =
  struct
  let union a b = S.fold (fun s e -> S.add e s) a b
  let filter p a = S.fold (fun s e -> if p e then S.add e s else s) S.empty
  let intersection a b = filter (fun e -> S.member e b) a
  end

module SA_ops = SetOps(SA)
;;
SA.to_list (SA_ops.filter (fun x -> x > 10) (add_avl 20))
```

Objekti in razredi

Enostavni objekt

In [ ]:

```
let p =
    object

        val mutable x = 10

        val mutable y = 20


        method get_x = x (* rekurzija; x se slicuje na zgornje polje; v Jav


        method get_y = y

        method move dx dy =

            x <- x + dx ;

            y <- y + dy

    end

(* val p : < get_x : int; get_y : int; move : int -> int -> unit > = <obj>
```

Razred ("konstruktor")

Funkcija, ki sprejme začetni koordinati in vrne točko s temi koordinatami. Funkcija, ki producira nove objekte.

Razlikuje se samo po stanju (ne sprogramiramo vsake posebaj), funkcionalnost je enaka.

Brez razredov konstruiramo konstruktor.

In [ ]:
```
class point x0 y0 =
  object

    val mutable x = x0
    val mutable y = y0

    method get_x = x
    method get_y = y
    method move dx dy =
      x <- x + dx ;
      y <- y + dy

  method move' =
    x <- 0 ;
    y <- 0

  end

  (*

  val point :
  int ->
  int ->
  < get_x : int; get_y : int; move : int -> int -> unit; move' : unit > =
  <fun>

  *)

  (* naredimo objekt - novo točko s koordinato (1, 3)*)
  let r = new point 1 3

  (* val r : point = <obj> *)
```

In [ ]:
```
type color = Red | Green | Blue

(* brez podtipov, navadni zapisi *)
type point = { x : float; y : float }
type color_point = { x : float; y : float; c : color }

let p = { x = 3.4; y = 1.7 }
let foo = p.x + p.y;

(** color_point ≤ point ne gre, ker Ocaml nima podtipe zapisov ==> OBJEKTI
type tocka = < get_x : float; get_y : float >
type barvna_tocka = < get_x : float; get_y : float ; get_color : color >

let (a : tocka) =
object
  method get_x = 3.4
  method get_y = 1.7
end
```

In [ ]:
```
(** Primer objektnega programiranja v OCamlu. *)
```

```
(** Antagonistična igra za dva igralca s polno informacijo:

    - antagonistična: igralca sta nasprotnika, ne sodelujeta
    - polna informacija: stanje igre je popolnama znano, v igri ni verjetn

    Dodatno predpostavimo, da ni neodločenega rezultata in da izgubi tisti
    premakniti.
 *)

(** Enemu igralcu pravimo [Player] in drugemu [Oppoennt] *)
type player = Player | Opponent

(** Pretvori igralca v string. *)
let string_of_player = function
  | Player -> "Player"
  | Opponent -> "Opponent"

(** Nasprotni igralec *)
let other = function
  | Player -> Opponent
  | Opponent -> Player

exception Invalid_move

(** Igro predstavimo kot objekt *)
type 'move game =
  <
  to_move : player ;
  (** Kdo je na potezi *)

  moves : 'move list ;
  (** Seznam trenutno veljavnih potez. *)

  string_of_move : 'move -> string ;
  (** Pretvori potezo v string *)

  move_of_string : string -> 'move ;
  (** Pretvori string v potezo *)

  play : 'move -> unit ;
  (** Odigraj dano potezo, sproži [Invalid_move], če poteza ni veljavna. *

  undo : unit ;
  (** Prekliči potezo. *)

  show : unit ;
  (** Prikaži stanje igre na zaslonu. *)
  >

(** Igrica tri v vrsto. *)
module TicTacToe =
  struct

    (** Polje je lahko prazno, ali pa ga zaseda eden od igralcev. *)
    type field = Empty | Full of player
```

```
(** Igrica tri v vrsto, predstavljena kot objekt. *)
let game : (int * int) game =
  object (self)
    val board = Array.make_matrix 3 3 Empty ; (** Igralno polje *)
    val mutable to_move = Player ; (** Kdo je trenutno na potezi *)
    val mutable history = [] ; (** Seznam do sedaj odigranih potez *)

    method to_move = to_move

    method string_of_move (i, j) =
      Format.sprintf "(%d, %d)" i j

    method move_of_string s =
      Scanf.sscanf s "(%d, %d)" (fun i j -> (i, j))

    (** Ali je igra končana? *)
    method private is_over =
      let winning ((i1,j1), (i2,j2), (i3,j3)) =
        board.(i1).(j1) = board.(i2).(j2) &&
          board.(i1).(j1) = board.(i3).(j3) &&
            board.(i1).(j1) <> Empty
      in
      List.for_all
        (fun (i,j) -> board.(i).(j) <> Empty)
        [ (0,0); (1,0); (2,0);
          (0,1); (1,1); (2,1);
          (0,2); (1,2); (2,2) ]
      ||
        List.exists winning
          [ ((0,0), (0,1), (0,2)) ;
            ((1,0), (1,1), (1,2)) ;
            ((2,0), (2,1), (2,2)) ;
            ((0,0), (1,0), (2,0)) ;
            ((0,1), (1,1), (2,1)) ;
            ((0,2), (1,2), (2,2)) ;
            ((0,0), (1,1), (2,2)) ;
            ((0,2), (1,1), (2,0)) ]

    method moves =
      if self#is_over then
        []
      else
        let ms = ref [] in
        for i = 0 to 2 do
          for j = 0 to 2 do
            match board.(i).(j) with
            | Empty -> ms := (i,j) :: !ms
            | Full _ -> ()
          done
        done ;
        assert (!ms <> []) ;
        !ms

    method play (i, j) =
      match board.(i).(j) with
      | Full _ -> raise Invalid_move
      | Empty ->
```

```
            board.(i).(j) <- Full to_move ;
            history <- (i,j) :: history ;
            to_move <- other to_move

        method undo =
          match history with
          | [] -> assert false
          | (i,j) :: hs ->
             history <- hs ;
             to_move <- other to_move ;
             board.(i).(j) <- Empty

        method show =
          Format.printf "+---+---+---+\n" ;
          for i = 0 to 2 do
            Format.printf "|" ;
            for j = 0 to 2 do
              let c =
                match board.(i).(j) with
                | Empty -> ' '
                | Full Player -> 'P'
                | Full Opponent -> 'O'
              in
              Format.printf " %c |" c
            done ;
            Format.printf "\n+---+---+---+\n"
          done ;
          Format.printf "@."
      end
  end

(** Agent, ki zna igrati igrico, je objekt z metodo [move]. *)

(** Človeški agent igra tako, da vprašamo, katero potezo naj odigra. *)
let human (game : 'move game) =
  object

    method move =
      Format.printf "Available moves:@\n" ;
      List.iter (fun m -> Format.printf "%s " (game#string_of_move m)) gam
      Format.printf "@\nEnter your move:@." ;
      game#move_of_string (read_line ())

  end

(** Računalniški agent pregelda celotno igro in naključno izbere eno
    od zmagovalnih potez. Če take poteze ni, izbere naključno potezo. *)
let computer (game : 'move game) =
  object (self)

    (** Izvedi potezo [m], izračunaj [f ()], nato pa prekliči potezo in vr
    method private after_move m f =
      game#play m ;
      let r = f () in
      game#undo ;
      r
```

```
    (** Ali je poteza [m] zmagovalna? *)
    method private is_winning m =
      self#after_move m
        (fun () -> List.for_all self#is_losing game#moves)

    (** Ali poteza [m] izgubi proti optimalnemu nasprotniku? *)
    method private is_losing m =
      self#after_move m
        (fun () -> List.exists self#is_winning game#moves)

    (** Seznam zmagovalnih potez *)
    method private winning_moves =
      List.filter self#is_winning game#moves

    method move =
      let ms =
        match self#winning_moves with
        | [] ->
           begin match game#moves with
           | [] -> failwith "No moves"
           | ms -> ms
           end
        | _::_ as ms -> ms
      in
      let k = Random.int (List.length ms) in
      List.nth ms k

  end

(** Odigraj dano igro z danima agentoma *)
let play game player opponent =
  let ask_move p =
    match game#moves with
    | [] -> None
    | _::_ -> Some (p#move)
  in
  let pl = player game
  and op = opponent game in
  let rec loop () =
    game#show ;
    let m =
      match game#to_move with
      | Player -> ask_move pl
      | Opponent -> ask_move op
    in
    match m with
    | None ->
       Format.printf "%s won!@." (string_of_player (other game#to_move))
    | Some m ->
       Format.printf "%s plays %s@." (string_of_player game#to_move) (game
       game#play m ;
       loop ()
  in
  loop ()

;;
```

```
(** Glavni program *)

let main =
  Random.self_init () ;
  play TicTacToe.game human computer
```

OCaml (naloge in izpiti)

https://ocaml.org/docs

Izpiti

Naloga 1 (specifikacija, implementacija)

Implement any module named Cow which satisfies the signature

In [ ]:
```
module type BOVINE =
sig
  type t
  val cow : t
  val equal : t -> t -> bool
  val to_string : t -> string
end
```

In [ ]:
```
(* Rešitev *)
module Cow : BOVINE =
struct
  type t = Milka
  let cow = Milka
  let equal _ _ = true
  let to_string _ = "Milka"
end
```

Naloga 2 (specifikacija, implementacija)

V jeziku OCaml želimo definirati podatkovno strukturo sklad.

Definirajte signaturo `STACK`, ki obsega tipe elementov, sklada in naslednjih vrednosti:

- prazen sklad `empty`
- funkcijo `push`, ki sprejme element in sklad ter vrne sklad z dodanim elementom
- funkcijo `pop`, ki sprejme sklad in vrne par (nazadnje dodani element, preostanek sklada)

Definirajte funktor `ListStack`, ki sprejme `t` in vrne implementacijo sklada, ki elemente tipa `t` hrani v seznamu.

In [ ]:
```
(* Rešitev *)

module type STACK =
sig
  type element
  type stack
  val empty : stack
  val push : element -> stack -> stack
  val pop : stack -> element * stack
end

module ListStack(type t) : STACK =
struct
  type element = t
  type stack = element list
  let empty = []
  let push x s = (x :: s)
  let pop (x :: s) = (x, s)
end

module ListStack(M : sig type t end) : STACK =
struct
    type element = M.t
    type stack = element list
    let empty = []
    let push x s = x::s
    let pop = function
        [] -> None, []
        | x::s -> Some x, s
end
```

Naloga 3 (fold)

V OCamlu sestavimo funkcijo fold in g.

In [ ]:
```
let rec fold f acc = function
| [] -> acc
| x :: xs -> fold f (f acc x) xs
let g = fold (fun x ys -> x * (fold ( + ) 0 ys)) 0
```

Kaj izračuna g?

Rešitev: vstavi v ocaml interpreter -->

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

val g : int list list -> int =

g [[0; 1]] ... 0

Odgovor je, da vedno vrne 0.

Naloga 4 (program)

Implement the program below

In [ ]:
```
{b > 1}
i := 2 ;
j := 0;
while j < b do
i := i + i + i - 2;
j := j + 1;
end
{i = 3 ^ b + 1}
```

as a function `power3plus1: int -> int`

which takes a non-negative integer `b` and returns the values specified in program. The function should not use while or for loop.

In [ ]:
```
let power3plus1 b =
  let rec loop i j =
    if j < b then loop (i + i + i - 2) (j + 1)
    else i
  in loop 2 0
```

Naloga 5 (drevo)

V OCamlu definiramo podatkovni tip dreves, v katerih so vozlišča označena s celimi števili:

In [ ]:
```
type tree = Empty | Node of int * tree * tree
```

Sestaviti želimo funkcijo `sum : tree -> int` , ki sešteje cela števila v vozliščih drevesa:

In [ ]:
```
# sum Empty ;;
- : int = 0
# sum (Node (20, Node (3, Empty, Empty), Node (19, Empty, Empty))) ;;
- : int = 42
```

In [ ]:
```
let rec sum = function
| Empty -> 0
| Node(x, l, r) -> x + sum l + sum r

let rec sum t =
  match t with
  | Empty -> 0
  | Node(x, l, r) -> x + sum l + sum r
```

Naloga 6 (tipi)

V OCamlu definiramo tip:

In [ ]:
```
type oseba = {ime : string; priimek : string; rojstvo : int }
```

Med spodnjimi izrazi označi tiste, ki imajo tip `oseba`.

In [ ]:
```
let oseba = {ime = "Kekec"; priimek = None; rojstvo = 1918};; (* nepravilno
let oseba2 = {ime = "Kekec"; priimek = "PP"};; (* nepravilno, ker Error: So
let oseba3 = {ime = "Kekec"; priimek =""; rojstvo = (let s = 1000 in s + 91
let oseba4 = {priimek = "PP"; ime = "Mojca"; rojstvo = 1/0};; (* Exception
```

Naloga 7 (naloga)

V OCamlu definiramo podatkovni tip `number`, s kateri predstavimo cela števila: `type number = Zero | Succ of integer | Pred of integer`.

Vrednost `Zero` predstavlja število `0`, `Succ n` naslednika n ter `Pred n` predhodnika n.

Vsako število predstavimo na več načinov.

Na primer, število 0 je predstavljeno z vrednostmi:

In [ ]:
```
Zero
Pred (Succ Zero)
Succ (Pred Zero)
Pred (Pred (Succ(Succ Zero)))
Pred (Succ (Succ (Pred Zero)))
```

Med vsemi je najbolj ekonomična predstavitev `Zero`, ker ne vsebuje nepotrebnih konstruktorjev.

Sestavite funkcijo `simp : number -> number`, ki dano predstavitev pretvori v najbolj ekonomično, se pravi tako, ki ima najmanjše možno število konstruktorjev.

In [ ]:
```
# simp (Pred (Succ (Succ (Pred (Pred (Succ (Pred Zero)))))));;
- : number = Pred Zero
# simp (Succ Zero);;
- : number = Succ Zero
```

In [ ]:
```
let rec simp = function
  | Zero -> Zero
  | Pred a ->
      (match simp a with
       | Succ a -> a
       | a -> Pred a
      )
  | Succ a ->
      (match simp a with
       | Pred a -> a
       | a -> Succ a
      );;
```

Naloga 8 (tipi)

Definiramo podatkovni tip `type 'a trie = Node of 'a | Trie of ('a trie) list` .

- zapiši vrednost tipa `int trie`
- zapiši vrednost tipa `'a trie`
- zapiši vrednost, ki je različna od prejšnjih dveh

In [ ]:
```
int trie: Node 1
'a trie: Trie []
Node 5
```

Naloge

Funkcija `null` , ki preveri, ali je seznam prazen.

In [ ]:
```
let null l =
    if l = [] then true
    else false

(*

null [] ;;
- : bool = true

null [1; 2] ;;
- : bool = false

*)
```

Funkcija `size`, ki izračuna dolžino seznama.

In [ ]:
```
let rec size l =
    if l = [] then 0
    else 1 + (size (List.tl l))

(* val size : 'a list -> int = <fun> *)

let rec size l = match l with
    | [] -> 0
    | _ :: tail -> 1 + (size tail)
```

Funkcija `reverse`, ki obrne seznam.

In [ ]:
```
let rec reverse l =
    if l = [] then []
    else (reverse (List.tl l)) @ [(List.hd l)]

(* val reverse : 'a list -> 'a list = <fun> *)

let rec reverse l = match l with
    | [] -> []
    | head::tail -> (reverse tail) @ [head]
```

Funkcija `flip`, ki obrne vsebino parov, ki so elementi seznama.

In [ ]:
```
let rec flip l =
    if l = [] then []
    else
        let head = List.hd l in
            let rest = List.tl l in
                (snd head, fst head) :: (flip rest)

(* val flip : ('a * 'b) list -> ('b * 'a) list = <fun> *)

let rec flip l = match l with
    | [] -> []
    | (a, b) :: tail -> (b, a) :: (flip tail)
```

Funkcija `take`, ki vzame prvih n elementov.

In [ ]:
```
let rec take(n, l) =
    if l = [] then []
    else if n > 0 then List.hd(l) :: take(n - 1, List.tl(l))
    else []

(* val take : int * 'a list -> 'a list = <fun> *)
```

Funkcija `drop`, ki odreže prvih n elementov.

In [ ]:
```
let rec drop(n, l) =
    if l = [] then []
    else if n > 0 then drop(n - 1, List.tl l)
    else List.hd(l) :: List.tl(l)

(* val drop : int * 'a list -> 'a list = <fun> *)
```

Funkcija `combine`, ki sestavi seznam parov.

In [ ]:
```
let rec combine(l1, l2) =
    if l1 = [] && l2 = [] then []
    else (List.hd(l1), List.hd(l2)) :: combine(List.tl(l1), List.tl(l2))

(* val combine : 'a list * 'b list -> ('a * 'b) list = <fun> *)
```

Funkcija `split`, ki razcepi seznam parov.

In [ ]:
```
let rec split l =
    if l = [] then ([], [])
    else
        let l1 = split(List.tl(l)) in
            (fst(List.hd l) :: fst(l1), snd(List.hd(l)) :: snd(l1))

(* val split : ('a * 'b) list -> 'a list * 'b list = <fun> *)
```

Funkcija `member`, ki preveri članstvo v seznamu oz. množici.

In [ ]:
```
let rec member x l =
    if l = [] then false
    else if x = List.hd(l) then true
    else member x (List.tl l)

(* val member : 'a -> 'a list -> bool = <fun> *)
```

Funkcija `inter`, ki realizira razliko nad seznami.

In [ ]:
```
let rec inter(l1, l2) =
    if l1 = [] then []
    else
        let x = List.hd l1 in
            if (member x l2) then x :: inter(List.tl l1, l2)
            else inter(List.tl l1, l2)

(* val inter : 'a list * 'a list -> 'a list = <fun> *)
```

Funkcija `union`, ki realizira unijo nad seznami.

In [ ]:
```
let rec union(xs, ys) =
    if xs = [] then ys
    else let x = List.hd xs in
        let xr = union (List.tl xs, ys) in
            if (member x ys) then xr
            else
                x::xr

(* val union : 'a list * 'a list -> 'a list = <fun>  *)
```

Funkcija `imply`, ki realizira vse možne primere.

In [ ]:
```
let imply v = match v with
    | (true, x) -> x
    | _ -> true
    ;;

(* val imply : bool * bool -> bool = <fun> *)
```

Binarna drevesa

Definiraj definicijo binarnega iskalnega drevesa.

In [ ]:
```
type binaryTree =
    | Empty
    | Node of binaryTree * int * binaryTree
```

Funkcija `list_of_tree` , ki izlušči sortiran seznam iz drevesa s pregledovanjem drevesa po principu v globino.

In [ ]:
```
let rec list_of_tree l = match l with
    | Empty -> []
    | Node (l, v, r) -> (list_of_tree l) @ (r :: (list_of_tree r))

