Principi programskih jezikov

Andrej Bauer

Contents

1. O programskih jezikih in aritmetičnih izrazih

V prvi lekciji bomo spoznali osnovni ustroj programskega jezika na primeru aritmetičnih izrazov. Ti pravzaprav niso samostojen splošen programski jezik, a lahko kljub temu z njimi ponazorimo osnovni pristop. Še prej pa povejmo nekaj besed o namenu predmeta.

1.1 O programskih jezikih

Programski jeziki so eno od glavnih orodij v računalništvu. Poznamo jih na tisoče, a samo peščica je takih, ki jih uporablja veliko število programerjev. Pri tem predmetu se bomo učili **osnovne principe**, ki so podlaga za načrtovanje, implementacijo in delovanje programskih jezikov. S tem nimamo v mislih prevajalnikov, strojne kode ipd. (to snov pokrivajo drugi predmeti), ampak **matematične koncepte**, ki jih srečamo v programskih jezikih.

Osnovno vodilo načrtovanja programskega jezika je:

> **Osnovni princip**
>
> Programski jezik je orodje, ki programerju omogoča, da na čim bolj neposreden način poda natančna navodila, kako naj računalnik opravi neko nalogo.

Človek bi si mislil, da bi do zdaj že lahko iznašli »najboljši programski jezik«, a v resnici jih je na tisoče. Zakaj?

1. Ker se programski jeziki sproti prilagajajo razvoju računalniške tehnologije in potrebam programerjev.
2. Ker se razvijajo novi programerski koncepti in tehnike.
3. Ker niso vsi stili programiranja enako primerni za reševanje vseh nalog.
4. Ker nekateri radi vsak mesec naredijo nov programski jezik.

1.2 Anatomija programskega jezika

Programski jezik je zasnovan kot sistem, ki ima naslednje komponente:

- **sintaksa:** pravila, kako se piše kodo, na primer: »vsak oklepaj mora imeti svoj zaklepaj«
- **statična semantika:** preverjanje, ali je program smiseln, na primer: »spremenljivka `i` ni nikjer deklarirana«
- **dinamična semantika:** kako se program izvede
- **denotacijska semantika:** matematični pomen programa

Programski jezik nima nujno vseh teh komponent, čeprav vsaj sintakso in dinamično semantiko vedno imamo. Opis jezika je lahko:

- **neformalen** dokument, napisan v naravnem jeziku, običajno zelo obsežen (C++ (cena: CHF 198), Java, Racket, OCaml, Python, Haskell) ali
- **formalne**: podana je matematična definicija (Definition of Standard ML).

Pogosto je definicija jezika kombinacija obeh pristopov. **Implementacija** jezika je program, ki preverja sintakso in statično semantiko jezika ter omogoča izvajanje programov. To je lahko tolmač (angl. interpreter), prevajalnik (angl. compiler), oboje, ali pa kombinacija (glej just-in-time compilation).

Pomemben del programskega jezika so tudi metode za **analizo programov**, s katerimi ugotavljamo lastnosti programa, in za **dokazovanje pravilnosti**, s katerimi dokazujemo, da ima program želene lastnosti.

1.3 Sintaksa aritmetičnih izrazov

Začeli bomo z zelo preprostim programskim jezikom, ki je tako preprost, da ga v praksi sploh ne obravnavamo kot samostojen programski jezik. Obravnavajmo **celoštevilske aritmetične izraze**: cela števila, operaciji * in + ter spremenljivkami. To bi lahko bil majhen košček resnega programskega jezika.

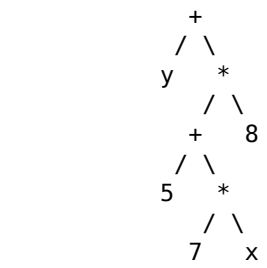
Sintaksa pove, kakšne izraze in programe lahko pišemo v programskem jeziku.

1.3.1 Konkretna sintaksa aritmetičnih izrazov

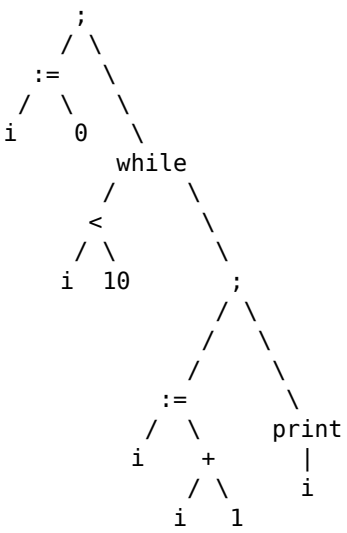
Programer zapiše program kot niz znakov, na primer:

```
"y + (5 + 7 * x) * 8"
```

Ta oblika je primerna za človeka, a ni primerna za obdelavo z računalnikom, saj ne odraža strukture izraza. Zgornji izraz predstavimo z drevesom takole:



Na ta način se znebimo presledkov in oklepajev in jasno ponazorimo strukturo izraza. Tudi programsko kodo lahko predstavimo z drevesi. Ali znate razbrati program, ki ga predstavlja naslednje drevo?