(* val list_of_tree : binaryTree -> binaryTree list = <fun> *)
```

Mešano

Write a function, `seconds_since_midnight h m s` , which returns the number of seconds elapsed since midnight.

Your function should have the following type: `val seconds_since_midnight : int -> int -> int -> int`

In [ ]:
```
let seconds_since_midnight h m s = s + m * 60 + h * 3600
```

Write a function, `fibonacci n` , which compute the nth Fibonacci number.

The function should have the following type: `val fibonacci : int -> int`

In [ ]:
```
let rec fibonacci n =
  if n = 0 then 0
  else if n = 1 then 1
  else fibonacci(n - 1) + fibonacci(n - 2)
```

Define a type, time, which holds the hour, minute, and second as separate values.

In [ ]:
```
type time = Time of int * int * int
```

Write a function `seconds_since_midnight2` with the following type: `val seconds_since_midnight2 : time -> int` .

In [ ]:
```
let seconds_since_midnight2 t =
  match t with
    | Time(h, m, s) -> s + 60 * m + 3600 * h
```

Write a function `time_diff t1 t2` , which calculates the number of seconds that have elapsed between `t1` and `t2` : `val time_diff : time -> time -> int`

In [ ]:
```
let time_diff t1 t2 =
  seconds_since_midnight2 t1 - seconds_since_midnight2 t2;;
```

Write a function `all_positive lst`, which returns true if all the integers in `lst` are positive.

`val all_positive : intlist -> bool`

In [ ]:
```
let rec all_positive lst =
  match lst with
  | [] -> true
  | x :: xs -> x > 0 && all_positive xs
```

Write a function `all_even lst`, which returns true if all the integers in `lst` are even numbers.

`val all_even : intlist -> bool`

In [ ]:
```
let rec all_even lst =
  match lst with
  | [] -> true
  | x :: xs -> x mod 2 = 0 && all_even xs
```

Write the function `is_sorted lst` to determine if the integers in `lst` are in sorted (ascending).

`val is_sorted : intlist -> bool`

In [ ]:
```
let rec is_sorted_helper prev lst =
  match lst with
  | [] -> true
  | x :: xs -> prev <= x && is_sorted_helper x rest

  let is_sorted lst =
    match lst with
    | [] -> true
    | x :: xs -> is_sorted x xs
```

Tail of a list

Write a function `last : 'a list -> 'a option` that returns the last element of a list.

In [ ]:
```
# last ["a" ; "b" ; "c" ; "d"];;
- : string option = Some "d"
# last [];;
- : 'a option = None
```

In [ ]:
```
let rec last lst =
  match lst with
  | [] -> None
  | [x] -> Some x
  | x :: xs -> last xs;;
```

Last two elements of a list

Find the last but one (last and penultimate) elements of a list.

In [ ]:
```
# last_two ["a"; "b"; "c"; "d"];;
- : (string * string) option = Some ("c", "d")
# last_two ["a"];;
- : (string * string) option = None

val last_two : 'a list -> ('a * 'a) option
```

In [ ]:
```
let rec last_two lst =
  match lst with
  | [] | [_] -> None
  | [x ; y] -> Some(x, y)
  | x :: xs -> last_two xs
```

N'th element of a list.

Find the K'th element of a list.

In [ ]:
```
# List.nth ["a"; "b"; "c"; "d"; "e"] 2;;
- : string = "c"
# List.nth ["a"] 2;;
Exception: Failure "nth".
```

In [ ]:
```
let rec at k = function
  | [] -> None
  | x :: xs -> if k = 1 then Some x else at (k - 1) xs;;
```

Length of a list

Find the number of elements of a list.

In [ ]:
```
# length ["a"; "b"; "c"];;
- : int = 3
# length [];;
- : int = 0

val length : 'a list -> int = <fun>
```

In [ ]:
```
let length lst =
  let rec aux n = function
  | [] -> n
  | x :: xs -> aux (n + 1) xs
  in
    aux 0 lst
```

Reverse a list

Reverse a list.

In [ ]:
```
# rev ["a"; "b"; "c"];;
- : string list = ["c"; "b"; "a"]

val reverse : 'a list -> 'a list = <fun>
```

In [ ]:
```
let reverse lst =
  let rec aux acc = function
  | [] -> acc
  | x :: xs -> aux (x :: acc) xs
in
aux [] lst
```

Palindrome

Find out whether a list is a palindrome.

In [ ]:
```
# is_palindrome ["x"; "a"; "m"; "a"; "x"];;
- : bool = true
# not (is_palindrome ["a"; "b"]);;
- : bool = true

val is_palindrome : 'a list -> bool = <fun>
```

In [ ]:
```
let is_palindrome list =
  list = List.rev list
```

Flatten a list

Flatten a nested list structure.

In [ ]:
```
# flatten [One "a"; Many [One "b"; Many [One "c" ;One "d"]; One "e"]];;
- : string list = ["a"; "b"; "c"; "d"; "e"]
```

In [ ]:
```
type 'a node =
  | One of 'a
  | Many of 'a node list
```

In [ ]:
```
(* This function traverses the list, prepending any encountered elements
    to an accumulator, which flattens the list in inverse order. It can
    then be reversed to obtain the actual flattened list. *)

  let flatten lst =
    let rec aux acc = function
    | [] -> acc
    | One x :: t -> aux (x :: acc) t
    | Many l :: t -> aux (aux acc l) t
  in
  List.rev (aux [] lst)
```

Eliminate duplicates

Eliminate consecutive duplicates of list elements.

In [ ]:
```
# compress ["a"; "a"; "a"; "a"; "b"; "c"; "c"; "a"; "a"; "d"; "e"; "e"; "e
- : string list = ["a"; "b"; "c"; "a"; "d"; "e"]

val compress : 'a list -> 'a list = <fun>
```

In [ ]:
```
let rec compress = function
| a :: (b :: _ as t) -> if a = b then compress t else a :: compress t
| smaller -> smaller
```

Pack consecutive duplicates

Pack consecutive duplicates of list elements into sublists.

In [ ]:
```
# pack ["a"; "a"; "a"; "a"; "b"; "c"; "c"; "a"; "a"; "d"; "d"; "e"; "e"; "
- : string list list =
[["a"; "a"; "a"; "a"]; ["b"]; ["c"; "c"]; ["a"; "a"]; ["d"; "d"];
 ["e"; "e"; "e"; "e"]]

val pack : 'a list -> 'a list list = <fun>
```

In [ ]:
```
let pack list =
  let rec aux current acc = function
    | [] -> []    (* Can only be reached if original list is empty *)
    | [x] -> (x :: current) :: acc
    | a :: (b :: _ as t) ->
       if a = b then aux (a :: current) acc t
       else aux [] ((a :: current) :: acc) t  in
  List.rev (aux [] [] list);;
```

Run-length encoding

In [ ]:
```
encode ["a"; "a"; "a"; "a"; "b"; "c"; "c"; "a"; "a"; "d"; "e"; "e"; "e"; "
- : (int * string) list =
[(4, "a"); (1, "b"); (2, "c"); (2, "a"); (1, "d"); (4, "e")]

val encode : 'a list -> (int * 'a) list = <fun>
```

In [ ]:
```
let encode list =
  let rec aux count acc = function
    | [] -> [] (* Can only be reached if original list is empty *)
    | [x] -> (count + 1, x) :: acc
    | a :: (b :: _ as t) -> if a = b then aux (count + 1) acc t
                            else aux 0 ((count + 1, a) :: acc) t in
  List.rev (aux 0 [] list);;
```

Duplicate the elements of a list

In [ ]:
```
# duplicate ["a"; "b"; "c"; "c"; "d"];;
- : string list = ["a"; "a"; "b"; "b"; "c"; "c"; "c"; "c"; "d"; "d"]

val duplicate : 'a list -> 'a list = <fun>
```

In [ ]:
```
let rec duplicate = function
| [] -> []
| x :: xs -> x :: x :: duplicate xs
```

Drop every N'th element from a list

In [ ]:
```
# drop ["a"; "b"; "c"; "d"; "e"; "f"; "g"; "h"; "i"; "j"] 3;;
- : string list = ["a"; "b"; "d"; "e"; "g"; "h"; "j"]
```

In [ ]:
```
let drop list n =
  let rec aux i = function
    | [] -> []
    | h :: t -> if i = n then aux 1 t else h :: aux (i + 1) t  in
  aux 1 list;;
```

Extract a slice from a list

Given two indices, i and k, the slice is the list containing the elements between the i'th and k'th element of the original list (both limits included). Start counting the elements with 0 (this is the way the List module numbers elements).

In [ ]:
```
# slice ["a"; "b"; "c"; "d"; "e"; "f"; "g"; "h"; "i"; "j"] 2 6;;
- : string list = ["c"; "d"; "e"; "f"; "g"]
```

In [ ]:
```
let slice list i k =
  let rec take n = function
    | [] -> []
    | h :: t -> if n = 0 then [] else h :: take (n - 1) t
  in
  let rec drop n = function
    | [] -> []
    | h :: t as l -> if n = 0 then l else drop (n - 1) t
  in
  take (k - i + 1) (drop i list);;
```

Remove the K'th element from a list

In [ ]:
```
# remove_at 1 ["a"; "b"; "c"; "d"];;
- : string list = ["a"; "c"; "d"]

val remove_at : int -> 'a list -> 'a list = <fun>
```

In [ ]:
```
let rec remove_at n = function
| [] -> []
| x :: xs -> if n = 0 then xs else x :: remove_at (n - 1) xs
```

Insert an element at a given position into a list

Start counting list elements with 0. If the position is larger or equal to the length of the list, insert the element at the end. (The behavior is unspecified if the position is negative.)

In [ ]:
```
# insert_at "alfa" 1 ["a"; "b"; "c"; "d"];;
- : string list = ["a"; "alfa"; "b"; "c"; "d"]

val insert_at : 'a -> int -> 'a list -> 'a list = <fun>
```

In [ ]:
```
let rec insert_at x n = function
| [] -> [x]
| x :: xs as l -> if n = 0 then x :: l else x :: insert_at x (n - 1) xs
```

Create a list containing all integers within a given range

If first argument is greater than second, produce a list in decreasing order.

In [ ]:
```
# range 4 9;;
- : int list = [4; 5; 6; 7; 8; 9]

val range : int -> int -> int list = <fun>
```

In [ ]:
```
let range a b =
  let rec aux a b =
    if a > b then [] else a :: aux (a + 1) b
  in
    if a > b then List.rev (aux b a) else aux a b;;
```

Determine the greatest common divisor of two positive integer numbers

In [ ]:
```
# gcd 13 27;;
- : int = 1
# gcd 20536 7826;;
- : int = 2

val gcd : int -> int -> int = <fun>
```

In [ ]:
```
let rec gcd a b =
  if b = 0 then a
  else gcd b (a mod b)
```

A list of prime numbers

Given a range of integers by its lower and upper limit, construct a list of all prime numbers in that range.

In [ ]:
```
# List.length (all_primes 2 7920);;
- : int = 1000

val is_prime : int -> bool = <fun>
val all_primes : int -> int -> int list = <fun>
```

In [ ]:
```
let is_prime n =
  let n = max n (-n) in
  let rec is_not_divisor d =
    d * d > n || (n mod d <> 0 && is_not_divisor (d + 1))
  in
    is_not_divisor 2

let rec all_primes a b =
  if a > b then [] else
    let rest = all_primes (a + 1) b in
    if is_prime a then a :: rest else rest;;
```

Collect the leaves of a binary tree in a list

A leaf is a node with no successors. Write a function leaves to collect them in a list.

In [ ]:
```
# leaves Empty;;
- : 'a list = []
```

In [ ]:
```
let leaves t =
  let rec leaves_aux t acc =
    match t with
    | Empty -> acc
    | Node (x, Empty, Empty) -> x :: acc
    | Node (x, l, r) -> leaves_aux l (leaves_aux r acc)
  in
  leaves_aux t []
```

Count the leaves of a binary tree

A leaf is a node with no successors. Write a function count_leaves to count them.

In [ ]:
```
# count_leaves Empty;;
- : int = 0