1.3.2 Abstraktna sintaksa aritmetičnih izrazov

Izraze lahko opišemo s podatkovno strukturo drevo. To je oblika, primerna za obdelavo, ni pa primerna za človeka.

Prednosti:

1. Iz drevesa je takoj razvidna struktura programa ali izraza.
2. Drevo ne vsebuje nepomembnih komponent (na primer presledkov in oklepajev).

Kako implementiramo drevesa, je odvisno od programskega jezika, ki ga uporabimo. V Javi se to seveda naredi z razredi. Kasneje bomo spoznali še druge načine.

1.3.3 Pravila sintakse

Pravila, ki opisujejo, kako tvorimo izraze ali drevesa, se imenujejo **pravila sintakse** (angl. syntax rules). Poznamo več načinov, kako podamo pravila, mi si bomo ogledali poenostavljeno verzijo t.i. oblike BNF, ki jo pogosto srečamo v praksi:

```
⟨izraz⟩ ::= ⟨aditivni-izraz⟩ EOF
⟨aditivni-izraz⟩ ::= ⟨multiplikativni-izraz⟩ | ⟨aditivni-izraz⟩ +
⟨multiplikativni-izraz⟩
⟨multiplikativni-izraz⟩ ::= ⟨osnovni-izraz⟩ | ⟨multiplikativni-izraz⟩ *
⟨osnovni-izraz⟩
```

```
⟨osnovni-izraz⟩ ::= ⟨spremenljivka⟩ | ⟨številka⟩ | ( ⟨aditivni-izraz⟩ )
⟨spremenljivka⟩ ::= [a-zA-Z]+
⟨številka⟩ ::= -?[0-9]+
```

V trikotnih oklepajih `⟨…⟩` so zapisani **neterminalni simboli**. Vsak od njih ima svoje pravilo, ki pove, kako ga razčlenimo. Ostali simboli (`+`, `*`, `(`, `)`, in regularni izrazi, ki opisujejo spremenljivke in številke) so **osnovni** ali **terminalni simboli** (v teoriji formalnih jezikov razlikujemo med tema dvema pojmoma, a se mi s tem be bomo obremenjevali).

Pri opisu spremenljivk in številke smo uporabili **regularne izraze**: v oglatih oklepajih navedemo, kateri znaki so dovoljeni, znak `+` pa pomeni »ena ali več ponovitev«.

Glede na pravila, je dani izraz veljaven ali ne. Primeri:

- izraz `x * (5 + 8)` je veljaven
- izraz `x * + 5` je neveljaven
- izraz `1 * 2 + x` je veljaven

Ali bi lahko `1 * 2 + x` razčlenili kot »enica pomnožena z vsoto dvojke in `x`, se pravi `1 * (2 + x)`, glede na zgornja pravila?

1.3.4 Iz konkretne v abstraktno sintakso

Konkretno sintakso predelamo v abstraktno sintakso s postopkomaa leksikalne in sintaksne analize:

- **leksikalna analiza:** niz razbijemoo na zaporedje podnizov, ki jih imenujemo **leksikalne enote** (angl. **lexemes**). Posamezne podnize predstavimo z
- **sintaksna analiza** (angl. parsing): niz osnovnih simbolov razčlenimo v sintaksno drevo

Leksikalna analiza odstrani nebistvene znake, kot so presledki in prehodi v novo vrsto, pogosto tudi komentarje.

Za aritmetične izraze so leksikalni elementi in pripadajoči osnovni simboli:

- niz `+` in simbol `PLUS`
- niz `*` in simbol `KRAT`
- spremenljivka, opisana z regularnim izrazom `[a-zA-Z]+` in simbol `SPREMENLJIVKA(x)`, kjer je `x` niz
- številka, opisana z regularnim izrazom `-?[0-9]+` in simbol `ŠTEVILKA(n)`, kjer je `n` število
- niz `(` in simbol `OKLEPAJ`
- niz `)` in simbol `ZAKLEPAJ`
- `EOF` poseben gradnik, ki pomeni »konec«

Primer: `foo * (5 + 42)` nam da leksikalnih elementov

```
"foo", "*", "(", "5", "+", "42", ")"
```

s pripadajočim nizom osnovnih simbolov

```
SPREMENLJIVKA("foo") KRAT OKLEPAJ ŠTEVILKA(5) PLUS ŠTEVILKA(42) ZAKLEPAJ EOF
```

Ta niz nam da ustrezno drevo.

Primer: `x * ((5 + 8` nam da niz leksikalnih elementov

```
"x", "*", "(", "(", "5", "+", "8"
```

s pripadajočim nizom osnovnih simbolov

```
SPREMENLJIVKA(x) KRAT OKLEPAJ OKLEPAJ ŠTEVILKA(5) PLUS ŠTEVILKA(8)
```

Ta niz ni veljaven in ne določa drevesa. Javimo sintaktično napako.