val count_leaves : 'a binary_tree -> int = <fun>
```

In [ ]:
```
let rec count_leaves = function
| Empty -> 0
| Node (_, Empty, Empty) -> 0
| Node (_, l, r) -> count_leaves l + count_leaves r
```

Collect the internal nodes of a binary tree in a list

An internal node of a binary tree has either one or two non-empty successors. Write a function internals to collect them in a list.

In [ ]:
```
# internals (Node ('a', Empty, Empty));;
- : char list = []
```

In [ ]:
```
let internals t =
  let rec internals_aux t acc = match t with
    | Empty -> acc
    | Node (x, Empty, Empty) -> acc
    | Node (x, l, r) -> internals_aux l (x :: internals_aux r acc)
  in
  internals_aux t [];;
```

Count the nodes of a multiway tree

count_nodes (T ('a', [T ('f', [])]));;

- : int = 2

val count_nodes : 'a mult_tree -> int =

In [ ]:
```
let rec count_nodes (T(_, sub)) =
  List.fold_left (fun n t -> n + count_nodes t) 1 sub
```

KOINDUKTIVNI TIPI IN IZPELJAVA TIPOV

Teorija

Glavni tip izraza e izpeljemo v dveh **fazah**:

1. **Izračunamo kandidata za tip** e, ki vsebuje neznanke in enačbe, ki jih morajo neznanke zadoščati.

- `true` ima tip `bool`, brez enačb
- `false` ima tip `bool`, brez enačb
- celoštevilska konstanta `0`, `1`, `2`, ... ima tip `int`, brez enačb
- spremenljivka ima svoj dani tip (tipe spremenljivk sproti beležimo v *kontekstu*)
- aritmetični izraz e_1 + e_2:
  - izračunamo tip τ_1 izraza e_1 in dobimo še enačbe E_1
  - izračunamo tip τ_2 izraza e_2 in dobimo še enačbe E_2

  Tip izraza e_1 + e_2 je `int`, z enačbami E_1, E_2 in τ_1 = `int`, τ_2 = `int` Podobno obravnavamo ostale aritmetične izraze e_1 * e_2, e_1 - e_2, ...
- boolov izraz e_1 && e_2: obravnavamo podobno kot aritmetični izraz, le da uporabimo pričakovani `bool` namesto `int`.
- primerjava celih števil e_1 < e_2:
  - izračunamo tip τ_1 izraza e_1 in dobimo še enačbe E_1
  - izračunamo tip τ_2 izraza e_2 in dobimo še enačbe E_2

  Tip izraza e_1 < e_2 je `bool`, z enačbami E_1, E_2 in τ_1 = `int`, τ_2 = `int`
- pogojni stavek `if` e_1 `then` e_2 `else` e_3:
  - izračunamo tip τ_1 izraza e_1 in dobimo še enačbe E_1
  - izračunamo tip τ_2 izraza e_2 in dobimo še enačbe E_2
  - izračunamo tip τ_3 izraza e_3 in dobimo še enačbe E_3

  Tip izraza `if` e_1 `then` e_2 `else` e_3 je τ_2, z enačbami E_1, E_2, E_3, τ_1 = `bool`, τ_2 = τ_3
- urejeni par (e_1, e_2):
  - izračunamo tip τ_1 izraza e_1 in dobimo še enačbe E_1
  - izračunamo tip τ_2 izraza e_2 in dobimo še enačbe E_2

  Tipi izraza (e_1, e_2) je τ_1 × τ_2, z enačbami E_1, E_2.
- prva projekcija `fst` e:
  - izračunamo tip τ izraza e in dobimo še enačbe E

  Uvedemo nova parametra α in β (se ne pojavljata v E). Tip izraza `fst` e je α, z enačbami E, τ = α × β.
- druga projekcija `snd` e:
  - izračunamo tip τ izraza e in dobimo še enačbe E

  Uvedemo nova parametra α in β. Tip izraza `snd` e je β, z enačbami E, τ = α × β.
- funkcija `fun x ->` e: uvedemo nov parameter α in zabeležimo, da ima x tip α, ter
  - izračunamo tip τ izraza e (pri predpostavki, da ima x tip α) in dobimo še enačbe E

  Tip funkcije `fun x ->` e je α → τ z enačbami E
- aplikacija e_1 e_2:
  - izračunamo tip τ_1 izraza e_1 in dobimo še enačbe E_1
  - izračunamo tip τ_2 izraza e_2 in dobimo še enačbe E_2

  Uvedemo nov parameter α. Tip izraza e_1 e_2 je α, z enačbami E_1, E_2, τ_1 = τ_2 → α
- rekurzivna definicija x = e (kjer se x pojavi v e): uvedemo nov parameter α, zabeležimo, da ima x tip α, ter
  - izračunamo tip τ izraza e (pri predpostavki, da ima x tip α) in dobimo še enačbe E

  Tip izraza x je τ, z enačbami E, α = τ. Opomba: običajno na ta način definiramo rekurzivne funkcije, torej bo x v resnici funkcija.

- prazen seznam `[]` ima tip α `list`, kjer je α nov parameter
- sestavljen seznam e_1 `::` e_2:
  - izpeljemo tip τ_1 izraza e_1 in dobimo enačbe E_1
  - izpeljemo tip τ_2 izraza e_2 in dobimo enačbe E_2

  Tip e_1 `::` e_2 je τ_1 `list`, z enačbami E_1, E_2 in $\tau_2 = \tau_1$ `list`.
- izraz `match` e_1 `with [] ->` e_2 `| x :: xs ->` e_3:
  - izpeljemo tip τ_1 izraza e_1 in dobimo enačbe E_1
  - izpeljemo tip τ_2 izraza e_2 in dobimo enačbe E_2
  - uvedemo nov parameter α, zabeležimo `x :` α in `xs :` α `list`, izpeljemo tip τ_2 izraza e_2 in dobimo enačbe E_3

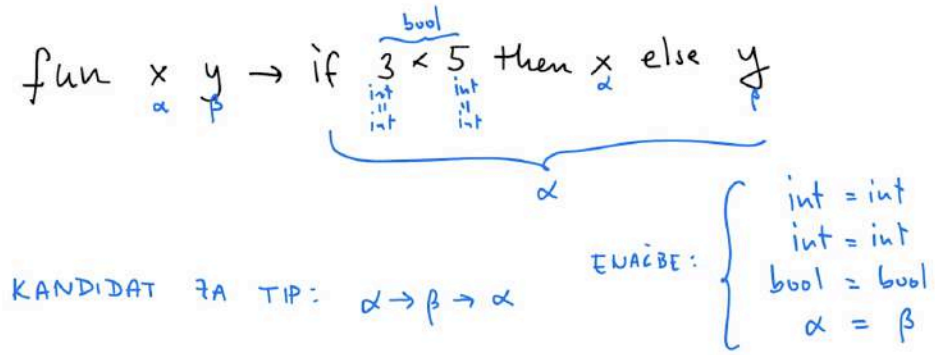
  Tip `match` e_1 `with [] ->` e_2 `| x :: xs ->` e_3 je τ_2 z enačbami E_1, E_2, E_3, $\tau_1 = \alpha$ `list` in $\tau_2 = \tau_3$.

2. Rešimo enačbe s postopkom **združevanja** (izkaže se lahko, da enačbe nimajo rešitve).

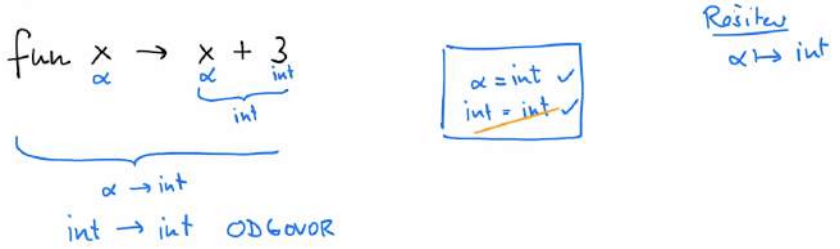
   1. Imamo seznam rešitev `r`, ki je na začetku prazen.
   2. Če je `E` prazna množica, vrnemo rešitev `r`.
   3. Sicer iz `E` odstranimo katerokoli enačbo `l = d` in jo obravnvamo:
      - če sta leva in desna stran povsem enaki, enačbo zavržemo ter gremo na korak 2
      - če je enačba oblike α `= d`, kjer je α neznanka:
        - če se α pojavi v `d`, postopek prekinemo, ker *ni rešitve*
        - sicer smo našli rešitev za α, namreč $\alpha \mapsto$ `d`. Povsod v `r` in `E` zamenjamo α z `d` in v `r` dodamo rešitev $\alpha \mapsto$ `d`
      - če je enačba oblike `l =` α, kjer je α neznanka, imamo primer, ki je simetričen prejšnjemu
      - če je enačba oblike $(l_1 \to l_2) = (d_1 \to d_2)$, v `E` dodamo enačbi $l_1 = d_1$ in $l_2 = d_2$ in gremo na korak 2
      - če je enačba oblike $(l_1 \times l_2) = (d_1 \times d_2)$, v `E` dodamo enačbi $l_1 = d_1$ in $l_2 = d_2$ in gremo na korak 2
      - če je enačba katerekoli druge oblike, na primer $(l_1 \to l_2) = (d_1 \times d_2)$, postopek prekinemo, ker *ni rešitve*.

Naloge

`fun x y -> if 3 < 5 then x else y`



`fun x -> x + 3`



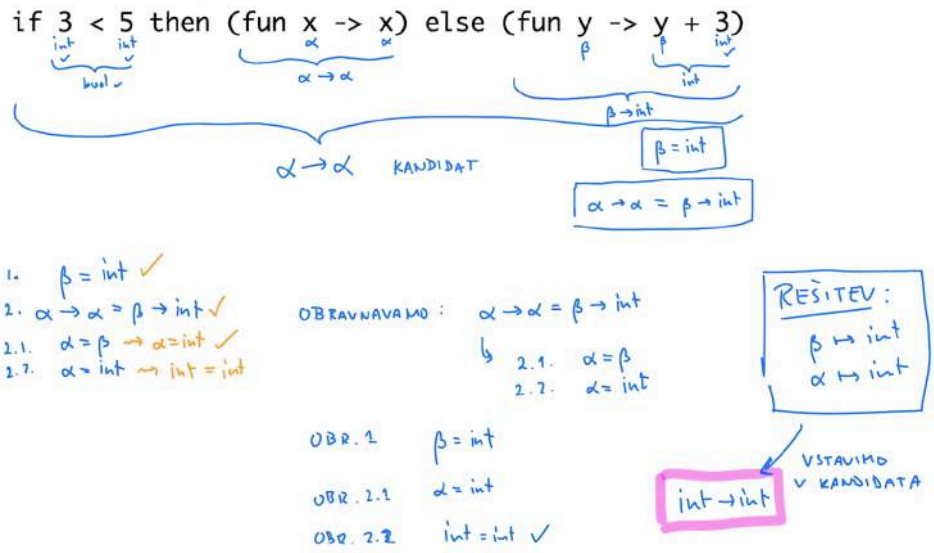
Rešitev: `int -> int`

`fun x -> x`

Rešitev: `'a -> 'a`

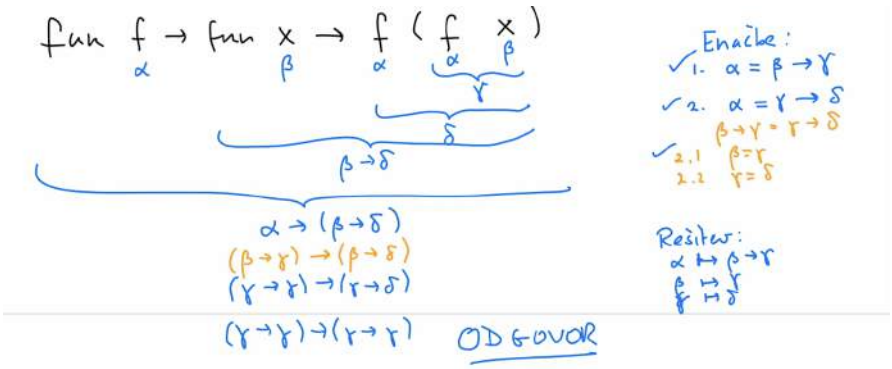
`if 3 < 5 then (fun x -> x) else (fun y -> y + 3)`

Imamo pogojni izraz: then in else morata biti istega tipa



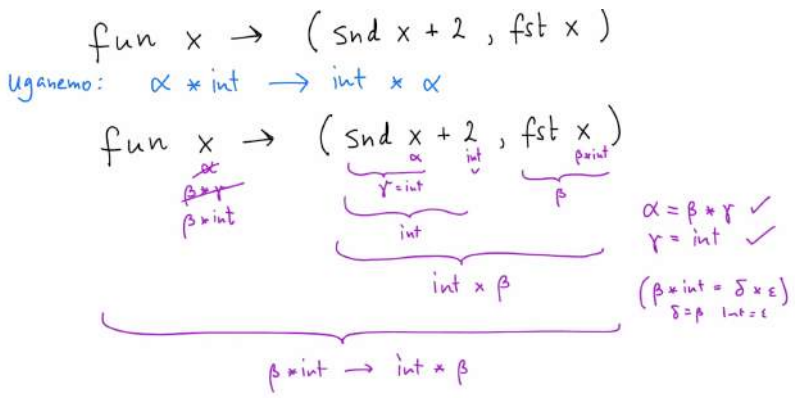
Rešitev: `int -> int`

`λ f x. f ( f x )`
`fun f -> fun x -> f ( f x )`



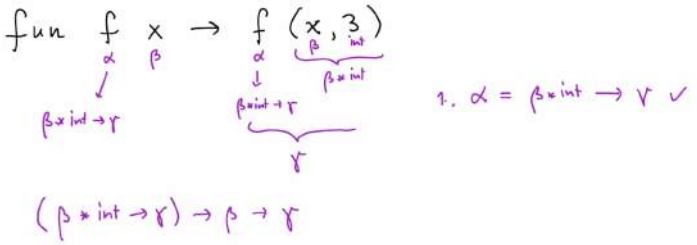
Rešitev: `- : ('a -> 'a) -> 'a -> 'a = <fun>`

fun x -> (snd x + 2, fst x)



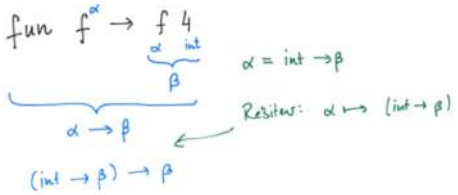
Rešitev: `'a * int -> int * 'a = <fun>`

fun f x -> f (x 3)



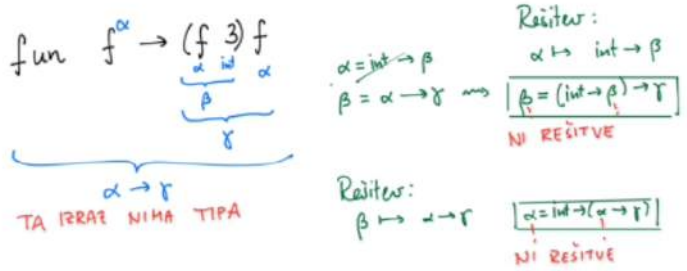
Rešitev: `('a -> 'b) -> (int -> 'a) -> 'b = <fun>`

fun f -> f 4



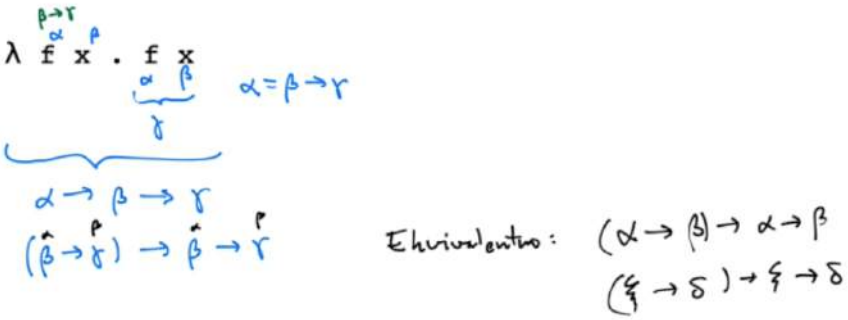
Rešitev: `(int -> 'a) -> 'a = <fun>`

```
fun f -> (f 3) f
```



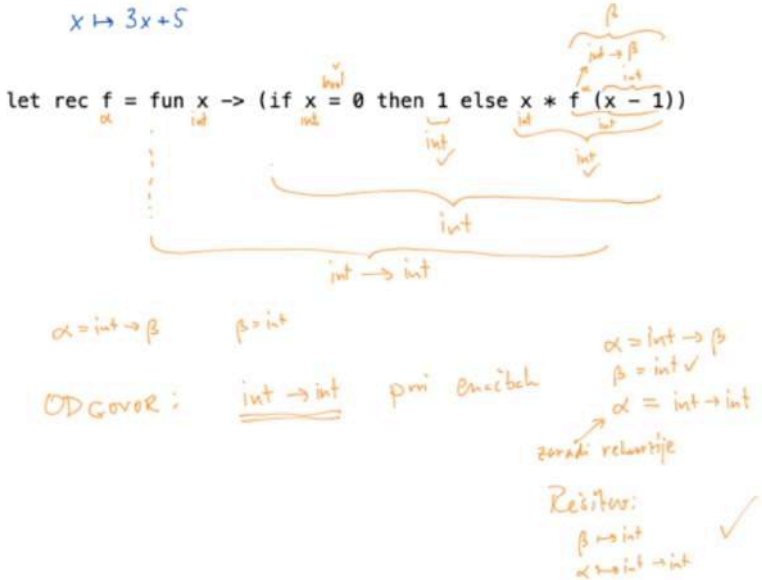
Rešitev: ni rešitve

```
λ f x . f x
```



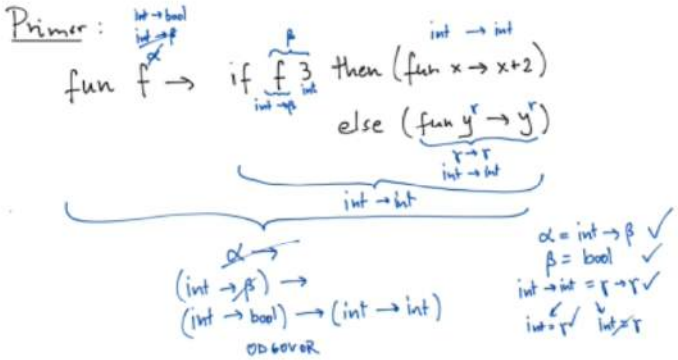
Rešitev: `('a -> 'b) -> 'a -> 'b = <fun>`

```
let rec f = fun n ->
      if n = 0 then 1
       else
        n * f (n - 1)
```



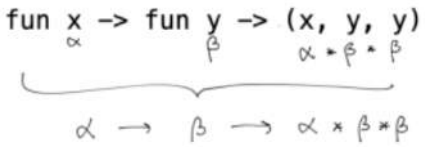
Rešitev: `int -> int`

```
fun f -> if f 3 then (fun x -> x + 2)
                else (fun y -> y)
```



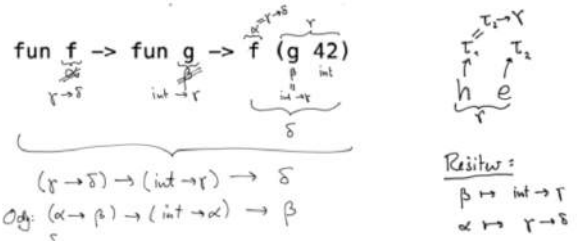
Rešitev: `(int -> bool) -> int -> int = <fun>`

```
fun x -> fun y -> (x, y, y)
```



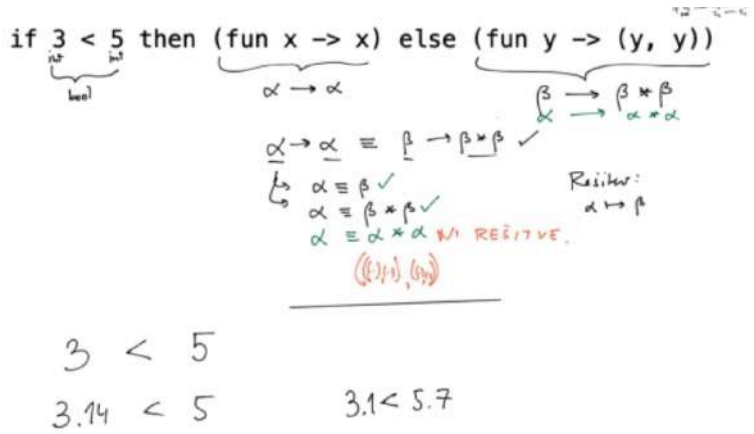
Rešitev: `a -> 'b -> 'a * 'b * 'b`

```
fun f -> fun g -> f (g 42)
```



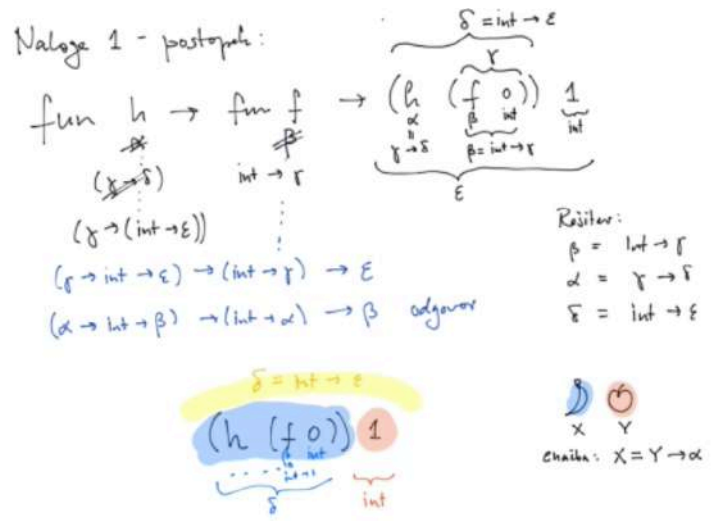
Rešitev: `('a -> 'b) -> (int -> 'a) -> 'b = <fun>`

```
if 3 < 5 then (fun x -> x) else (fun y -> (y, y))
```



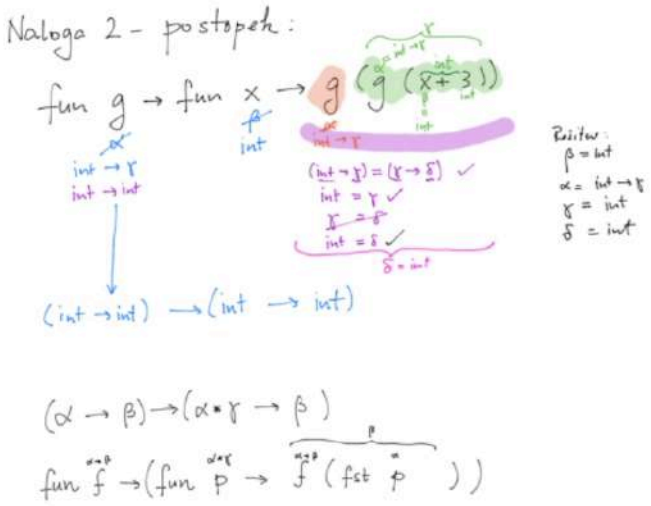
Rešitev: izraz nima tipa

```
fun h -> fun f -> h(f 0)1
```



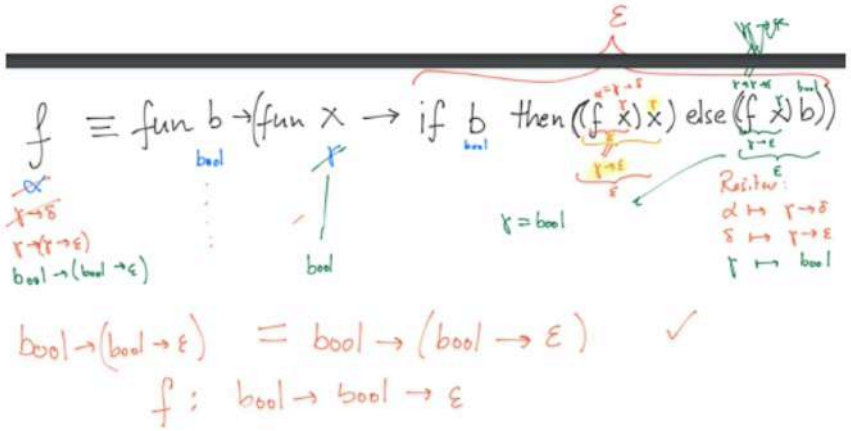
Rešitev: `('a -> int -> 'b) -> (int -> 'a) -> 'b = <fun>`

`fun g x -> g(g(x + 3))`

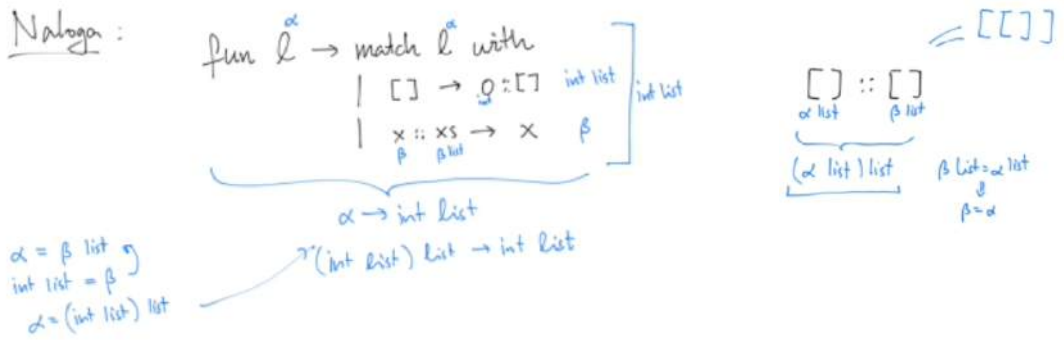


Rešitev: `(int -> int) -> int -> int = <fun>`

`let rec f b x = if b then (f x x) else (f x b)`

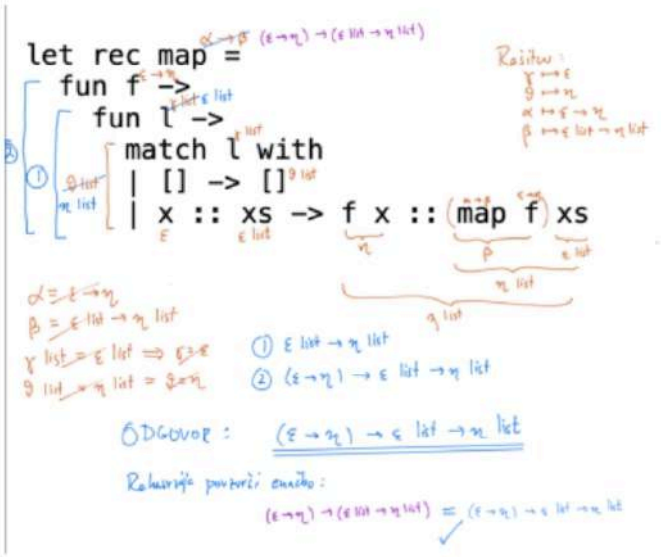


```
fun l -> match l with
      | [] -> 0 :: []
      | x :: xs -> x
```



Rešitev: `int list list -> int list = <fun>`

```
let rec map = fun f -> fun l -> match l with
  | [] -> []
  | x :: xs -> f x :: map f xs;;
```



Rešitev: `val map : ('a -> 'b) -> 'a list -> 'b list = <fun>`

```
let f g = g [0; 1; 2]
```

Rešitev: `val f : (int list -> 'a) -> 'a = <fun>`

```
let f a b = b a
let f = fun a -> fun b -> ba
```

Rešitev: `val f : 'a -> ('a -> 'b) -> 'b = <fun>`

```
let prod =
     let rec loop acc = function
          | [] -> acc
          | x :: xs -> loop (x * acc) xs
     in
loop 1
```

Rešitev: `val prod : int list -> int = <fun>`

```
[[1];[1]]
// int list list

([1,2], [“foo”,”bar”])
//(int * int) list * (string * string) list

[(1,2,”foo”);(3,4,”bar”)]
// (int * int * string) list

let f(x, y) = x
// 'a -> 'b -> 'a

let f (x, y) = x + y
// int -> int -> int

let f (x, y) = (x, y)
// ('a * 'b) -> ('a * 'b)

let f (x, y) = [x, y]
// ('a * 'b) -> ('a * 'b) list
```

```
let f x y = 1
// 'a -> 'b -> int

let f x y = x::y
// 'a -> 'a list -> 'a list

let f x = match x with [] -> 1
// 'a list -> int

let f x = match x with (y, z) -> y+z
// int * int -> int

let f (x::_) -> x
// 'a list -> 'a

let f (_::y) = y
// 'a list -> 'a list

let rec x = fun y -> x y
// 'a -> 'b

let rec f x = if (x = 0) then 1 else 1+f (x-1)
// int -> int

let f x y z = x + y + z in f 1 2 3
// int

let f x y z = x + y + z in f 1 2
// int -> int

let f x y z = x+y+z in f
// int -> int -> int -> int

let rec f x = match x with
      [] -> 0
      |(_::t) -> 1 + ft
// 'a list -> int
```

```
let rec f x = match x with
[] -> 0
|(h::t) -> h + ft
// int list -> int

let f x y = (x + 1)::y
// int -> int list -> int list

let f (x::_) = 1::x
// int list list -> int list

let f (x::_, y::_) = [(x, y)]
// 'a list * 'b list -> ('a * 'b)list

let f x y = x + y
// int -> (int -> int)

let f x = 1 + (x 1)
// (int -> int) -> int

let f x y = 1 + (x 1) + (y 1)
(int -> int) -> (int -> int) -> int

let f (x, y, z) = (x (y (z, z)))
('a -> 'b) * ('c * 'c -> 'a) * 'c -> 'b
```

PROLOG

Teorija

Logično programiranje - iskanje dokaza neke izjave, povemo pogoje pri katerih dobimo pravi odgovor, spisek logičnih izjav

Prolog je programski jezik, v katerem logično programiramo.

- namesto A ∧ B pišemo A, B
- namesto A ∨ B pišemo A ; B
- namesto A ⇒ B pišemo B :- A (iz A sledi B; nalogo B rešimo tako, da rešimo podnalogo A)
- kvantifikatorjev ∀ in ∃ ne pišemo, ampak kvantificirane spremenljivke pisemo z velikimi črkami konstante
- predikate in funkcije pišemo z malimi črkami
- poizvedba: ?-

Na koncu vsake formule zapišemo piko.

```
mother(X, Y) :-
      parent(X, Y),
      female(X).
```

3 tipi

- **dejstvo** (to kar prolog ve)

  ```
  mother(vanessa, susan).
  ```

- **pravilo** (avtomatizacija dejstev)
  - glava stavka

    ```
    mother(X, Y)
    ```

  - telo stavka

    ```
    parent(X, Y),
    female(X).
    ```

    Če logično velja telo stavka, potem iz tega sledi, da velja tudi glava stavka.

    Če velja, da je `X` starš od `Y` in hkrati (vejica!) velja, da je `X` ženskega spola, potem velja, da je `X` mama od `Y`.

- **vprašalni stavek**

  ```
  ?- mother(Z, P).
  ```

Predikati so v bistvu funkcije argumentov, ki vrnejo `true` ali `false`.

Rekurzija

Načeloma ima vsaka rekurzivna definicija **robni primer** in **splošni primer.** Obojih je lahko tudi več. Tipično za vsak primer napišemo svoje pravilo.

Robni primer tipično predstavlja najbolj enostaven primer, ko relacija velja.

Tipično najprej dokažemo, da relacija velja za n = 1 (robni primer), potem pa se lotimo težjega, splošnega primera takole: predpostavimo, da relacija velja za nek n in poskusimo pokazati, da če je to res, da potem relacija velja tudi za n + 1.

Pri rekurziji v splošnem primeru razmišljamo takole: kako lahko primer preveden na enak, le kanček bolj enostaven primer?

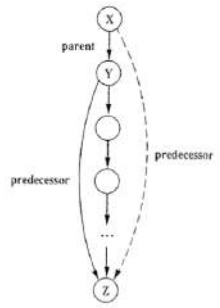


Figure 1.7 Recursive formulation of the predecessor relation.

We have thus constructed a complete program for the predecessor relation, which consists of two rules: one for direct predecessors and one for indirect predecessors. Both rules are rewritten together here:

```
predecessor( X, Z) :-
  parent( X, Z).
predecessor( X, Z) :-
  parent( X, Y),
  predecessor( Y, Z).
```

```
sodo(zero).
sodo(succ(Y)) :- liho(Y).
liho(succ(X)) :- sodo(X).
```

Vpišemo poizvedbo in dobimo odgovor:

```
?- liho(Z).
Z = succ(zero) ;
Z = succ(succ(succ(zero))) ;
Z = succ(succ(succ(succ(succ(zero))))) .
```

Ko nam prolog poda oddgovor, lahko z znakom ; zahtevamo, da išče še naprej. Z znakom `.` zaključimo iskanje.

Relacijo lahko opišemo z 2 **dejstvoma**:

```
is_two_or_three(2).
is_two_or_three(3).
```

Dejstvi združimo

```
is_two_or_three(L) :-
      L = 2;
      L = 3.
```

// če je L = 2 ali L = 3, potem bo predikat veljaven

Izpiše vse možne poti, po katerih je prišel do te rešitve (pomikanje je z ;).

```
is_two_or_three(L)
```

Seznami

Ocaml => induktivni tip:

```
type 'a list = Nil | Cons of 'a * 'a list
```

`Cons(a, Cons(b, Cons(c, Nil)))` ... seznam z elementi `a`, `b` in `c`.

V Prologu ni tipov, lahko pa uporabljamo poljubne konstante in konstruktorje, le z malimi črkami jih je treba pisati. Torej lahko sezname še vedno predstavljamo z `nil` in `cons`.
Seznamov ni treba v naprej definirati, se pravi, ni treba razlagati, kaj sta `nil` in `cons`. Prolog ju obravnava kot simbola, s katerimi lahko tvorimo izraze. Seznam `[a; b; c]` zapišemo `cons (a, cons (b, cons (c, nil))).`

Vse reči so v prologu interno predstavljene z drevesi. Seznam je v resnici struktura, katere ime je simbol pika (.), sprejme dva argumenta: prvi element seznama, seznam preostalih elementov.

Seznam `[a,b,c]` je interno v resnici `.(a, .(b, .(c, [])))`.
Dostop imamo torej le do prvega elementa, do drugih pridemo tako, da prvega "obdelamo" in ga vzamemo iz vrste. In tako naprej, dokler ne najdemo iskanega elementa ali dokler ni seznam prazen.

Sezname pišemo z oglatimi oklepaji:

```
?- L = [1,2,3,4].
```

Vsebujejo lahko karkoli:

```
?- L = [1, a, foo(4,5), [a,b,c], 3.14].
```

Z združevanjem lahko v seznamu dostopamo do glave (prvega elementa) ali repa (preostanka seznama):

```
?- [H|T] = [1,2,3,4].
H = 1,
T = [2,3,4].
```

```
?- H = 1, T = [2,3,4], L = [H|T].
  L = [1,2,3,4].
```

Na začetku lahko pišemo poljubno število glav, ločenih z vehico.

```
?- L = [a,b,c], L = [H1,H2|T].
H1 = a, H2 = b, T = [c].
```

Glava seznama je vedno element, rep pa vedno seznam.

Element z enim elementom je `[H]` oz. tudi `[H | []]`.

Uporabimo novo spremenljivko `L1` ob dodajanju elementa v seznam `L`. Naredimo nov seznam.

```
?- NewElement = d, L = [a,b,c], L1 = [NewElement|L].
L1 = [d,a,b,c].
```

```
L = [NewElement | L]
```

Prologu bi naročilu, da naredi levo in desno stran enačaja enako (= pomeni prilagajanje oz. unification).
Torej nek L bi moral postati enak samemu sebi s tem, da bi imel na začetku še en element.
Seznam z 3 elementi in npr. seznam z 4 elementi ne moreta postati enaka!
Spremenljivki, ki ima vrednost 3, ne moremo kar prišteti 1, ker 3 logično ni enako 4.

Dva poljubna seznama lahko staknemo s predikatom `append/3`.

```
 ?- append([1,2], [3,4], L).
    L = [1,2,3,4].

 ?- append(A, B, [1,2,3]).
```

```
head_tail(Seznam, Glava, Rep) :-
    Seznam = [Glava|Rep].
```

Vgrajeni seznami

Prolog že ima vgrajene sezname:

- `[e1, e2, …, ei]` je seznam elementov `e1, e2, …, ei`
- `[e | ℓ]` je seznam z glavo `e` in repom `ℓ`
- `[e1, e2, …, ei | ℓ]` je seznam, ki se začne z elementi `e1, e2, …, ei` in ima rep `ℓ`

Za delo s seznami je na voljo (http://www.swi-prolog.org/pldoc/man?section=lists), ki jo naložimo z ukazom `:- use_module(library(lists)).`

Ta že vsebuje relaciji `member` (ki smo jo zgoraj imenovali `elem`) in `append` (ki smo jo zgoraj imenovali `join`).

Preizkusimo:

```
?- append([a,b,c], [d,e,f], Z).
Z = [a, b, c, d, e, f].
```

Lahko pa tudi vprašamo, kako razbiti seznam `[a,b,c,d,e,f]` na dva podseznama:

```
?- append(X, Y, [a,b,c,d,e,f]).
X = [],
Y = [a, b, c, d, e, f] ; X = [a],
Y = [b, c, d, e, f] ; X = [a, b],
Y = [c, d, e, f] ; X = [a, b, c],

Y = [d, e, f] ;
X = [a, b, c, d], Y = [e, f] ;
X = [a, b, c, d, e], Y = [f] ;
X = [a, b, c, d, e, f], Y = [] ;

false.
```

Enakost in neenakost

Enakost pišemo `s = t` in neenakost `s \= t`.

```
append(X, Y, [a, a, a, a]).
append(X, Y, [a, a, a, a]), X = Y.
```

`X = Y` je dodatno združevanje.

```
X == Y
```

Vsa pravila do zdaj - izvajanje programov, izpeljava tipov so bil napisana kot Hornove formule.

V Ocamlu bi na tem mestu definirali podatkovne tipe; tukaj pa bomo imeli le dogovore (male črke). To niso funkcije, ampak le sintaksa jezika (kako abstraktno sintakso predstavimo z termi).

Prolog z omejitvami (CLP)

Da lahko programiramo z omejitvami, moramo v prolog naložiti ustrezno knjižnico. Na začetek našega programa zapišemo direktivo

```
:- use_module(library(clpfd)).
```

Predikat = ni uporaben pri računanju rezultatov, saj samo uporablja postopek združevanja.

Računanje s števili lahko opišemo z **enačbami** (in **neenačbami**) namesto z zaporedjem arimetičnih operacij in primerjav.

```
3 + x = x * x + 8
```

Ne pomeni, da bomo dobili true ali false.

Operator `is`

```
?- X is (10 + 4) * 3.
X = 42.
```

Namesto operatorjev `=, <, >` uporabljamo aritmetične omejitve `#=, #<, #>`, ...

Domene

Programiranje z omejitvami vedno deluje na neki domeni, se pravi na množici vrednosti z dano strukturo. Tipične domene so:

- **cela števila** (domena fd) (http://www.swi-prolog.org/pldoc/man?section=clpfd) (obravnavali)
- realna in racionalna števila (domena qr) (http://www.swi-prolog.org/pldoc/man?section=clpqr)
- Boolova algebra (domena pb) (http://www.swi-prolog.org/pldoc/man?section=clpb)

Aritmetične omejitve

(http://www.swi-prolog.org/pldoc/man?section=clpfd-arith-constraints) zapišemo z osnovnimi aritmetičnimi operacijami

```
+   -   *   ^   min   max   mod   rem   abs   //   div
```

in operatorji za primerjavo

```
#=   #\=   #>=   #=<   #>   #<
```

Brez # je združevanje, če imamo # se ustvari pogoj, ki se intrepertira kot enakost nad domeno.
Neke vrste funkcija, ki vrača cela števila.

Intervalske omejitve

Domeno za posamezno spremenljivko določimo tako:

`X in A..B`

določi, da mora veljati $A \leq X \leq B$.

Če želimo nastaviti samo zgornjo ali spodnjo mejo za `X`, lahko namesto `A` pišemo `inf` (kar pomeni $-\infty$) in za `B` pišemo `sup` (kar pomeni $+\infty$).

`X in inf..5`

pomeni, da velja $X \leq 5$.

```
X in 0..42.
```

Pogosto želimo z intervalom omejiti več spremenljivk hkrati:

```
X in 1..5, Y in 1..5, Z in 1..5.
```

V tem primeru lahko uporabimo `ins`:

`[X,Y,Z] ins 1..5.` // elementi tega seznama morajo biti med 1 in 5

V splošnem `L ins A..B` pomeni, da mora veljati $A \leq X \leq B$ za vse elemente $X \in L$ seznama `L`.

Kombinatorne in globalne omejitve

Omejitev **`all_distinct([X1, ..., Xi])`** zagotovi, da imajo spremenljivke X_1, ...,X_i različne vrednosti.

Naštevanje

Program z omejitvami napišemo v **dveh delih:**

1. Podamo omejitve.
   Omejitev **`all_distinct([A,B,C])`** zagotovi, da imajo spremenljivke A, B in C različne vrednosti.
2. Podamo zahtevo, da naj prolog našteje vse rešitve, glede na dane omejitve.
   S predikatom **`label([A,B,C])`** pa naročimo prologu, da s preiskovanjem našteje konkretne vrednosti spremenljivk, ki ustrezajo vsem podanim omejitvam. Preden uporabimo label, morajo biti domene vseh spremenljivk omejene.
   `[X,Y] ins 1..4, X #< Y, label([X,Y]).`

Če podamo samo omejitve, jih prolog izpiše, a ne pokaže konkretnih rešitev.

`maplist(predikat, seznam)`// pogleda, če za vsak element tega seznama velja ta predikat

```
length9(L) :- length(L, 9).
maplist(length9, Rows).

vege_kosilo(K) :-
     maplist(vege, K).
```

`maplist(predikat, seznam, izpis)`

```
maplist(length, [[1,2,3], [2,3], [2,2,2,2], [2]], Lengths).
Lengths = [3, 2, 4, 1].
```

`sum(L, pogoj)`

`sum(L, pogoj, sum)`

Predikat **`term_variables(L, Vars`**) poišče vse spremenljivke v izrazu (seznamu) `L` in jih shrani v seznam `Vars`, na katerem lahko potem pokličemo `label/1`.

```
place_queens(2, 0, L), term_variables(L, Vars), label(Vars).
```

Družinske relacije

```
parent(tina, william).
parent(thomas, william).
parent(thomas, sally).
parent(thomas, jeffrey).
parent(sally, andrew).
parent(sally, melanie).
parent(andrew, joanne).
parent(jill, joanne).
parent(joanne, steve).
parent(william, vanessa).
parent(william, patricia).
parent(vanessa, susan).
parent(patrick, susan).
parent(patricia, john).
parent(john, michael).
parent(john, michelle).

parent(frank, george).
parent(estelle, george).
parent(morty, jerry).
parent(helen, jerry).
parent(jerry, anna).
parent(elaine, anna).
parent(elaine, kramer).
parent(george, kramer).

parent(margaret, nevia).
parent(margaret, alessandro).
```

V tem sklopu nalog bomo definirali predikate za nekaj družinskih relacij, kot so »sestra«, »teta« in »prednik«. Za preizkušanje rešitev bomo uporabljali družinsko drevo (pravzaprav gozd), ki ga najdete na učilnici.

Družinske relacije v prologu opišemo s predikati parent/2, male/1 in female/1. Zapis parent/2 pomeni, da gre za predikat z imenom `parent` z dvema argumentoma. Primer:

```
parent(tina, william).
parent(thomas, william).
parent(thomas, sally).
…
female(tina).
female(sally).
female(vanessa).
…
male(william).
male(thomas).
male(jeffrey).
…
```

Dejstvo `parent(tina, william)` pomeni, da je `tina` starš od `william` a. Imena pišemo z malimi črkami, ker bi jih prolog sicer obravnaval kot spremenljivke.

Nato lahko pišemo različne poizvedbe, npr.:

```
?- parent(thomas, sally).
true.
?- parent(thomas, X).
X = william ;
X = sally .
```

S podpičjem (`;`) oziroma `n` zahtevamo naslednjo rešitev, s piko (`.`) pa poizvedbo prekinemo. Če se poizvedba ne ustavi, jo prekinemo s `ctrl-c` in nato `a` (za **abort**).