Zakaj ločimo med leksikalnim elementom in osnovnim simbolom? Na primer, zakaj je treba imeti * in `KRAT`? Iz vsaj dveh razlogov:

1. Morda želimo imeti več različnih znakov za množenje *, × in ·, ki jih vse predstavimo z istim osnovnim simbolom, saj vsi predstavljajo isto enoto v abstraktni sintaksi
2. Osnovne simbole naštejemo (na primer kot algebraični tip, to bomo še spoznali) v kodi, da prevajalnik točno ve, kateri simboli se lahko pojavijo. Če bi uporabljali nize, nas prevajalnik ne more opozoriti na morebitne tipkarske napake v kodi. Konkretno, če preverjamo `if (simbol == KART) then ...`, bo prevajalnik javil napako, saj ne ve, kaj je `KART`. Če bi preverjali neposredno nize z `if (simbol = "**") then ...`, prevajalnik ne bi vedel, da smo se zatipkali in da bi moralo pisati *.

Na vajah boste spoznali **sitnaksni analizator** (angl. parser) za aritmetične izraze, implementiran v Javi. Običajno pa razčlenjevalnika ne implementiramo z golimi rokami, ker je to precej zamudno, ampak uporabimo **sintaksni generator** - program, ki sprejme slovnična pravila in iz njih naredi sintaksni analizator (primer takega opisa za aritmetične izraze v OCamlu).

1.3.5 Iz abstraktne v konkretno sintakso

Pretvorba abstraktne sintakse v konkretno je preprosta, saj le obidemo sintaktično drevo in zgradimo ustrezni niz. Pojavi se vprašanje, kako vstaviti oklepaje, na katerega boste odgovorili na vajah.

1.4 Operacijska semantika

Kakšen je postopek, s katerim izračunamo vrednost izraza? Podali bomo dva osnovna načina.

Ko računamo vrednost izraza, moramo poznati vrednosti spremenljivk. Preslikavi, ki spremenljivke slika v njihove vrednosti, pravimo **okolje** (angl. environment). Na primer, če ima `x` vrednost `3`, `y` vrednost `7` in `z` vrednost `10`, to predstavimo z okoljem

```
[x ↦ 3, y ↦ 7, z ↦ 10]
```

1.4.1 Semantika velikih korakov

Semantika velikih korakov se imenuje tako, ker iz izraza (abstraktnega drevesa) dobimo njegovo vrednost (število) v enem »velikem« koraku. Predstavimo jo z relacijo

```
η | e ↪ n
```

kjer je `η` okolje, `e` je izraz in `n` celo število. Zgornji izraz preberemo takole: »V okolju `η` se izraz `e` evalvira v število `n`.«

Na primer, pričakujemo, da velja

```
[x ↦ 3, y ↦ 2, z ↦ 5] | x + 2 * y ↪ 7
```

Pravila za računanje izrazov podamo kot **pravila sklepanja**. Pravilo sklepanja zapišemo takole:

```
P₁ P₂ ⋯ Pᵢ
----------
    S
```

To preberemo: Če smo že dokazali predpostavke `P₁, P₂, ⋯, Pᵢ` , potem sledi tudi sklep `S`.

Na primer:

```
x > 0      y < 0
---------------
    x · y < 0
```

Preberemo: »če je `x` pozitiven in `y` negativen, potem je `x · y` negativen.«

Lahko se zgodi, da pravilo nima predpostavk:

```
--------
   S
```

Takemu pravilu pravimo tudi **aksiom**, saj pove da S velja. Primer aksioma je zakon refleksivnosti za enakost:

```
-----
x = x
```

Podajmo pravila za semantiko velikih korakov, pri čemer uporabimo oznake:

- η je okolje
- n je število
- e, e_1, ... so izrazi

Pravila:

```
η(x) = n
----------
 η | x ↪ n

---------
η | n ↪ n

 η | e₁ ↪ n₁     η | e₂ ↪ n₂    n₁ · n₂ = n
-----------------------------------------
             η | e₁ * e₂ ↪ n

 η | e₁ ↪ n₁     η | e₂ ↪ n₂    n₁ + n₂ = n
-----------------------------------------
             η | e₁ + e₂ ↪ n
```

Pozor, v pravilu za seštevanje znak + nad črto pomeni matematično operacijo seštevanje, pod črto pa je to del sintakse aritmetičnih izrazov, se pravi + je samo simbol v izrazu. Pri pravilu za množenje te težave nismo imeli, ker smo matematično množenje označili z ·, množenje kot simbol pa z *.

1.4.2 Semantika malih korakov

Semantika velikih korakov deluje hierarhično: najprej izračunamo vrednosti podizrazov in nato vrednost celotnega izraza. V šoli pa otroke učimo, da se računa »po korakih«, se pravi, da opravimo eno operacijo naenkrat. Tak postopek se imenuje **semantika malih korakov**. Podamo jo z relacijo (pozor, puščico ↪ smo spremenili v puščico ↦)

```
η | e ↦ e'
```

ki pove, kako naredimo en osnovni korak v računanju. Pravila se glasijo:

```
 η(x) = n
----------
η | x ↦ n


      η | e₁ ↦ e₁'
----------------------
η | e₁ + e₂ ↦ e₁' + e₂


      η | e₂ ↦ e₂'
----------------------
η | n₁ + e₂ ↦ n₁ + e₂'


  n₁ + n₂ = n
---------------
 η | n₁ + n₂ ↦ n
```

```
      η | e₁ ↦ e₁'
----------------------
η | e₁ * e₂ ↦ e₁' * e₂


      η | e₂ ↦ e₂'
----------------------
η | n₁ * e₂ ↦ n₁ * e₂'


  n₁ · n₂ = n
-----------------
 η | n₁ * n₂ ↦ n
```