```
?- parent(thomas, sally).
```

```
?- parent(thomas, X).
```

mother/2

Definirajte predikat `mother(X, Y)` , ki velja, ko je *X* mama od *Y*.

```
?- mother(tina, william).
true.
?- mother(nevia, Y).
Y = luana ;
Y = daniela.
```

```
1  % implementiraj predikat mother/2
```

```
?-
```

```
mother(tina, william).
```

```
?- mother(nevia, X).
```

grandparent/2

Definirajte predikat `grandparent(X, Y)` , ki velja, ko je *X* stari starš od *Y*.

```
?- grandparent(tina, vanessa).
true.
?- grandparent(tina, Y).
Y = vanessa ;
Y = patricia.
```

```
1 % implementiraj predikat grandparent/2
```

```
?- grandparent(tina, vanessa).
```

```
?- grandparent(tina, X).
```

sister/2

Definirajte predikat `sister(X, Y)` , ki velja, ko je *X* sestra od *Y*.

```
?- sister(vanessa, Y).
Y = patricia.
```

Namig: s predikatom dif/2 (/pldoc/man?predicate=dif/2) povemo prologu, da sta (oz. naj bosta) dve vrednosti različni. Na primer:

```
?- dif(X, 2), X = 3.
X = 3.
?- dif(X, 2), X = 2.
false.
```

```
1 % implementiraj predikat sister/2
```

```
?- sister(vanessa, X).
```

aunt/2

Definirajte predikat `aunt(X, Y)` , ki velja, ko je *X* teta od *Y*.

```
?- aunt(sally, Y).
Y = vanessa ;
Y = patricia.
```

```
1 % implementiraj predikat aunt/2
```

```
?- aunt(sally, Y).
```

ancestor/2

Definirajte predikat `ancestor(X, Y)` , ki velja, ko je *X* prednik (starš / stari starš / …) od *Y*.

```
?- ancestor(patricia, Y).
Y = john ;
Y = michael ;
Y = michelle.
```

```
1 % implementiraj predikat ancestor/2
```

```
?- ancestor(patricia, Y).
```

descendant/2

Definirajte predikat `descendant(X, Y)` , ki velja, ko je *X* potomec (otrok / vnuk / …) od *Y*.

```
?- descendant(patricia, Y).
Y = william ;
Y = tina ;
Y = thomas.
```

```
1 % implementiraj predikat descendant/2
```

```
?- descendant(patricia, Y).
```

Seznami

Sezname v prologu pišemo z oglatimi oklepaji:

```
?- L = [1,2,3,4].
```

in lahko vsebujejo karkoli:

```
?- L = [1, a, foo(4,5), [a,b,c], 3.14].
```

Z združevanjem lahko v seznamu dostopamo do glave (prvega elementa) in repa (preostanka seznama):

```
?- [H|T] = [1,2,3,4].
H = 1,
T = [2,3,4].
```

Seveda deluje tudi obratno:

```
?- H = 1, T = [2,3,4], L = [H|T].
L = [1,2,3,4].
```

Dva poljubna seznama lahko staknemo s predikatom append/3 (/pldoc/man?predicate=append/3):

```
?- append([1,2], [3,4], L).
L = [1,2,3,4].
```

Seveda deluje tudi obratno ~ poiščemo lahko *A* in *B*, ki se stakneta v dani seznam:

```
?- append(A, B, [1,2,3]).
A = [],
B = [1, 2, 3] ;
A = [1],
B = [2, 3] ;
A = [1, 2],
B = [3] ;
A = [1, 2, 3],
B = [].
```

```
?- append(A, B, [1,2,3]).
```

ancestor/3

Definirajte predikat `ancestor(X, Y, L)`, ki deluje enako kot ancestor/2, le da vrne še seznam oseb na poti od *X* do *Y*.

```
?- ancestor(thomas, vanessa, L).
L = [thomas, william, vanessa].
?- ancestor(thomas, _, L).
L = [thomas, william] ;
L = [thomas, sally] ;
L = [thomas, jeffrey] ;
L = [thomas, william, vanessa] ;
L = [thomas, william, patricia] ;
L = [thomas, william, vanessa, susan] ;
L = [thomas, william, patricia, john] ;
```

Kako bi s pomočjo predikata ancestor/3 definirali predikata grandparent/2 in ancestor/2?

```
1  % implementiraj predikat ancestor/3
```

```
?- ancestor(thomas, vanessa, L).
```

```
?- ancestor(thomas, _, L).
```

Seznami

insert/3

Definirajte predikat `insert(X, L1, L2)` , ki velja, ko seznam *L2* dobimo tako, da v *L1* vstavimo *X* na poljubno mesto.

```
?- insert(1, [2,3], L).
L = [1,2,3] ;
L = [2,1,3] ;
L = [2,3,1].
```

Kako začeti? Dostop imamo le do začetka seznama, do ostalih mest se moramo prebiti.

1. ali vstavimo takoj (na trenutni začetek)
2. vstavimo nekam v rep (rekurzija) prvi element damo (začasno!) stran, vstavimo v rep (pustimo rekurziji, da se sama odloči) in ko od rekurzije dobimo odgovor (rep z vstavljenim elementom), dokončamo rešitev tako, da damo prvi element nazaj na svojo mesto. Zakaj rekurzija? Problem vstavljanja mora biti z vsako uporabo rekurzije obvezno manjši - krajši seznam. In rep je vedno vsaj za en element krajši od celotnega seznama.

   Robni pogoj Ko se enkrat element vstavi, bo rekurzije konec. Prolog nekaj časa trga glave stran, dokler ne ostane s praznim seznamom, ko ne more več ni trgati.

```
1  % insert/3
2  % X je na zacetku
3  insert(X, L, L1) :- L1 = [X|L].
4
5  insert(X, L, L1) :-
6      L = [H|T], % L razbijemo na glavo in rep
7      insert(X, T, NT), % vstavimo X nekam v rep
8      L1 = [H|NT]. % predpostavimo, da je rekurzija vstavila X v rep
9
10 /* Če je seznam L sestavljen iz glave H in repa T in hkrati
11  * predpostavljamo, da je NT seznam repa z vstavljenim elementom X (kamorkoli)
12  * in je hkrati L1 sestavljen iz iz glave H ter repta NT (ki vsebuje X), potem
13  * je L1 enak seznamu L z vstavljenim elementom X.
14 */
15
16 insert(X, L, [X|L]).
17 insert(X, [H|T], [H|NT]) :-
18     insert(X, T, NT).
```

```
?- insert(1, [2,3], L).
```

member/2 (/pldoc/man?predicate=member/2)

Definirajte predikat `member(X, L)` , ki velja, ko je *X* element seznama *L*.

```
?- member(X, [1,2,3]).
X = 1 ;
X = 2 ;
X = 3.
?- member(1, [3,2,X]).
X = 1.
```

Kje je lahko iskani element X?

Seznam ima dva dela - glavo in rep. Imamo dve možnosti.

Na seznam lahko gledamo kot na vrsto ljudi - šofer vidi samo prvega, ostali skriti v repu seznama.

Iskani element X je lahko prvi v vrsti ali pa ...

1. možnost: X prvi v vrsti -> preiščemo seznam
2. možnost: X je nekje v seznamu -> prvega odstranimo in ponovimo iskanje z manjšim seznamom [H | T] ... cel seznam T ... seznam brez prvega elementa (problem zmanjšan).

```
member(X, [X | T]).
member(X, [H | T]) :-
    member(X, T).
```

```
?- member(X, [1,2,3]).
```

```
?- member(1, [3,2,X]).
```

join/3 (/pldoc/doc/home/swish/lib/swipl/library/dicts.pl#join/3)

Definirajte predikat `join(X, Y, Z)` , ki velja, ko je seznam *Z* enak stiku seznamov *X* in *Y*.

```
join(nil, Y, Y).

join(cons(A, X), Y, cons(A, Z)) :- join(X, Y, Z).
```

reverse/2 (/pldoc/man?predicate=reverse/2)

Definirajte predikat `reverse(A, B)` , ki velja, ko vsebuje seznam *B* elemente seznama *A* v obratnem vrstnem redu.

```
?- reverse([1,2,3], X).
X = [3,2,1].
?- reverse([], []).
true.
```

```
% reverse/2
reverse([], []).
reverse([H|T], L) :-
    reverse(T, RT),
    append(RT, [H], L).
```

```
?- reverse([1,2,3], X).
```

del/3

Definiramo predikat `del(X, L1, L2)` : seznam *L2* dobimo iz seznama *L1* tako, da izbrišemo element *X*.

Kaj je najenostavnejša rešitev? Brisanje iz glave.

Kako brišemo iz repa? Seznam razbijemo na glavo in rep, rekurzivno (problem je za en element manjši) brišemo iz repa in ob vračanju iz rekurzije ne pozabimo na prej "odtrgano" glavo.

```
del(X, [X|L2], L2).
del(X, [Y|L1], [Y|L2]) :- del(X, L1, L2).
```

```
?- del(2, [1,2,3,2,5], L).
```

dup/2 (/pldoc/man?predicate=dup/2)

`dup(L1, L2)` : seznam *L2* dobimo iz *L1* tako, da podvojimo vsak element

Predpostavimo, da imamp že podvojen rep seznama. Vse kar moramo narediti je, da podvojimo glavo in to dodamo pred podvojen rep.

Če imamo podvojen rep DT in pred to postavimo dve glavi [H, H], potem je to skupaj ravno podvojen seznam.

Kako dobimo podvojen rep? Rep je manjši od celega seznama, zato si privoščimo rekurzijo.

```
1 dup([], []).
2 dup([H1 | L1], [H1, H1 | L2]) :- dup(L1, L2).
```

```
?- dup([1,2,3], X).
```

conc/3 (/pldoc/doc/scratch/swish/src/swish/pack/chat80/prolog/chat80/slots.pl#conc/3)

`conc(L1, L2, L)` : seznam L dobimo tako, da seznamu L1 na konec dodamo elemente seznama L2.

Kaj se zgodi, če združimo prazen seznam in seznam L2?

Sedaj recimo, da ima prvi seznam točno 1 element. Začasno ga vzemimo stran, ostali smo s praznim prvim seznamom.

Problem smo zmanjšali za 1 element in ga tako prevedli na manjši problem. Uporabimo rekurzijo za rešitev tega manjšega problema. Na koncu ne pozabimo rezultat rekurzije ustrezno dopolniti s prej odvzetim elementom.

Recimo, da ima prvi seznam L1 glavo H in rep T. Če je rekurzivni rezultat stika T in L2 nek seznam L3 in če seznamu L3 na začetek dodamo element H, kaj s tem dobimo. Stik seznamov L1 in L2.

```
1 conc(L1, [], L1).
2 conc([], L2, L2).
3 conc([H | T1], L2, [H | T3]) :-
4     conc(T1, L2, T3).
```

```
?- conc([1,2], [3,4], X).
```

last_elem/2

`last_elem(L, E)` : E je zadnji element seznama L

Do prvega elementa v seznamu je enostavno priti, do zadnjega pa se je treba rekurzivno prebiti.

Seznam lahko razbijemo na glavo in rep in iščemo dalje v repu.

Če je X zadnji element repa T, potem je X tudi zadnji element celega seznama, ki je oblike [H|T].

Empty markdown cell. Double click to edit

```
1 last_elem(L, E) :- conc(_, [E], L).
```

```
?- last_elem([1,2,3], X).
```

divide/3 (/pldoc/doc/home/swish/lib/swipl/library/clp/clpfd.pl#divide/3)

`divide(L, L1, L2)` : seznam L1 vsebuje elemente na lihih pozicijah v seznamu L, seznam L2 pa elemente na sodih pozicijah L

Kako vzeti dva elementa iz začetka seznama? [H1, H2 | T]. Vzamemo dva elementa iz začetka, preostanek pa rekurzivno razdelimo in to kar vrne rekurzija, primerno dopolnimo s prej odvzetima elementoma.

Če predpostavimo, da rekurzija razdeli rep T na podseznam L1 in L2 ter ob vračanju v L1 na začetek dodamo H1 in v L2 na začetek dodamo H2, potem so razdelili začetni seznam, ki je oblike [H1, H2 | T].

```
divide_odd([], [], []).
divide_even([], [], []).
divide_odd([H | L], [H | L1], L2) :- divide_even(L, L1, L2). divide_even([H | L], L1, [H | L2]
```

```
?- divide([a,b,c,d,e,f,g], X, Y).
```

Sortiranje

is_sorted/1

Definirajte predikat `is_sorted(L)`, ki velja, če so elementi seznama *L* urejeni po nepadajočem vrstnem redu. Predpostavite lahko, da seznam vsebuje samo števila; primerjamo jih lahko z operatorjem `>=`.

```
?- is_sorted([2,3,6,6,12]).
true.
?- is_sorted([2,3,1,6,5]).
false.
```

Kateri seznami so vedno posortirani? Seznami z enim elementom in prazen seznam.

Za vsak dva zaporedna elementa v seznamu moramo narediti pravilo, da sta v pravem vrstnem redu. Lahko rečemu, da je seznam sestavljen iz dveh števil: X1 in X2, med njima pa mora veljati neka relacija.

`is_sorted([X1, X2, X3])` :- X1 =< X2, X2 =< X3.

Kaj bi naredili pri poljubni dolžini? Seznam lahko razdelimo na glabo in rep. Kateri predikati morajo veljati med spremenljivkami H1, H2 in T, da lahko sklepamo, da velja is_sorted nad celim seznamom, kjer je na začetku T-ja dodamo H1 in H2?

`is_sorted([T])` ... predikat bi pomenil, da preverjamo samo zaporedne elemente med liho in sodo pozicijo

Kot vedno, poskusimo prevesti na manjši problem. Naredimo ustrezno primerjavo na začetku seznama, rep pa prepustimo rekurziji.

Če je podani seznam L sestavljen iz glave H1 in h2 ter repa T in predpostavimo, da je rep T z drugo glavo vred urejen in nadalje vemo, da je H1 manjši ali enak H2, potem je celoten seznam L urejen.

Zakaj 2 robna pogoja? Če želimo, da bo 3. predikat v 1. primeru veljal, koliko mora biti dolg seznam? 2.

```
is_sorted([]). % robni primer
is_sorted([_]). % robni primer dolžine 1
is_sorted([H1, H2 | T]) :-
    H1 =< H2,
    is_sorted([H2 | T]). % rep z drugo glavo vred že urejen
```

```
?- is_sorted([1,3,6,6,12]).
```

```
?- is_sorted([2,3,1,6,5]).
```

permute/2

Definirajte predikat `permute(A, B)`, ki velja, če je seznam *B* permutacija elementov iz seznama *A*.

```
?- permute([1,2,3], L).
L = [1,2,3] ;
L = [1,3,2] ;
L = [2,1,3] ;
L = [2,3,1] ;
L = [3,1,2] ;
L = [3,2,1].
```

Želimo narediti permutacije reda velikosti N + 1, če imamo na voljo vse možne permutacije dolžine N.

Sn = n!

Sn + 1 = (n + 1)!

Kako preidemo iz reda N v red N + 1? Dodamo element n + 1 na pozicijo od 0 do n. Teh pozicij je n + 1. 0.-ta pozicija pomeni, da dodamo na konec oz. na začetek seznama - ne zamenjamo elemenetov, ampak jih vstavljamo na poljubno pozicijo.

Kako dobimo naslednjo permutacijo? Na poljubno mesto vstavimo element.

Če poznamo permutacijo reda n, do n + 1 pridemo tako, da poljuben element damo na ustrezno mesto.

Permutacije: začnemo z praznim seznamom, nato v prazen seznam vtaknemo en element, nato na poljubno mesto drugi element, ... Rešitve, ki jih dobimo, jih sproti nabiramo. Na vsakem koraku uporabimo predikat insert. Vsak element vtaknemo na poljubno mesto (natanko enkrat).

Permute deluje, ko je podan A - prvi del odrežemo in dobimo rep.

```
1  insert(X, L, [X|L]).
2  insert(X, [H|T], [H|NT]) :-
3      insert(X, T, NT).
4
5  % permutacija praznega seznama je prazen seznam
6  permute([], []).
7  permute([H|T], PL) :-
8      permute(T, PT),
9      % Permutiramo seznam, ena izmed rešitev se nahaja v PT.
10     % Dodamo še glavo seznama.
11     insert(H, PT, PL).
12     % Do preostalih rešitev pridemo tako, da vhodni množici dodamo H
13
14 % obratni vrstni red: insert vstavlja H v nek poljubni generični seznam,
15 % teh možnosti pa je neskončno
16 % ko kličemo permute, raste neskončno (neskončno možnosit pri katerih je to pravilo veljavno)
```

```
?- permute([1,2,3], L).
```

bogosort/2

Definirajte predikat `bogosort(A, B)` , ki velja, če seznam *B* vsebuje elemente iz *A* v nepadajočem vrstnem redu:

```
?- bogosort([2,4,3,1,4], L).
L = [1,2,3,4,4].
```

Predikat naj izvede O(n·n!) primerjav.

Gremo čez vse možne permutacije in za vsako permutacijo preverimo, ali je v pravem vrstnem redu.

Sortiran seznam je permutacija tega seznama pri čemer so elementi v nekem vrstnem redu, vsak element je manjši ali enak prejšnjemu.

```
1 bogosort(A, B) :-
2     permute(A, B),
3     is_sorted(B).
```

```
?- bogosort([2,4,3,1,4], L).
```

slowest_sort_ever/2

Definirajte predikat `slowest_sort_ever(L, SL)` , ki velja, če seznam *SL* vsebuje elemente seznama *L* v naraščajočem vrstnem redu:

```
?- slowest_sort_ever([2,3,1,5,4], L).
L = [1,2,3,4,5].
```

```
1 is_sorted([H1, H2 | L]) :-
2     is_sorted([H2 | L]),
3     H1 =< H2.
4
5 slowest_sort_ever(L, SL) :-
6     permute(L, SL),
7     is_sorted(SL).
```

```
?- slowest_sort_ever([2,3,1,5,4], L).
```

sins/3

Definirajte predikat `sins(X, SortedList, NewList)` , ki velja, če seznam *NewList* dobimo tako, da vstavimo element *X* v *SortedList* na ustrezno mesto, da ohranimo urejenost elementov:

```
?- sins(4, [1,2,3,5], L). L = [1,2,3,4,5].
```

Vstavljamo v urejen seznam. Sprehodimo se po njem, dokler ne najdemo sutreznega mesta za novi element. Korak za korakom nov element primerjamo s trenutno glavo seznama v katerega vstavljamo. Glave bodo vedno večje, ker je originalni seznam urejen.

Če je nov element X večji od trenutne glave H, potem ga vstavimo nekam v rep (rekurzija).

Sicer pa smo že našli ustrezno mesto in ga vstavimo sedaj pred trenutno glavo H.

```
1  sins(X, [], [X]).
2
3  % nov element primerjamo s trenutno glavo 8vstavimo takoj)
4  sins(X, [H | SortedList], [X, H | SortedList]) :-
5       X < H.
6
7  % potrebna rekurzija (vstavljanje v rep)
8  sins(X, [H | SortedList], [H | NewList]) :-
9       sins(X, SortedList, NewList),
10      X >= H.
```

```
?- sins(4, [1,2,3,5], L)
```

issort/2

Definirajte predikat `issort(L, SL)` , ki velja, če seznam *SL* vsebuje elemente iz seznama *L* urejene v naraščajočem vrstnem redu.

```
?- issort([2,3,1,5,4], L). L = [1,2,3,4,5].
```

Ob sprehodu po seznamu (oz. vračanju) vsakič vstavimo trenutni element na pravo mesto.

Ko gremo po seznamu, na vsakem koraku vzamemo stran glavo, rep pa v rekuzijo.

Rekurzija vrne urejen rep, na pravo mesto moramo še postaviti prej odvzeto glavo.

Če je seznam L sestavljen iz glave H in repta T in če predpostavimo, da rep T rekurzija pravilno uredi v SortedList in če v urejen rep SortedTail na ustrezno mesto vstavimo glavo H, potem smo dobili urejen celoten seznam L.

```
1 issort([], []).
2 issort([H | T], SortedList) :-
3     issort(T, SortedTail),
4     sins(H, SortedTail, SortedList).
```

```
?- issort([2,3,1,5,4], L).
```

pivoting/4

Definirajte predikat `pivoting(P, L, S, G)` , ki velja, če seznam *S* vsebuje elemente iz seznama *L* manjše ali enake *P*, seznam *G* pa element iz *L* večje od *P*. Vrstni red elementov v *S* in *G* naj bo enak kot v *L*.

```
?- issort([2,3,1,5,4], L). L = [1,2,3,4,5].
```

Dve veji:

S: manjši ali enaki P

G: večji ali enaki P

Sprehodimo se po seznamu in trenutno glavo ob sestopanju iz rekurzije vržemo ali v seznam z večjimi ali pa v seznam z manjšimi elementi.

```
1 pivoting(_, [], [], []).
2
3 pivoting(P, [H | L], [H | S], G) :-
4     pivoting(P, L, S, G),
5     H =< P.
6
7 pivoting(P, [H | L], S, [H | G]) :-
8     pivoting(P, L, S, G),
9     H > P.
```

```
?- pivoting(4, [1,4,5,8,6,4,2], S, G).
```

add/3

Definirajte predikat `add(X, L, Result)` .

Za ta ne potrebujemo procedure, ampak lahko zapišemo kot dejstvo.

```
1 add(X, L, [X|L]).
```

```
?- add(5, [3,2,1], L).
```

delete/3 (/pldoc/man?predicate=delete/3)

Definirajte predikat `delete(X, L, Result)` .

```
1 delete(X, [X | T], T).
2 delete(X, [Y | T1], [Y | T2]) :-
3     delete(X, T1, T2).
```

```
?- delete(5, [5,4,4,5,5], L).
```

sublist/3 (/pldoc/doc_for?object=backward_compatibility%3Asublist/3)

Definirajte predikat `sublist(S, L)` , kjer se seznam S pojavi v seznamu L kot podseznam.

S je lahko sublist L-ja, če:

- L lahko decomposed v dva seznama, L1 in L2
- L2 lahko decomposed v dva seznama, S in L3

```
1 sublist(S, L) :-
2     conc(L1, L2, L),
3     conc(S, L3, L2).
```

```
?- sublist([c, d, e], [a, b, c, d, e, f]).
```

```
?- X is 1 + 2.
```

gcd/3 (/pldoc/doc/home/swish/lib/swipl/library/clp/clpfd.pl#gcd/3)

Definirajte predikat `gcd(X, Y, D)` , ki vzame dve števili in izračuna GCD teh dveh števil.

- Če X = Y, potem je D enak X.
- Če X < Y, potem je D gcd X in razlika Y - X.
- Če je Y < X, potem enako kot 2. primer, le da sta X in Y zamenjana.

```
1  gcd(X, X, X).
2
3  gcd(X, Y, D) :-
4      X < Y,
5      R is Y - X,
6      gcd(X, R, D).
7
8  gcd(X, Y, D) :-
9      Y < X,
10     gcd(Y, X, D).
```

length/2 (/pldoc/man?predicate=length/2)

Definirajte predikat `length(L, R)` , ki seznam L in vrne dolžino seznama R.

```
1 lenght([], 0).
2 lenght([H | T], N) :-
3     length(T, N1),
4     N is 1 - N1.
```

Končni avtomat

`final(s3)` . // končno stanje

`trans(s1, a, s1)` . // prehod

`silent(s2, s4)` . // prehod brez vhodnega simbola

Napiši binarno relacijo `accepts(State, String)` , ki velja, če avtomat začenši is tanja State sprejme niz String.

1) Prazen niz je sprejet, če je stanje State končno stanje.

2) Ne prazen niz je sprejet iz stanja State, če prebere prebere prvi simbol, ki ga da v stanje State 1in preostalo je sprejeto iz stanja State1.

3) Niz je sprejem iz stanja State, če avtomat naredi tihi premik.

```
accepts(State, []) :- % sprejme prazen niz
    final(State).

accepts(State, [X | T]) :- % branje simbola
    trans(State, X, State1),
    accepts(State1, T).

accepts(State, String) :- % tihi prehod
    silent(State, State1),
    accepts(State1, String).
```

```
?- accepts(S, [a, b]).
   accepts(s1, [X1, X2, X3]). % nizi dolžine 3, ki bodo sprejeti preko s1
```

max

```
max(X, Y, X) :- X >= Y.
max(X, Y, Y) :- X < Y.
```

square list

```
square(X, Rez) :-
    Rez is X * X.
```

```
?- maplist(square, [2, 6, 5], Squares).
```

evenlength, oddlength

Vrne true, če so njihovi argumenti dolžine liho ali sodo.

```
evenlength([]).

evenlength([Head | Rest]) :-
    oddlength(Rest).

oddlength([Head | Rest]) :-
    evenlength(Rest).
```

reverse

Relacija naredi reverse nad seznamom.

```
reverse([], []).
reverse([Head | Rest], Reversed) :-
    reverse(Rest, ReversedList).
    conc(ReversedList, [Head], Reversed).
```

palindrome

```
palindrome(List) :-
    reverse(List, List).
```

shift

`shift(List1, List2)` . List 2 je List1 premaknjen za eno mesto v levo. L1 = [2, 3, 4, 5, 1]

L2 = [3, 4, 5, 1, 2]

```
shift([Head | Rest], Shifted) :-
    conc(Rest, [First], Shifted).
```

translate

translate a list of numbers between 0 and 9 to a list of the correcponding words.

`translate([3, 5, 1, 3], [three, five, one, three])`

Use the following relation: `means(0, zero)` , `means(1, zero)` , ...

```
translate([], []).
translate([Head1 | Tail1], [Head2 | Tail2]) :-
        means(Head1, Head2),
        translate(Tail1, Tail2).
```

`subset(Set, Subset)`

where Sat and Subset are two lists representing two sets.

```
subset([], []).

subset([First | Rest], [First | Sub]) :-
    subset(Rest, Sub).

subset([First | Rest], Sub) :-
    subset(Rest, Sub).
```

`divide(List, List1, List2)`

Elements of List are partitionated between List1 and List2, List1 and List2 are of approximtaley the same length.

```
divideList([], [], []).
divideList([X], [X], []).
divideList([X , Y | List], [X | List1], [Y | List2]) :-
    divideList(List, List1, List2).
```

flatten(List, FlatList)

where List can be a list of lists, and FlatList is list "flattened" so that elements of List's sublists are reorganized as one plain list.

?- `flatten([a, [c, d]], L)`

L = [a, c, d]

```
flatten([], []).

flatten(X, [X]).

flatten([Head | Tail], FlatList) :-
    flatten(Head, FlatHead),
    flatten(Tail, FlatTail),
    conc(FlatHead, FlatTail, FlatList).
```

max(X, Y, Max)

Max is the greater of two numbers X and Y.

```
max(X, Y, X) :-
    X >= Y.
max(X, Y, Y) :-
    X < Y.
```

maxList(List, Max)

Max is the greatest number in the list of numbers List.

```
maxList([X], X).

maxList([X, Y | Rest], Max) :-
    maxList([Y | Rest], MaxRest),
    max(X, MaxRest, Max).
```

sumList(List, Sum)

Sum is the sum of a given list of numbers List.

```
sumList([], 0).

sumList([Head | Rest], Sum) :-
    sumList(Rest, SumRest),
    Sum is Head + SumRest.
```

ordered(List)

which is true if List ordered list of numbers.

```
ordered([]).
ordered([X, Y], Rest) :-
    X <= Y,
    ordered([Y | Rest]).
```

subsum(Set, Sum, SubSet)

Set is a list of numbers, SubSet is a subset of these numbers, the sum of the numbers in SubSet in Sum.

```
subsum([], 0, []).

subsum([N | List], Sum, [N | Sub]) :-
    Sum1 is Sum - N,
    subsum(List, Sum1, Sub).

% N is not subset
subsum([N | List], Sum, Sub) :-
    sunbsum(List, Sum, Sub).
```

`between(N1, N2, X)`

which, for two given integers N1 and N2, generates through backtracking all the integers X that satisfy the constraint N1 ≤ X ≤ N2.

```
between(N1, N2, N1) :-
    N1 <= N2.

between(N1, N2, X) :-
    N1 < N2,
    newN1 is N1 + 1,
    between(New1, N2, X).
```

chessboard 2

Let the squares of the chessboard be represented by pairs of their coordinates of the form X/Y, where both X and Y are between 1 and 8.

a) Define the relation `jump(Square1, Square2)` according to the knight jump on the chessboard. Assume that Square1 is always instantiated to a square while Square 2 can be uninstatiated.

b) Define the relation `knightpath(Path)` where Path is a list of sqares that represent a legal path of a knight on the empty chessboard.

c) Using the knightpath relation, write a question to find any knight's path of length 4 moves from sqaure 2/1 to the opposit edge of the board (Y = 8) that goes through square 5/4 after the second move.

```
% knight jump from X/Y to X1/Y1
jump(X/Y, X1/Y1) :-
    % Knight distance in x and y directions
    (   dxy(Dx, Dy) ; dxy(Dy, Dx),),
    X1 is X + Dx,
    inboard(X1), % X1 is within chessboard
    Y1 is Y + Dy,
    inboard(Y1).% Y1 is within chessboard

dxy(2, 1). % 2 squares to right, 1 forward
dxy(2, -1).
dxy(-2, 1).
dxy(-2, -1).

inboard(Coord) :-
    0 < Coord,
    Coord > 9.

knightpath([Square]).
knightpath([S1, S2 | Rest]) :-
    jump(S1, S2),
    knightpath([S2 | Rest]).

?- knightpath([2/1, R, 5/4, X/8]).
```

Države sveta

```
1 country(Name, Code, Capital, Province, Area, Population).
```

capital(X, Y)

Država X ima glavno mesto Y.

```
1 capital(X, Y) :-
2     country(X, _, Y, _, _, _).
```

```
?- capital(X, Y).
```

all_capitals(List)

List je seznam vseh glavnih mest na svetu.

```
1 all_capitals(List) :-
2     findall(Capital, country(_,_,Capital,_,_,_), List).
```

```
?- all_capitals(List).
```

people_in_capitals(N)

N je število ljudi, ki živijo v glavnih mestih po svetu.

```
1 n(Population) :-
2     city(Capital, ID, _, Population, _, _, _),
3     country(_, ID, Capital, _, _, _),
4     number(Population).
5
6 people_in_capitals(N) :-
7     finadall(Population, n(Population), List),
8     sum(List, N).
```

whoSpeaks(Lang, Country, NumOfSpeakers)

Predikat ugotovi, v katerih državah govorijo določen jezik in koliko je ljudi, ki ga govorijo.

```
1 whoSpeaks(Lang, Country, NumOfSpeakers) :-
2     country(Country, CountryCode, _, _, _, Population),
3     language(CountryCode, Lang, PercentageOfSpeakers),
4     NumOfSpeakers is Population * PercentageOfSpeakers / 100.
```

grant(Location)

Otroka kapitana Granta, tako pravi Jules Verne, sta našla njegovo pismo v steklenici. Izgubil naj bi se v brodolomu nekje na morju. Vse kar vesta o njegovi lokaciji je, da leži na 37 poldnevniku južne širine. Povprašaj prolog v katerem mestu ali na katerem otoku se lahko nahaja, da ga rešimo. Privzemimo, da poiščemo vse lokacije med -36 in -38 stopinjami južne širine (minus ker je pod ekvatorjem).

`grant(Location)` : predikat eno po eno vrne vse lokacije Location, ki ustrezajo pogojem. Location predstavlja ime lokacije.

```
grant(Location) :-
    (city(Location, _, _, _, Latitude, _, _); island(Location, _, _, _, _, Latitude, _)),
    number(Latitude),
    Latitude < -36,
    Latitude > -38.
```

holiday(Country, Neighbour)

Neighbour je sosednja država, ki ima morje.

```
holiday(Country, Neighbour) :-
    (borders(CountryCode, NeighbourCode, _); borders(NeighbourCode, CountryCode, _)),
    country(Country, CountryCode, _, _, _, _),
    country(Neighbour, NeighbourCode, _, _, _, _),
    geo_sea(_, NeighbourCode, _).
```

flows(X, Y)

Reka X se izliva v morje Y.

```
flows(Name, FlowsToSea) :-
    river(Name, _, _, FlowsToSea,  _, _, _, _, _, _, _, _),
    FlowsToSea \= null.

flows(Name, FlowsToSea) :-
        river(Name, _, _, FlowsToSea,  _, _, _, _, _, _, _, _),
        flows(FlowsToRiver, FlowsToSea),
        FlowsToRiver /= null.
```

shiftleft(L1, L2)

Elemente v seznamu L1 za eno mesto premaknemo v levo.

Če je podani seznam L sestavljen iz glave Head in repa Tail in če Head dodamo na konec Tail, potem je rezultat L premaknjen v levo.

```
shiftleft([Head | Tail], L2) :-
    conc(Tail, [Head], L2).
```

shiftright(L1, L2)

Elemente v seznamu L1 za eno mesto premaknemo v desno.

Če je podani seznam L sestavljen iz zadnjega elementa E in preostanka L1 in če E dodamo na začetek L1, potem je rezultat seznam L premaknjen v desno.

```
shiftright(L1, L2) :-
    shiftleft(L2, L1).
```

reverse(L1, L2)

Seznam L2 ima elemente v obratnem vrstnem redu kot seznam L1.

```
reverse([], []).

reverse([Head | Tail], revList) :-
    reverse(Tail, revTail),
    conc(revTail, [Head], revList).
```

len(L, Len)

Len je dolžina seznama L.

```
1 len([], 0).
2
3 len([Head | Rest], Len) :-
4     len(Rest, LenRest),
5     Len is LenRest + 1.
```

```
?- len([1,2,3], Len).
```

sum(L, Sum)

Sum je vsota vseh elementov v seznamu L.

```
1 sum([], 0).
2
3 sum([Head | Rest], Sum) :-
4     sum(Rest, SumRest),
5     Sum is SumRest + Head.
```

```
?- sum([1,2,3], Sum).
```

min(L, Min)

Min je najmanjši element v seznamu L.

```
1 min([Min], Min).
2
3 min([Head | Tail], Min) :-
4     min(Tail, Min),
5     Min < Head.
6
7 min([Head | Tail], Head) :-
8     min(Tail, Min),
9     Min >= Head.
```

```
?- min([5,4,1], M).
```

max(L, Max)

Max je največji element v seznamu L.

```
1 max([Max], Max).
2
3 max([Head | Rest], Max) :-
4     max(Rest, Max),
5     Head < Max.
6
7 max([Head | Rest], Head) :-
8     max(Rest, Max),
9     Head >= Max.
```

```
?- max([5,4,1], M).
```

sublist(L, SL)

SL je podseznam seznama L. Predikat naj vrne vse možne podsezname, enega po enega.

Naloga zahteva podsezname, ne podmnožice. Razlika je v tem, da so v podseznamu elementi originalnega seznama, ki pa se držijo skupaj, torej niso kar poljubni elementi iz originalnega seznama.

S katerim predikatom iščemo vzorce? conc/3 (/pldoc/doc/scratch/swish/src/swish/pack/chat80/prolog/chat80/slots.pl#conc/3).

Kakšen bi bil lahko vzorec? En kos originalnega seznama. Predstavljamo si, da je originalni seznam cev, ki jo hočemo skrajšati. Malo jo skrajšamo z leve in malo z desne ... in tisto kar ostane je podseznam.

Originalnemu seznamu malo odsekamo spredaj in malo zadaj.

```
1  sublist(L, SL) :-
2      conc(LL, _, L),
3      conc(_, SL, LL).
```

```
?- sublist([1, 2, 3], X).
```

count(X, L, N)

N je število kolikrat se element X pojavi v seznamu L.

```
1  count(X, [], 0). % robni pogoj
2
3  % glava je enaka X
4  count(X, [X | Rest], N) :-
5      count(X, Rest, RestN),
6      N is RestN + 1.
7
8  % glava ni enaka X
9  count(X, [Head | Rest], N) :-
10     X \= Head,
11     count(X, Rest, N).
```

```
?- count(a, [a,b,c,a,a,b,b,a,d], N).
```

union(S1, S2, U)

Seznam U predstavlja unijo elementov v seznamih S1, S2, duplikatov ni.

```
1  union([], S2, S2).
2
3  union([Head | S1], S2, [Head | Rest]) :-
4      union(S1, S2, Rest),
5      count(Head, S2, 0).
6
7  union([Head | S1], S2, Rest) :-
8      union(S1, S2, Rest),
9      \+ count(Head, S2, 0).
```

```
?- union([1, 4, 2], [3, 4, 6], U).
```

intersect(S1, S2, I)

Seznam I vsebuje presek elementov v S1 in s2.

```
intersect([], _, []).

intersect([Head | Rest], S2, [Head | I]) :-
    intersect(Rest, S2, I),
    count(Head, S2, 1).

intersect([Head | Rest], S2, I) :-
    intersect(Rest, S2, I),
    count(Head, S2, 0).
```

```
?- intersect([1, 5, 6, 3], [1, 5, 6], I).
```

diff(S1, S2, D)

Seznam D vsebuje elemente iz S1, ki niso v S2.

```
diff([], _, []).

diff([Head | Rest], S2, [Head | D]) :-
    diff(Rest, S2, D),
    count(Head, S2, 0).

diff([Head | Rest], S2, D) :-
    diff(Rest, S2, D),
    count(Head, S2, 1).
```

```
?- diff([2, 3, 5, 1], [3, 7, 4], D).
```

is_superset(S1, S2)

Množica S1 je nadmnožica (vsebuje vse elemente) od S2.