Primer

V okolju `[x ↦ 3, y ↦ 2, z ↦ 5]` izračunamo `x + 2 * y`:

```
x + 2 * y  ↦
3 + 2 * y  ↦
3 + 2 * 2  ↦
3 + 4      ↦
7
```

Pravila ne dopuščajo nobene svobode pri računanju. Na primer, če želimo izračunati

```
[] | 2 * 3 + 5 * 6
```

potem *moramo* naprej izračunati `2 * 3`, da dobimo `6 + 5 * 6` in šele nato `5 * 6`, da dobimo `6 + 30`. Ugotovite, zakaj je tako.

Primer

Izvajanje se lahko tudi zatakne, na primer, če spremenljivka nima vrednosti:

```
[x ↦ 3]  |  x + 2 * y  ↦  3 + 2 * y
```

Naslednjega koraka ni, ker ne moremo uporabiti nobenega od pravil, ki so na voljo.

2. Ukazni programski jezik

V prejšnji lekciji smo spoznali aritmetične izraze s spremenljivkami. Spremenljivke smo obravnavali po mačehovsko, saj se jim ni dalo nastavljati vrednosti in ni bilo možno definirati novih spremenljivk.

V tej lekciji bomo spoznali ukazni programski jezik, ki ima prave spremenljivke, pogojne stavke in zanke. Po vrsti bomo obravnavali:

- sintaksa jezika
- operacijska semantika – kako se jezik izvaja
- ekvivalenca programov – kaj pomeni, da sta dva programa ekvivalentna?
- denotacijska semantika - kaj je matematični pomen programa?
- prevajalnik v strojno kodo

2.1 Sintaksa

V prejšnji lekciji smo spoznali aritmetične izraze. Dodali bomo še boolove izraze in ukaze. Podajmo abstraktna sintaksa jezika:

```
⟨aritmetični-izraz⟩ ::=
  ⟨spremenljivka⟩ |
  ⟨številka⟩ |
  ⟨aritmetični-izraz⟩ + ⟨aritmetični-izraz⟩ |
  ⟨aritmetični-izraz⟩ * ⟨aritmetični-izraz⟩

⟨boolov-izraz⟩ ::=
   true | false |
   ⟨aritmetični-izraz⟩ = ⟨aritmetični-izraz⟩ |
   ⟨aritmetični-izraz⟩ < ⟨aritmetični-izraz⟩ |
   ⟨boolov-izraz⟩ and ⟨boolov-izraz⟩ |
   ⟨boolov-izraz⟩ or ⟨boolov-izraz⟩ |
   not ⟨boolov-izraz⟩

⟨ukaz⟩ ::=
   skip |
   ⟨spremenljivka⟩ := ⟨aritmetični-izraz⟩ |
   ⟨ukaz⟩ ; ⟨ukaz⟩ |
   while ⟨boolov-izraz⟩ do ⟨ukaz⟩ done |
   if ⟨boolov-izraz⟩ then ⟨ukaz⟩ else ⟨ukaz⟩ end
```

Da bi iz zgornjih pravil dobili konkretno sintakso, moramo podati še informacijo o prioriteti in asociativnosti operatorjev. Naštejmo operatorje od nižje do višje prioritete:

- `;` (levo)
- `or` (levo)
- `and` (levo)
- `not`
- `<`, `=`
- `+` (levo)
- `*` (levo)

Na primer, `or` je levo asociativen in ima prednost pred `;`. To še vedno ni dovolj za povsem konkretno sintakso, na primer, dodati bi morali še pravila za pisanje oklepajev in pojasniti, kako se naredi leksikalno analizo (kakšna so pravila za presledke, nove vrste, komentarje ipd.)

Primer

Program, ki sešteje števila od `1` do `100` in rezultat shrani v `s`:

```
s := 0 ;
i := 0 ;
while i < 101 do
```

```
  s := s + i;
  i := i + 1
done
```

Zgornji program bi lahko zapisali v Javi takole:

```
s = 0 ;
i = 0 ;
while (i < 101) {
  s = s + i ;
  i = i + 1 ;
}
```

Abstraktna sintaksa obeh programov je enaka (vaja: narišite drevo, ki predstavlja ta program).

Tu in v nadaljevanju se ne bomo preveč posvečali podrobnostim konkretne sintakse. To *ne* pomeni, da je konkretna sintaksa nepomembna v praksi; navsezadnje so se pripravljeni programerji skregati že zaradi zamikanja kode. V zvezi s tem omenimo Wadlerjev zakon. Priporočamo tudi, da si lahko ogleda implementacijo sintakse jezika `comm` v PL Zoo.

2.2 Operacijska semantika

Sedaj nadgradimo operacijsko semantiko izrazov še s pravili za boolove izraze in ukaze. Še vedno imamo okolje η, ki spremenljivkam priredi njihove vrednosti, na primer

```
η = [x ↦ 4, y ↦ 10, u ↦ 1]
```

V našem jeziku bomo spremenljivke vedno hranile samo cela števila. Ker jim bomo tudi nastavljali vrednosti, potrebujemo ustrezno operacijo, s katero to naredimo. Če je η okolje, x spremenljivka in n celo število, potem zapis

```
η [x ↦ n]
```

pomeni okolje η, v katerem je vrednost x nastavljena na n.