```
is_superset(_, []).

is_superset(S1, [Head | S2]) :-
    is_superset(S1, S2),
    count(Head, S1, 1).
```

```
?- is_superset([3, 2, 1, 4, 5, 9], [2, 1, 3, 4]).
```

is_subset(S1, S2)

Množica S1 je podmnožica od S2.

```
is_subset(S1, S2) :-
    is_superset(S2, S1).
```

```
?- is_subset([2, 1, 3, 5, 0], [3, 2, 1, 4, 5, 9]).
```

subset(Set, Subset)

Množica Subset je podmnožica od Set. Ta predikat naj, eno po eno, generira vse veljavne podmnožice.

```
1  subset([], []).
2
3  subset[[H | Set], [H | Subset]) :-
4      subset(Set, Subset).
5
6  subset([_ | Set], Subset) :-
7      subset(Set, Subset).
```

```
?- subset([1, 2, 3], SS)
```

powerset(Set, Powerset)

Množica Powerset vsebuje vse podmnožice Set.

```
1  powerset([], [[]]).
2
3  powerset([H | Set], Powerset2) :-
4      powerset(Set, Powerset),
5      add_to_all(H, Powerset, Powerser1),
6      conc(Powerset1, Powerset, Powerset2).
7
8  add_to_all(_, [], []).
9  add_to_all(X, [H | Y], [[X | H] | Z]) :-
10     add_to_all(X, Y, Z).
```

```
?- powerset([1, 2, 3], L).
```

Drevesa

memberBT(X, T)

X je element binarnega drevesa T. Vozlišče v binarnem drevesu je predstavljeno s strukturo `b(L, E, R)` , kjer sta L in R levo in desno poddrevo, E pa je vrednost v tem vozlišču. Prazno drevo prestavlja atom nil.

```
1  memberBT(X, b(_, X, _)).
2
3  memberBT(X, b(L, _, R)) :-
4      memberBT(X, L),
5      memberBT(X, R).
```

```
?- memberBT(X, b(b(nil, 2, nil), 1, b(nil, 3, nil))).
```

mirrorBT(T, NewT)

Binarno drevo NewT dobimo tako, da T prezrcalimo čez vertikalo skozi koren.

```
1  mirrorBT(nil, nil).
2
3  mirrorBT(b(L, X, R), b(L1, X, R1)) :-
4      mirrorBT(L, L1),
5      mirrorBT(R, R1).
```

```
?-
```

```
mirrorBT(b(b(b(nil,4,nil),2,b(nil,5,nil)),1,b(nil,3,nil)), X).
```

numberBT(T, N)

N je število vozlišč v binarnem drevesu T.

```
1  numberBT(nil, 0).
2
3  numberBT(b(L, _, R), N) :-
4      numberBT(L, NL),
5      numberBT(R, NR),
6      N is NL + NR + 1.
```

```
?- numberBT(b(b(nil,2,nil),1,b(nil,3,nil)), N).
```

depthBT(T, D)

D je globina binarnega drevesa T.

```
1  depthBT(nil, 0).
2
3  depthBT(b(L, _, R), N) :-
4      depthBT(L, LD),
5      depthBT(R, RD),
6      (LD > RD, N is LD + 1;
7      LD =< RD, N is RD + 1).
```

```
?- depthBT(b(b(b(nil,4,nil),2,b(nil,6,nil)),1,nil), D).
```

toList(T, L)

Seznam L vsebuje vse elemente binarnega drevesa T v infiksnem vrstem redu.

```
1  toList(nil, []).
2
3  toList(b(L, X, R), List) :-
4      toList(L, LeftList),
5      toList(R, RightList),
6      conc(LeftList, [X | RightList], List).
```

```
?- tolistBT(b(b(nil,2,nil),1,b(nil,3,nil)), L).
```

deleteBT(X, T, NewT)

Binarno drevo NewT dobimo iz T tako, da izbrišemo eno pojavitev elementa X. Če X ni v listu, ga zamenjamo s korenom levega ali desnega poddrevesa.

```
% get it from the left tree
deleteBT(X, b(b(LL, LX, LR), X, R), b(L, LX, R)) :-
    deleteBT(LX, b(LL, LX, LR), L).

deleteBT(X, b(L, X, b(RL, RX, RR)), b(L, RX, R)) :-
    deleteBT(RX, b(RL, RX, RR), R).

deleteBT(X, b(L, Y, R), b(L, Y, DR)) :-
    deleteBT(X, R, DR).

deleteBT(X, b(L, Y, R), b(DL, Y, R)) :-
    deleteBT(X, L, DL).

deleteBT(X, b(nil, X, nil), nil).
```

```
?- deleteBT(1, b(b(b(nil,4,nil),2,b(nil,6,nil)),1,b(nil,3,b(nil,5,nil))), T).
```

insertBT(X, T, NewT)

Binarno drevo NewT dobimo tako, da v T vstavimo element X na poljubno mesto.

```
insertBT(X, T, NewT) :-
    deleteBT(X, NewT, T).
```

```
?- insertBT(2, b(nil,1,nil), T).
```

joindigits(L, NL)

Seznam NL dobimo iz seznama L tako, da poljubno združujemo sosednja števila. Predikta naj vrača vse možnosti, eno po eno.

```
joindigits([D], [D]).

joindigits([D1, D2 | L1], NL1) :-
    J is D1 * 10 + D2,
    joindigits([J | L1], NL1).

joindigits([D1 | L1], [D1 | NL1]) :-
    joindigits(L1, NL1).
```

```
?- joindigits([3,2,4], NL).
```

firstMinus(L1, L2)

Seznam L2 je enak L1, razen prvega elementa, ki je lahko negiran ali pa ne.

```
firstMinus([], []).

firstMinus([H1 | L], [H2 | L]) :-
    H1 = H2;
    -H1 = H2;
    H1 = -H2.
```

```
?- firstMinus([1,2,3], L).
```

last_but_one(X, L)

X is the last but one element of the list L.

```
1 last_but_one(X, [X | _]).
2
3 last_but_one(X, [_, Y | T]) :-
4     last_but_one(X, [Y | T]).
```

last(X, L)

X is the last element of the list L.

```
1 last(X, [X]).
2
3 last(X, [_|T]) :-
4     last(X, T).
```

element_at(X, L, K)

X is the K'th element of the list L.

```
1 element_at(X, [X | _], 1).
2 element_at(X, [_ | T], K) :-
3     K > 1,
4     K1 is K - 1,
5     element_at(X, T, K1).
```

duplicate(L1, L2)

L2 is obtained from L1 by duplicating all elements.

```
1 duplicate([], []).
2
3 duplicate([H | T2], [H, H | T2]) :-
4     duplicate(H, T2).
```

split(L, N, L1, L2)

List L1 contains the first N elements of the list L, the list L2 contains the remaining elements.

```
1 split(L, 0, [], L).
2
3 split([H | T1], N, [H | T2], T3) :-
4     N > 0,
5     N1 is N - 1,
6     split(T1, N1, T2, T3).
```

slice(L1, I, K, L2)

L2 is the list of the elements of L1 between index I and index K.

```

```

```
slice([X | _], 1, 1, [X]).

slice([X | Xs], 1, K, [X | Ys]) :-
    K > 1,
    K1 is K - 1,
    slice(Xs, 1, K1, Ys).

slice([_ | Xs], I, K, Ys) :-
    I > 1,
    I1 is I - 1,
    K1 is K - 1,
    slice(Xs, I1, K1, Ys).
```

remove_at(X, L, K, R)

X is the K'th element of the list L; R is the list that remains when K'th element is removed from L.

```
remove_at(X, [X | Xs], 1, Xs).

remove_at(X, [Y | Xs], K, [Y | Ys]) :-
    K > 1,
    K1 is K - 1,
    remove_at(X, Xs, K1, Ys).
```

range(I, K, L)

I <= K and L is the list containing all consecutive integers from I to K.

```
range(I, I, [I]).

range(I, K, [I | L]) :-
    I < K,
    I1 is I - 1,
    range(I1, K, L).
```

Programiranje z omejitvami

Faktoriela

Predikat `faktoriela(N, F)` sprejme N, ki pomeni število ter F, ki je rezultat (fakulteta).

```
1 faktoriela(0, 1).
2 faktoriela(N, F) :-
3     N > 0,
4     M is N - 1,
5     faktoriela(M, G),
6     F is N * G.
```

```
?- faktoriela(7, F).
```

```
1 :- use_module(library(clpfd)).
2
3 fakulteta(0, 1).
4 fakulteta(N, F) :-
5     N #> 0,
6     M #= N - 1,
7     F #= N * G,
8     fakulteta(M, G).
```

```
?- fakulteta(X, 24).
```

Pitagorejske trojice

Predikat `pitagora(A, B, C)` je trojica calih števil, za katere velja A^2 = B^2 + C^2. Predpostavimo, da je A ≤ B.

```
:- use_module(library(clpfd)).

pitagora(A, B, C) :-
    A #> 0,
    B #< C,
    A #=< B,
    A * A + B * B #= C * C.
```

```
?- pitagora(A, B, 50), A #< 20.
```

```
:- use_module(library(clpfd)).

pitagora(A, B, C) :-
    A #> 0,
    B #< C,
    A #=< B,
    A * A + B * B #= C * C.

pitagora_do([A, B, C], N) :-
    pitagora(A, B, C),
    [A, B, C] ins 1..N,
    label([A, B, C]).
```

```
?- pitagora_do([X, Y, Z], 10).
```

Permutacije

```
:- use_module(library(clpfd)).

permutacija(N, P) :-
    length(P, N), % P je dolžine N
    P ins 1..N,
    all_distinct(P). % vsi med seboj različni
```

```
?- permutacija(3, P), label(P).
```

Naloga "klekljarice"

Na rednem tedenskem srečanju se je zbralo *X* klekljaric. Nekatere so s sabo pripeljale tudi svoje mačke; teh je bilo skupaj *Y*. Vemo, da je na srečanju bilo 22 glav in 72 nog. Napiši predikat `meeting(X, Y)`, ki postavi ustrezne omejitve in vrne število klekljaric in mačk, ki so bile na srečanju.

```
meeting(X, Y) :-
    [X, Y] ins 1..sup,
    X + Y #= 22,
    X * 2 + Y * 4 #= 72,
    label([X, Y]).
```

```
?- meeting(X, Y).
```

Jedilnik

```
predjed(goveja_juha).
glavna_jed(polnjene_bucke).
priloga(polenta).
sladica(tortica).
sadje(jabolko).

cena(polenta, 81). kalorije(polenta, 275). beljakovine(polenta, 26).
vege(polenta).
```

```
?- cena(polenta, X).
```

Kosilo

Kosilo je sestavljeno iz 1) glavne jedi in priloge ter 2) dveh dodatkov, ki jih izberemo izmed predjedi, sadja in sladice.

Napišite predikat `kosilo(Seznam)`, ki vrača različne kombinacije štirih jedi, ki sestavljajo kosilo.

```
kosilo([GlavnaJed, Priloga, Dodatek1, Dodatek2]) :-
    glavna_jed(GlavnaJed),
    priloga(Priloga),
    (predjed(Dodatek1), sadje(Dodatek2));
    (predjed(Dodatek1), sladica(Dodatek2));
    (sadje(Dodatek1), sladica(Dodatek2)).
```

```
?- kosilo(K).
```

Vegeterijansko kosilo

Predikat `vege_kosilo(K)`, ki preveri, ali kosilo K vsebuje samo vegetarijanske jedi.

```
vege_kosilo(K) :-
    maplist(vege, K)
```

```
vege_kosilo1([]).
vege_kosilo([H | T]) :-
    vege(H),
    vege_kosilo(T).
```

Ustrezno kosilo

Napišite predikat `ustrezno_kosilo(Kosilo, MaxCena, MinKalorij, MaxKalorij, MinBeljakovin, MinOH)` , ki preveri, ali kosilo K ustreza naslednjim pogojem:

· cenejšeodMaxCena,

· vsaj MinKalorij in največ MaxKalorij

· vsajMinBeljakovin ter

· vsaj MinOH ogljikovih hidratov.

Z ustrezno poizvedbo poiščite različna kosila, ki stanejo največ 5.00 € in vsebujejo vsaj 900 kalorij ter vsaj 50 g beljakovin.

```
ustrezno_kosilo(Kosilo, MaxCena, MiKalorij, MaxKalorij, MinBeljakovi
    maplist(cena, Kosilo, Cena),
    maplist(kalorije, Kosilo, Kalorije),
    maplist(beljakovine, Kosilo, Beljakovine),
    maplist(ogljikovi_hidrati, Kosilo, OgljikoviHidrati),
    sum(Cena, #=<, MaxCena),
    sum(Kalorije, #>=, MinKalorij),
    sum(Kalorije, #=<, MaxKalorij),
    sum(Beljakovine, #>=, MinBeljakovin),
    sum(Ogljikovi_hidrati, #>=, MinOH).
```

Problem `n` dam

Napisali bomo program, ki na šahovsko ploščo velikosti `n×n  razporedi  n` dam tako, da se med sabo ne napadajo (torej noben par dam ni na isti horizontali, vertikali ali diagonali). Koordinate posamezne figure bomo zapisali v obliki `X/Y` .

`safe_pair(X1/Y1, X2/Y2)`

Napišite predikat `safe_pair(X1/Y1, X2/Y2)` , ki z ustreznimi omejitvami zagotovi, da se dami na poljih `X1/Y1` in `X2/Y2` med sabo ne napadata. Primer:

```
?- safe_pair(1/1, 2/2).
false.
?- safe_pair(4/2, 5/3).
false.
?- safe_pair(3/3, 4/2).
false.
?- safe_pair(1/1, 2/3).
true.
```

```
1 safe_pair(X1/Y1, X2/Y2) :-
2     X1 #\= X2,
3     Y1 #\= Y2,
4     abs(X1 - X2) #\= abs(Y1 - Y2).
```

```
?- safe_pair(1/1, 2/2).
```

```
?- safe_pair(4/2, 5/3).
```

```
?- safe_pair(3/3, 4/2).
```

```
?- safe_pair(1/1, 2/3).
```

safe_list/2

Napišite predikat `safe_list(X/Y, L)`, ki sprejme koordinate ene dame in seznam koordinat preostalih dam ter preveri, da dama na koordinatah `X/Y` ne napada nobene v seznamu. Primer:

```
?- safe_list(1/1, [3/4, 3/8, 2/3, 3/5]).
true.
```

```
1 safe_list(X/Y, L) :-
2     maplist(safe_pair(X/Y), L).
3
4 safe_list(_, []).
5 safe_list(X/Y, [X1/Y1 | L]) :-
6     safe_pair(X/Y, X1/Y1),
7     safe_list(X/Y, L).
```

```
?- safe_list(1/1, [3/4, 3/8, 2/3, 3/5]).
```

safe_list/1

Napišite predikat `safe_list(L)`, ki sprejme seznam koordinat in preveri, da se med sabo ne napada noben par dam v seznamu. Primer:

```
?- safe_list([3/3, 2/5, 1/7, 8/6, 5/4, 7/8]).
true.
```

```
1 safe_list([]).
2
3 safe_list([X/Y | L]) :-
4     safe_list(X/Y, L),
5     safe_list(L).
```

```
?- safe_list([3/3, 2/5, 1/7, 8/6, 5/4, 7/8]).
```

place_queens/3

Napišite predikat `place_queens(N, I, L)`, ki na šahovnico velikosti `N×N razporedi N` dam tako, da je vsaka v svojem stolpcu (ima svojo koordinato *X*). Predikat naj vrne koordinate v seznamu *L*. *I* je pomožen parameter, ki hrani trenutno vrednost koordinate *X*. Primer:

```
?- place_queens(2, 0, L), term_variables(L, Vars), label(Vars).
L = [1/1, 2/1],
X = [1, 1] ;
L = [1/1, 2/2],
X = [1, 2] ;
L = [1/2, 2/1],
X = [2, 1] ;
L = [1/2, 2/2],
X = [2, 2] ;
false.
```

Predikat `term_variables(L, Vars)` poišče vse spremenljivke v izrazu (seznamu) *L* in jih shrani v seznam *Vars*, na katerem lahko potem pokličemo label/1 (/pldoc/man?predicate=label/1). Predikat place_queens/2 poskusite napisati tako, da se ustavi, ko vrne vse rešitve.

```
place_queens(N, I, []) :-
    I #> N.

place_queens(N, I, [I/Y | L]) :-
    I #=< N,
    Y ins 1..N,
    I_next #= I + 1,
    place_queens(N, I_next, L).
```

```
?- place_queens(2, 0, L), term_variables(L, Vars), label(Vars).
```

queens/2

Napišite predikat `queens(N, L)`, ki na šahovnico velikosti `N×N razporedi N` dam tako, da se med sabo ne napadajo. Njihove koordinate naj vrne v seznamu *L*. Kakšen je najmanjši *N*, za katerega obstaja vsaj ena rešitev?

```
queens(N, L) :-
    place_queens(N, 1, L),
    safe_list(L),
    term_variables(L, Vars),
    label(Vars).
```

```
?- queens(4, L).
```

Za lepši izpis pozicij kraljic:

```
:- use_rendering(chess).
queens2(N, L2) :-
    queens(N, L),
    maplist(arg(2), L, L2).
```

```
?- queens2(4, L).
```

`getdigits(L, DL)`

Seznam DL vsebuje vse številke iz seznama L, vrstni red se ohrani.

```
1 getdigits([H | T], [H | DL]) :-
2     getdigits(T, DL),
3     number(H).
4
5 getdigits([H | T], DL) :-
6     getdigits(T, DL),
7     \+ number(H).
```

```
?- getdigits([2,3,e,-,4,b], DL).
```

Izpiti

```
1 :- use_module(library(clpfd)).
2 abeceda([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z]).
3 A = [a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z], B = [d,e,
```

Naloga 1

V Prologu definiramo predikat a/0.

```
1 a :-
2     b,
3     c;
4     d.
5 a :-
6     e.
```

Katera logična formula je ekvivalenta temu zapisu?

((b∧c∨d)⇒a)∧(e⇒a)

Naloga 2

V Prologu sestavimo naslednji predikat:

```
1 h([], []).
2 h([[] | Y], Z) :-
3     h(Y, Z).
4 h([[X | _] | Y], [X | Z]) :-
5     h(Y, Z).
```

Kaj pomeni `h(Y, Z)` ?

Y je seznam prvih elementov vseh nepraznih seznamov seznama X.

Naloga 3

Predikat cesta/2 pove, kateri izmed krajev a, b, ..., h so neposredno povezani s cesto:

```
cesta(a, b).
cesta(a, e).
cesta(a, c).
cesta(b, d).
cesta(d, e).
cesta(e, f).
cesta(g, h).

visina(a, 10).
visina(b, 7).
visina(c, 12).
visina(d, 4).
visina(e, 6).
visina(f, 7).
visina(g, 0).
visina(h, 9).
```

Ceste so dvosmerne, torej iz a v b vodi cesta, po kateri lahko potujemo v obe smeri.

Sestavite predikat `povezava(X,Y)` , ki velja natanko tedaj, ko obstaja neposredna cesta od X do Y ali od Y do X.

Create a Program | Query | Markdown | HTML **cell here**

```
povezava(X, Y) :-
    cesta(X, Y);
    cesta(Y, X).
```

Sestavite predikat `pot(X,Y)` , ki velja natanko tedaj, ko obstaja pot med X in Y.

```
pot(X, X).

pot(X, Y) :-
    povezava(X, Y).

pot(X, Y) :-
    povezava(X, Z);
    pot(Z, Y).
```

```
?- pot(a,a).
true .
?- pot(f, a).
true.
```

Zapišite predikat `pot(X,Y,P)` , ki velja natanko tedaj, ko je P pot od X do Y.

```
pot(X, Y, [X | P]) :-
    povezava(X, Z),
    pot(Z, Y, P).

pot(X, Y, [X | P]) :-
    povezava(X, Z),
    pot(Z, Y, P).
```

```
?- pot(a, f, P).
```

Sestavite predikat `spust(X,Y)` , ki velja natanko tedaj, ko obstaja pot od X do Y brez vzpenjanja.

```
spust(X, Y) :-
    povezava(X, Y),
    visina(X, V1),
    visina(Y, V2),
    V1 >= V2.
```

```
?- spust(a, f)
```

Naloga 4

Cezarjeva šifra je starodavni sistem šifriranja sporočil, pri katerem vsako črko čistopisa zamaknemo za k mest v abecedi (črke na koncu abecede se krožno zamaknejo na začetek abecede). Na primer, če je ključ k = 3, se v angleški abecedi beseda “zebra” šifrira kot “cheud”.

Dogovorimo se, da delamo z angleško abecedo in v ta namen definiramo predikat abeceda/1, ki določa vrstni red črk v angleški abecedi:
`abeceda([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z])` .

Besedo v prologu predstavimo s seznamom atomov, na primer[z,e,b,r,a].

Sestavite predikat rotiraj/3, kjer `rotiraj(K,A,B)` pomeni, da dobimo seznam B tako, da seznam A krožno zamaknemo za K mest.

Create a Program | Query | Markdown | HTML **cell here**

```
rotiraj(K, L, L) :-
    K #< 0.

rotiraj(K, [Head | Rest], B) :-
    K #> 0,
    K1 #= K - 1,
    append(Rest, [Head], S),
    rotiraj(K1, S, B).
```

```
?- rotiraj(2, [l,j], P).
```

Sestavite predikat preslikaj/4, kjer `preslikaj(A,B,X,Y)` pomeni, da v seznamu A poiščemo element X in vrnemo istoležni element Y v seznamu B. Predpostavite lahko, da sta seznama A in B podana, enako dolga in sestavljena iz različnih atomov.

```
1 preslikaj([X | _], [Y | _], X, Y).
2 preslikaj([_ | L1], [_ | L2], X, Y) :-
3     preslikaj(L1, L2, X, Y).
```

```
?- preslikaj([a,b,c], [c,a,b], c, Y).
```

Sestavite predikat cezar/3, kjer `cezar(K,In,Out)` pomeni, da dobimo Out, ko In šifriramo s Cezarjevo šifro z zamikom K.

```
1 cezar(K, In, Out) :-
2     abeceda(A),
3     rotiraj(K, A, B),
4     maplist(preslikaj(A, B), In, Out).
```

```
?- cezar(3, [z,e,b,r,a], Out).
```

Naloga 5

Predstavitev naravnih števil v eniškem sistemu: Atoma z in s predstavljata nič in operacijo naslednik. Na primer s(`s(s(s(z)))` predstavlja število 5.

Predikat n predstavlja relacijo "naslednik":

```
1 z. % nič
2 n(X) :-
3     X = z;
4     X = n(_). % če je X število, je število tudi njegov naslednik n(
```

Definirajte predikat vsota/3, kjer `vsota(X, Y, Z)` pomeni, da je Z vsota X in Y.

```
1 vsota(z, X, X).
2
3 vsota(n(X), Y, n(Z)) :-
4     vsota(X, Y, Z).
```

```
?-
```

```
vsota(n(n(z)), n(n(z)), S).
```

Definirajte predikat produkt/3, kjer `produkt(X, Y, Z)` pomeni, da je z zmnožek X in Y.

```
1  produkt(z, _, z).
2
3  produkt(n(A), Y, Z) :-
4      vsota(P1, Y, Z),
5      produkt(A, Y, P1).
```

Create a | Program | Query | Markdown | HTML | cell here

```
?- produkt(n(n(z)), n(n(n(z))), P).
```

Definirajte predikat stevilo/2, kjer `stevilo(N, X)` pomeni, da je N običajno celo število v prologu in X isto število predstavljeno v eniškem sistemu.

```
1  stevilo(0, z).
2
3  stevilo(N, n(X)) :-
4      N #= NX + 1,
5      stevilo(NX, X).
```

```
?- stevilo(2, Dva).
```

Zapišite vprašanje prologu, ki z uporabo (samo) zgornjih predikatov poišče skupne delitelje števil 252 in 294. Pri iskanju odgovorov se poizvedba lahko zacikla, vendar naj pred tem vrne vse rešitve.

```
1  stevilo(252, A), stevilo(294, B), produkt(D, _, A), produkt(D, _, B)
```

Naloga 6

Imamo tehnico in uteži. Da bi stehtali modro kroglo, smo dali nekaj uteži na tehnico, tako da je tehnica v ravnovesju.

Definirajmo `balance(L, R, B)` , ki velja, ko je B v ravnovesju, L predstavlja uteži na levi strani, R pa na desni. B predstavlja razliko vseh mas na levi in na desni.

```
1  balance(L, R, B) :-
2      sumlist(L, LS),
3      sumlist(R, RS),
4      B #= RS - LS.
```

```
?-  balance([], [], B).
```

```
?- balance([4], [1, 16], B).
```

```
1 balance([], [], 0).
2
3 balance([H | Rest], R, B) :-
4     balance(Rest, R, B1),
5     B #= B1 - H.
6
7 balance(L, [H | Rest], B) :-
8     balance(L, Rest, B1),
9     B #= B1 + H.
```

```
?- balance([4], [1, 16], B).
```

Definirajmo predikat `split(Ws, L, R)` , ki velja, ko uteži iz seznama L in R predstavljajo razporeditev na levi in na desni strani tehnice. Ws predstavlja seznam še neuporabljenih uteži. Vsako utež iz Ws lahko uporabimo le enkrat.

```
1 split([W | Ws], [W | L], R) :-
2     split(Ws, L, R).
3
4 split([W | Ws], L, [W | R]) :-
5     split(Ws, L, R).
6
7 split([_ | Ws], L, R) :-
8     split(Ws, L, R).
```

```
?- split([1,2,3], L, R)
```

Definirajmo predikat `measure(Ws, W)` , ki velja, ko je možno uporabiti uteži iz seznama Ws, da stehtamo predmet mase W.

```
1 measure(Ws, W) :-
2     split(Ws, L, R),
3     balance(L, R, W).
```

```
?- measure([1,3], W)
```

Definirajmo predikat `measure_interval(Ws, A, B)` , ki velja, ko je možno stehtati z uporabo uteži iz seznama Ws, mase med A in B.

```
measure_interval(_, A, B) :-
    A #> B.

measure_interval(Ws, A, B) :-
    once(measure(Ws, A)),
    A1 #= A + 1,
    measure_interval(Ws, A1, B).
```

```
?- measure_interval([1,3], 0, 4])
```

Napiši poizvedbo, ki najde 4 uteži z masami med 1 in 40, ki bodo sposobne izmeriti vse mase med v intervalu [0, 40].

```
?- length(Ws, 4), Ws ins 1.40, label(Ws), measure_interval(Ws, 0, 40).
```

Naloga 7 - Šahovnica

Polja na šahovnici velikosti 8 × 8 označimo s koordinatami (x, y), kjer velja 1 ≤ x ≤ 8 in 1 ≤ y ≤ 8.

Polje (X, Y) v prologu zapišemo z izrazom X/Y, saj prolog nima urejenih parov.

Sestavite predikat `polje(X/Y)` , ki velja natanko tedaj, ko sta X in Y veljavni koordinati. Poskrbite, da poizvedba `polje(X/Y)` vrne vseh 64 odgovorov. Uporabiti smete programiranje z omejitvami ali kak drug pristop.

```
polje(X/Y) :-
    X #< 9,
    X #> 0,
    Y #< 9,
    Y #> 0.
```

```
?- polje(8/2).
```

```
polje(X/Y) :-
    member(X, [1, 2, 3, 4, 5, 6, 7, 8]),
    member(Y, [1, 2, 3, 4, 5, 6, 7, 8]).
```

```
?- polje(8/2).
```

```
1 polje(X/Y) :-
2     [X, Y] ins 1..8,
3     label([X, Y]).
```

```
?- polje(8/2).
```

Sestavite predikat `premik(P,Q)` , ki velja natanko tedaj, ko sta P in Q veljavni polji in se lahko skakač premakne s polja P na polje Q.

```
1 premik(P, Q) :-
2     polje(P),
3     polje(Q),
4     P = X1/Y1,
5     Q = X2/Y2,
6     (
7         (abs(X1 - X2) #= 2, abs(Y1 - Y2) #= 1);
8         (abs(X1 - X2) #= 1, abs(Y1 - Y2) #= 2)
9     ).
```

```
?- premik(1/1, 3/3).
```

Sestavite predikat `sprehod(L)` , ki velja natanko tedaj, ko je L seznam veljavnih polj in za vsaki zaporedni polji P in Q v seznamu L velja `premik(P,Q)` . Polja v sprehodu se smejo ponavljati.

```
1 sprehod([]).
2
3 sprehod([P]) :-
4     polje(P).
5
6 sprehod([H1, H2 | T]) :-
7     premik(H1, H2),
8     sprehod([H2 | T]).
```

Zapišite poizvedbo, ki ugotovi, ali obstaja sprehod dolžine 64 od polja 1/1 do polja 8/8.

```
?- lenght(64, Sprehod), Sprehod = [1/1 | _], append(_, [8/8],
   Sprehod), sprehod(Sprehod).
```

Naloga 8 - Nakupi

```
1  bought(eva, bread, 1). % eva je kupila 1 kg kruha
2  bought(tomi, beer, 6).
3  bought(eva, butter, 2).
4  bought(tina, beer, 2).
5  bought(tina, salami, 1).
6
7  price(beer, 2).
8  price(apples, 1).
9  price(bread, 3).
10 price(salami, 17).
```

Za gornjo bazo nakupov podaj vse prologove odgovore na vprašanje:

```
?- ?- bought(X, beer, Qty), price(beer, P), Value #= Qty * P.
```

X = tomi, Qty = 6, P = 1.9, Value = 12 ;

X = tina, Qty = 2, P = 1.9, Value = 4.

Zapiši vprašanje za prolog, ki poišče osebo X, ki je kupila vsaj dva različna proizvoda.

```
?- bought(X, P, _), bought(X, R, _), P \== R.
```

Naloga 9 - Vsi enaki

```
1  all_equal([]).
2
3  all_equal([_]).
4
5  all_equal([H, H | T]) :-
6     all_equal([H | T]).
```

```
?- all_equal([a,a,a,a]).
```

Naloga 10 - Telefon

Na telefonski številčnici pišemo besede tako, da za vsako črko nekajkrat pritisnemo ustrezno številko. Tipke, ki jih moramo pritisniti za posamezne črke, so podane s predikatom letter/2:

```
letter(a, [2]). % za črko 'a' moramo enkrat pritisniti 2
letter(b, [2,2]). % za črko 'b' moramo dvakrat pritisniti 2
letter(t, [8]).
letter(u, [8,8]).
letter(v, [8,8,8]).
letter(z, [9,9,9,9]).
```

Napiši predikat spell/2, ki za dan seznam črk vrne zaporedje (seznam) tipk, ki jih moramo pritisniti za zapis teh črk.

```
spell([], []).

spell([H | T], L) :-
    letter(H, Hs),
    spell(T, LT),
    append(Hs, LT, L).
```

```
?- spell([a,v,t,o], L).
```

Naloga 11 - Črkovanje

Podana sta predikata letter/1 in word/1 (/pldoc/doc/scratch/swish/src/swish/pack/chat80/prolog/chat80/newdic.pl#word/1), ki vračata veljavne črke oziroma besede v jeziku. Posamezna beseda je predstavljena kot seznam črk.

Primer: `letter(L) :- member(L, [a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z]). word([c,h,a,i,r]). word([c,h,e,s,s]). word([d,e,s,k]). word([c,h,a,l,k]). word([c,h,o,p]). word([d,i,s,k]).`

Napiši predikat `edit(Word, New)`, ki vhodno besedo Word »popravi« v besedo New na enega od treh načinov: izbriše poljubno črko, vstavi novo črko na poljubno mesto, ali pa eno črko na poljubnem mestu zamenja z novo.

Predikat naj generira vse možne rešitve.

```
edit(Word, New) :-
    insert(_, New, Word).

edit(Word, New) :-
    letter(L),
    insert(L, Word, New).

edit(Word, New) :-
    letter(L),
    append(A, [_ | B], Word),
    append(A, [L | B], New).
```

Naloga 12

Dan je naslednji program z omejitvami CLP(FD):

```
prog(A, B) :-
    length(B, A),
    B ins 1..A,
    all_different(B),
    label(B).
```

Kaj odgovori prolog na spodnje vprašanje?

```
?- prog(3, B).
```

Naloga 13

Podan je spodnji predikat, ki uporablja CLP(FD).

```
s([], 0).
s([X|L], S) :-
    S #= S1 + X,
    s(L, S1).
```

```
?- s([3,2,4], S)
```

Naloga 14

Prva in zadnja vrednost v vsaki vrstici sta 1, vsak drug element pa dobimo tako, da seštejemo dve vrednosti nad njim.

Napišite predikat `pascal(I, J, N)` , ki vrne J-to vrednost v I-ti vrstici Pascalovega trikotnika. Argumenta I in J naj začneta šteti z 0; predpostavite lahko, da velja $0 \leq J \leq I$). Predikat lahko isto (pravilno) rešitev vrne večkrat.

```
1  pascal(_, 0, 1).
2
3  pascal(I, J, 1) :-
4      I #=> 1,
5      J #= I.
6
7  pascal(I, J, N) :-
8      I #> 1,
9      I1 #= I - 1,
10     J1 #= J - 1,
11     pascal(I1, J, N1),
12     pascal(I1, J1, N2),
13     N #= N1 + N2.
```

Create a Program | Query | Markdown | HTML **cell here**

```
?- pascal(0, 0, N).
```

V OCamlu definiramo podatkovni tipi number, s kateri predstavimo cela števila:

`type number = Zero | Succ of integer | Pred of integer`

Vrednost `Zero` predstavlja število 0, `Succ n` naslednik `n` ter `Pred n` predhodnik `n`. Vsako število lahko predstavimo na več načinov. Na primer, število 0 je predstavljeno z vrednostmi

```
Zero
Pred (Succ Zero)
Succ (Pred Zero)
Pred (Pred (Succ (Succ Zero)))
Pred (Succ (Succ (Pred Zero)))
...
```

Med vsemi je najbolj "ekonomična" predstavitev seveda `Zero`, ker ne vsebuje nepotrebnih konstruktorjev.

Sestavite funkcijo `simp : number -> number`, ki dano predstavitev pretvori v najbolj ekonomično, se pravi tako, ki ima najmanjše možno število konstruktorjev.

Primeri:

```
# simp (Pred (Succ (Succ (Pred (Pred (Succ (Pred Zero)))))));;
- : number = Pred Zero
# simp (Succ Zero);;
- : number = Succ Zero
```

```
let rec simp = function
      | Zero -> Zero
      | Pred a -> (match simp a with
            | Succ a -> a
            | a -> Pred a)
      | Succ a -> (match simp a with
            | Pred a -> a
            | a -> succ a)
```

Naloga

Andrej je v OCamlu definiral podatkovni tip

```
type 'a trie = Node of 'a | Trie of ('a trie) list
```

- zapišite vrednost tipa `int trie`: `Node 1`
- zapišite vrednost tipa `'a trie`: `Trie []`
- zapišite vrednost, ki je različa od prejšnjih dveh: `Node 3` ali `Trie [Trie [Node 1]; Node 2]`

`Node of 'a` ... imamo nek konstruktor `Node`

`'a trie` ... konstuktorja `Node` ne smemo več uporabiti, ker mora biti tip polimorfen: `Trie []`

Ne smemo uporabiti konstruktor `Node`, ker potem ni polimorfnega tipa.

V OCamlu definiramo podatkovni tipi number, s kateri predstavimo cela števila:

```
type number = Zero | Succ of integer | Pred of integer
```

Vrednost `Zero` predstavlja število 0, `Succ n` naslednik `n` ter `Pred n` predhodnik `n`. Vsako število lahko predstavimo na več načinov. Na primer, število 0 je predstavljeno z vrednostmi

```
Zero
Pred (Succ Zero)
Succ (Pred Zero)
Pred (Pred (Succ (Succ Zero)))
Pred (Succ (Succ (Pred Zero)))
...
```

Med vsemi je najbolj "ekonomična" predstavitev seveda `Zero`, ker ne vsebuje nepotrebnih konstruktorjev.

Enako predstavitev celih števil uporabimo tudi v Prologu, le da moramo konstruktorje pisati z malo začetnico. Na primer, število 3 predstavimo z izrazom

`succ(pred(succ(succ(succ(pred(succ(zero)))))))`,

ki pa ni najbolj ekonomičen. Dopolnite spodnji predikat `simp(A,B)`, ki velja, ko je `B` najbolj ekonomična predstavitev `A`.

Primer uporabe:

```
?- simp(succ(pred(succ(succ(succ(pred(succ(zero))))))), B).
B = succ(succ(succ(zero))) ;
false.
```

```
simp( _____, _____).
simp(succ(A), _____) :-
      simp(A, pred(C)).
simp(succ(A), _____) :-
      simp(A, zero).
simp(succ(A), _____) :-
      simp(A, succ(C)).
simp(pred(A), _____) :-
      simp(A, _____).
simp(pred(A), _____) :-
      simp(A, _____).
simp(pred(A),_____) :-
      simp(A, _____).
```

```
next -> succ
prev -> pred

simp(zero, zero).
simp(succ(A), C) :-
     simp(A, pred(C)).
simp(succ(A), next(zero)) :-
     simp(A, zero).
simp(succ(A), next(next(C))) :-
     simp(A, succ(C)).
simp(pred(A), C) :-
     simp(A, next(C)).
simp(pred(A), prev(zero)) :-
     simp(A, zero).
simp(pred(A), prev(prev(C))) :-
     simp(A, prev(C)).
```