Primer

Če je `η = [x ↦ 10, y ↦ 5]`, potem je `η[x ↦ 20]` enako `[x ↦ 20, y ↦ 5]`.

2.2.1 Operacijska semantika aritmetičnih in boolovih izrazov

Pravila za aritmetične izraze smo že spoznali zapišimo jih še enkrat:

```
―――――――――-
η | n ↪ n


 η(x) = n
――――――――――
η | x ↪ n

η | e₁ ↪ n₁     η | e₂ ↪ n₂
――――――――――――――――――――――
 η | e₁ + e₂ ↪ n₁ + n₂


η | e₁ ↪ n₁     η | e₂ ↪ n₂
――――――――――――――――――――――
 η | e₁ - e₂ ↪ n₁ - n₂


η | e₁ ↪ n₁     η | e₂ ↪ n₂
――――――――――――――――――――――
 η | e₁ * e₂ ↪ n₁ · n₂
```

Tudi Boolovi izrazi ne predstavljajo večje težave:

$$\frac{}{\eta \mid \texttt{true} \hookrightarrow \texttt{true}}$$

$$\frac{}{\eta \mid \texttt{false} \hookrightarrow \texttt{false}}$$

$$\frac{\eta \mid b \hookrightarrow \texttt{false}}{\eta \mid \texttt{not}\ b \hookrightarrow \texttt{true}}$$

$$\frac{\eta \mid b \hookrightarrow \texttt{true}}{\eta \mid \texttt{not}\ b \hookrightarrow \texttt{false}}$$

$$\frac{\eta \mid b_1 \hookrightarrow \texttt{false}}{\eta \mid b_1\ \texttt{and}\ b_2 \hookrightarrow \texttt{false}}$$

$$\frac{\eta \mid b_1 \hookrightarrow \texttt{true} \qquad \eta \mid b_2 \hookrightarrow v_2}{\eta \mid b_1\ \texttt{and}\ b_2 \hookrightarrow v_2}$$

$$\frac{\eta \mid b_1 \hookrightarrow \texttt{true}}{\eta \mid b_1\ \texttt{or}\ b_2 \hookrightarrow \texttt{true}}$$

$$\frac{\eta \mid b_1 \hookrightarrow \texttt{false} \qquad \eta \mid b_2 \hookrightarrow v_2}{\eta \mid b_1\ \texttt{or}\ b_2 \hookrightarrow v_2}$$

$$\frac{\eta \mid e_1 \hookrightarrow n_1 \qquad \eta \mid e_2 \hookrightarrow n_2 \qquad n_1 < n_2}{\eta \mid e_1 < e_2 \hookrightarrow \texttt{true}}$$

$$\frac{\eta \mid e_1 \hookrightarrow n_1 \qquad \eta \mid e_2 \hookrightarrow n_2 \qquad n_1 \geq n_2}{\eta \mid e_1 < e_2 \hookrightarrow \texttt{false}}$$

Ko računamo boolove vrednosti, imamo pri računanju b_1 `and` b_2 izbiro:

1. **Polno vrednotenje:** (angl. complete evaluation): vedno izračunamo b_1 *in* b_2 in nato vrednost b_1 `and` b_2
2. **Kratkostično vrednotenje** (angl. short-circuit evaluation): najprej izračunamo samo b_1. Če dobimo `false`, je vrednost b_1 `and` b_2 enaka `false` ne glede na b_2, zato ga ne izračunamo. Če je vrednost b_1 enaka `true`, izračunamo še b_2.

Zgoraj smo uporabili kratkostično vrednotenje.

Naloga

1. Kako se iz zgoraj podanih pravil vidi, da se b_1 `and` b_2 vrednoti kratkostično?
2. Podajte pravilo za polno vrednotenje b_1 `and` b_2.
3. Ali ima tudi b_1 `or` b_2 polno in kratkostično vrednotenje?

4. Podajte primer iz programerske prakse, kjer je pomembno, da vrednotimo boolove izraze kratkostično.

Naloga

Dodajte pravila za enakost celih števil ==.

2.2.2 Operacijska semantika ukazov

Semantika malih korakov za ukaze je podana z relacijo

`(η, c) ↦ (η', c')`

ki jo preberemo: »v okolju η ukaz `c` v enem koraku spremeni okolje v `η'` in se nadaljuje z ukazom `c'`«.

Relacija je določena z naslednjimi pravili:

$$\frac{\eta \mid e \hookrightarrow n}{(\eta, (x := e)) \mapsto (\eta[x \mapsto n], \mathtt{skip})}$$

$$\frac{(\eta, c_1) \mapsto (\eta', c_1')}{(\eta, (c_1 \,;\, c_2)) \mapsto (\eta', (c_1' \,;\, c_2))}$$

$$\frac{}{(\eta, (\mathtt{skip} \,;\, c_2)) \mapsto (\eta, c_2)}$$

$$\frac{\eta \mid b \hookrightarrow \mathtt{true}}{(\eta, (\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2\ \mathtt{end})) \mapsto (\eta, c_1)}$$

$$\frac{\eta \mid b \hookrightarrow \mathtt{false}}{(\eta, (\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2\ \mathtt{end})) \mapsto (\eta, c_2)}$$

$$\frac{\eta \mid b \hookrightarrow \mathtt{false}}{(\eta, (\mathtt{while}\ b\ \mathtt{do}\ c\ \mathtt{done})) \mapsto (\eta, \mathtt{skip})}$$

$$\frac{\eta \mid b \hookrightarrow \mathtt{true}}{(\eta, (\mathtt{while}\ b\ \mathtt{do}\ c\ \mathtt{done})) \mapsto (\eta, (c \,;\, \mathtt{while}\ b\ \mathtt{do}\ c\ \mathtt{done}))}$$

Pravila določajo, kako se ukaz c_1 v okolju η_1 izvaja kot zaporedje korakov

$(\eta_1, c_1) \mapsto (\eta_2, c_2) \mapsto (\eta_3, c_3) \mapsto \cdots$

Zaporedje se lahko nadaljuje v nedogled ali pa se ustavi pri ukazu `skip`, saj je to edini ukaz, ki nima naslednjega koraka.

Primer

V okolju `[x ↦ 3, y ↦ 10]` izvedemo ukaz

```
x := y + 2 ; if x < 8 then y := 0 else y := 1 end
```

takole:

```
( [x ↦  3, y ↦ 10],   x := y + 2 ; if x < 8 then y := 0 else y := 1 end ) ↦
( [x ↦ 12, y ↦ 10],   skip ; if x < 8 then y := 0 else y := 1 end       ) ↦
( [x ↦ 12, y ↦ 10],   if x < 8 then y := 0 else y := 1 end              ) ↦
( [x ↦ 12, y ↦ 10],   y := 1                                            ) ↦
( [x ↦ 12, y ↦  1],   skip)
```

Naloga

Podajte čim bolj preprost program, ki se izvaja v nedogled.

2.3 Ekvivalenca programov

Pravimo, da sta dva ukaza ekvivalentna, če se v vseh pogledih obnašata enako. To pomeni, da lahko vedno enega zamenjamo z drugim. Kako bi to razložili natančneje?

Najprej definiramo **evalvacijski kontekst**, to je del programske kode z »luknjo«, v katero lahko vstavimo kodo, označimo ga z `C[ ]`, kjer `[ ]` predstavlja luknjo. Če v `C[ ]` vstavimo kodo `A`, dobimo kodo `C[A]`.

Primer

Naj bo `C[ ]` evalvacijski kontekst:

```
while i < n do
  i := i + 1 ;
  [ ]
done
```

Tedaj je `C[s := s + i ; p := p * s]` koda

```
while i < n do
  i := i + 1 ;
  s := s + i ;
  p := p * s
done
```

Definicija

Programska koda `A` je **ekvivalentna** programski kodi `B`, če za *vse* evalvacijske kontekste `C[ ]` velja, da imata `C[A]` in `C[B]` enak rezultata in enako spreminjata okolje.

Primer

Programa

```
x := x + 1 ;
x := x + 2
```

in

```
x := x + 3
```

sta ekvivalentna. Kako bi to dokazali?

Primer

Programa

```
x := x + 1 ;
x := x + 2
```

in

```
y := y + 3
```

nista ekvivalentna, saj ju lahko razločimo s kontekstom in okoljem `η = [x ↦ 0, y ↦ 0]`.

```
x := 0 ;
y := 0 ;
[ ]
```

Če vstavimo v luknjo prvi program, bo okolje spremenil v `[x ↦ 3, y ↦ 0]`, drugi pa v `[x ↦ 0, y ↦ 3]`.

Naloga

Ugotovite, ali je program, ki sešteje prvih 100 števil

```
i := 1 ;
s := 0 ;
while i < 101 do
  s := s + i ;
  i := i + 1
done
```

ekvivalentne programu

```
s := 5050
```

2.4 Denotacijska semantika

Denotacijski semantiki se bomo posvetili na kratko, s preprostimi zgledi, saj bi natančna obravnava zahtevala več časa.

Osnovno vprašanje, na katerega odgovarja denotacijska semantika je: *»Kaj je matematični pomen programa?«* Na primer, pomen izraza `3 * (6 + 8)` je celo število `42`, matematični pomen Python funkcije

```
def fakt(n):
   if n == 0:
      return 1
   else:
      return n * fakt(n-1)
```

je matematična funkcija `n ↦ n!`, itn.

Ukaz `c` v našem programskem jeziku prav tako predstavlja funkcijo, ki sprejme okolje in vrne okolje. Na primer,

```
x := x + 1 ;
y := 10
```

predstavlja funkcijo, ki okolje `[x ↦ a, y ↦ b]` preslika v okolje `[x ↦ a+1, y ↦ 10]`. Ukaz

```
i := 1 ;
s := 0 ;
while i < n do
  s := s + i ;
  i := i + 1
done
```

predstavlja funkcijo, ki sprejme okolje `[i ↦ a, s ↦ b, n ↦ c]` takole:

- če je `c ≤ 0`, je vrednost funkcije `[i ↦ 1, s ↦ 0, n ↦ c]`
- če je `c > 0`, je vrednost funkcije `[i ↦ c, s ↦ c·(c-1)/2, n ↦ c]`

Funkcija, ki jo računa ukaz, je lahko *delna*, kar pomeni, da njena vrednost ni nujno definirana. Ukaz

```
while not (i = 100) do
  i := i + 1
done
```

predstavlja funkcijo, ki sprejme okolje `[i ↦ a]` in

- če je `a > 100`, je vrednost funkcije nedefinirana (ker se zanka nikoli ne konča)
- če je `a ≤ 100`, je vrednost funkcije okolje `[i ↦ 100]`.

2.5 Prevajalnik

Implementirajmo prevajalnik za ukazni programski jezik. Zlgedovali se bomo po prevajalniku za comm v Programming Languages Zoo, ki je razširitev ukaznega jezika, ki smo ga obravnavali do sedaj.

Jezik `comm` podpira:

- aritmetične in boolove izraze
- spremenljivke
  - deklaracija nove lokalne spremenljivke `new x := e in c`
  - nastavljanje vrednosti `x := e`
- pogojni stavek `if b then c₁ else c₂ done`
- zanka `while b do c done`
- ukaz `skip`
- sestavljeni ukaz `c₁ ; c₂`
- ukaz `print e`

Ukaz `print e` ni presenetljiv, saj le na zaslon izpiše vrednost izraza `e`.

Bolj zanimiv je ukaz `new x := e in c`, s katerim deklariramo spremenljivko `x` z začetno vrednostjo `e`, ki je veljavna v ukazu `c`. Na primer, ukaz

```
new i := 1 in
new s := 0 in
  while i < 100 do
    new j := i * s in
    i := i + 1 ;
    s := s + j
  done
```

bi v Javi zapisali kot blok

```
{
   int i = 1 ;
   int s = 0 ;
   while (i < 100) {
      int j = i * s ;
      i = i + 1 ;
      s = s + j
   }
}
```

Naloga

Ukaz `new x := e in c` programernja prisili, da novo spremenjivko inicializira, kar pomeni, da mora podati njeno začetno vrednost. Mnogi programski jeziki dopuščajo deklaracijo nove spremenljivke tako, da ni treba podati njene začetne vrednosti.

1. Kaj dopušča Java?
2. Kakšne prednosti ima jezik, ki programerja sili v inicializacijo spremenljivk?
3. Kakšne prednosti ima jezik, ki dopušča neinicializirane spremenljivke?

2.5.1 Strojni jezik

Ukaze bomo prevajali v strojni jezik za preprost procesor. Vsa aritmetična in logična obdelava podatkov poteka na **skladu**, trajnejše vrednosti so v **RAM-u**, potek izvajanja pa vodi **števec ukazov**.

Arhitektura stroja sestoji iz:

- **Program**: tabela ukazov, ki naj jih stroj izvaja (opisani so spodaj)
- **Pomnilnik RAM**: tabela celih števil (`int`), dostop po indeksih
- **Števec ukazov (`cp`)**: indeks trenutno izvajajočega se ukaza v programu.
- **Kazalec na sklad (`sp`)**: sklad je shranjen v RAM in raste navzdol. Vrh sklada je na naslovu, na katerega kaže `sp`.
- **Logične vrednosti**: `0` pomeni neresnica, vsak neničelni `int` pomeni resnica.
- **Vhod/izhod**: ukaz `PRINT` izpiše celo število z vrha sklada.

Ob inicializaciji: RAM je zapolnjen z ničlami, `pc = 0`, `sp = ram_size - 1`. Sklad je torej sprva prazen.

Stroj izvaja program v zanki: prebere navodilo na naslovu `pc`, ga izvede, nato se `pc` običajno poveča za `1`. Izjema so skoki:

- **relativni skoki** (`JMP k`, `JMPZ k`) popravijo `pc` z relatvnim odmikom `k`, nato glavni cikel `pc` še poveča za `1`. Efektivni cilj izvedenega skoka je zato `pc + k + 1` glede na začetni `pc` ukaza skoka.
- `JMPZ` pred odločitvijo porabi (pop) vrh sklada in skoči le, če je ta vrednost `0`.

Vsi aritmetični in logični ukazi delujejo na vrhu sklada. Binarne operacije vedno vzamejo najprej `y = pop`, nato `x = pop`, izračunajo rezultat `x ∘ y` in ga potisnejo nazaj (`push`). Unarne operacije vzamejo en `pop` in vrnejo en `push`.

Ukazni nabor stroja:

- `NOOP` — ne naredi ničesar (niti sklada niti `pc` ne spremeni).
- `SET k` — vzemi `a = pop` in zapiši `a` v RAM na naslov `k`.
- `GET k` — preberi RAM na naslovu `k` in prebrano potisni na sklad.
- `PUSH c` — potisni celoštevilsko konstanto `c` na sklad.
- `ADD` — `x + y`.
- `SUB` — `x - y`.
- `MUL` — `x * y`.
- `DIV` — `x / y`; če je `y = 0`, sproži napako *division by zero*.
- `MOD` — `x mod y`; če je `y = 0`, sproži napako *division by zero*.
- `EQ` — vrne `1`, če `x = y`, sicer `0`.
- `LT` — vrne `1`, če `x < y`, sicer `0`.
- `AND` — logična konjunkcija: `1` iff (`x ≠ 0` in `y ≠ 0`), sicer `0`.
- `OR` — logična disjunkcija: `1` iff (`x ≠ 0` ali `y ≠ 0`), sicer `0`.
- `NOT` — logična negacija vrha sklada: neničelen arrow.r `0`, `0` arrow.r `1`.
- `JMP k` — relativni skok z odmikom `k` (glej razdelek o skokih).
- `JMPZ k` — relativni pogojni skok: vzemi `a = pop`; če je `a = 0`, skoči z odmikom `k`.
- `PRINT` — `a = pop`; izpiši `a` na standardni izhod.

Stroj lahko med izvajanjem sproži izjemo:

- **`Illegal_address`** — poskus branja ali pisanja izven meja RAM-a.
- **`Zero_division`** — deljenje ali ostanek z ničelnim deliteljem (`DIV`, `MOD`).
- **`Illegal_instruction`** — rezervirano za neveljavna navodila (v dani različici je definirano, a se ne sproža).

Opomba: model je namerno minimalen — ni preverjanja prepolnitve/podpraznjenja sklada (push/pop lahko trčita izven dovoljenega območja RAM-a, kar se izrazi kot `Illegal_address`).

2.5.2 Kako deluje prevajanje

Kako točno deluje prevajalnik, je razvidno iz implementacije v OCamlu. Tu je kratek besedni opis.

Prevajalnik pretvori izvorni program v zaporedje strojnih ukazov. Pri tem vodi **kontekst spremenljivk** (seznam trenutno veljavnih imen), ki vsako spremenljivko preslika v **lokacijo v RAM-u** po načelu *de Bruijnovih nivojev*: prva deklaracija je na lokaciji 0, naslednja na 1, itn. Tako lahko ukaz `let` novi spremenljivki dodeli naslednjo lokacijo, `:=` pa preprosto naslovi že dodeljeno lokacijo.

Aritmetične in logične izraze prevajalnik prevede tako, da se izračunajo na skladu: najprej izračuna levi in desni podizraz (vsak potiska vrednosti na sklad), nato doda eno samo navodilo za operacijo. Boolovi vezniki so prevedeni s polnim vrednotenjem (niso kratkostični). Zaporedje ukazov prevede tako, da stakne skupaj prevode posameznih ukazov.

Pogojni stavek `if` se prevede v izračun pogoja, pogojni skok čez vejo `then`, veja `then` s skokom čez vejo `else` in nazadnje veja `else`. Zanka `while` se prevede v test na začetku zanke, pogojni skok če telo zanke, nato telo zanke in na koncu *negativen* relativni skok nazaj na test.

2.5.3 Implementacija v OCamlu

Podrobneje si oglejmo implementacijo `comm` v OCAmlu:

- abstraktna sintaksa je definirana s podatkovnimi tipi v `syntax.ml`
- konkretna sintaksa je opisana v `lexer.mll` in `parser.mly`; uporabimo generator parserjev Menhir
- preprost simulator procesorja z RAM in skladom najdemo v `machine.ml`
- prevajalnik iz `comm` v strojni jezik je v `compile.ml`
- glavni program je v `comm.ml`

Prevajalnik neposredno pretvori program v strojno kodo, ker je `comm` zelo preprost jezik. Prevajanje pravih programskih jezikov poteka preko več stopenj, z vmesnimi jeziki. Vsak naslednji jezik je nekoliko bolj preprost in bližje strojni kodi.

2.5.4 Primeri

Na primerih preizkusimo, kako se prevajajo programi in kako hitro delujejo.

`comm`:

```
# Print the sum of prime numbers up to n
new n := 1000000 in
print n ;
new s := 0 in
new k := 2 in
while k < n do
  new i := 2 in
  new isPrime := 1 in # 1 = true, 0 = false
  while isPrime = 1 and i * i < k + 1 do
    if k % i = 0 then isPrime := 0 else skip end ;
    i := i + 1
  done ;
  if isPrime = 1 then s := s + k else skip end ;
  k := k + 1
done ;
print s
```

Python:

```
# The sum of primes below n
n = 1000000
print(n)
s = 0
k = 2
```

```
while k < n:
  i = 2
  isPrime = True
  while isPrime and i * i < k + 1:
    if k % i == 0:
        isPrime = False
    else:
        pass
    i = i + 1
  if isPrime:
      s = s + k
  else:
      pass
  k = k + 1
print(s)
```

C:

```
#include <stdio.h>

int main() {
  /* The sum of primes below n */
  const long n = 1000000 ;
  printf("%ld\n", n);
  long s = 0 ;
  long k = 2 ;
  int i ;
  int isPrime ;
  while (k < n) {
    i = 2 ;
    isPrime = 1 ;
    while ((isPrime == 1) && (i * i < k + 1)) {
        if (k % i == 0) {
          isPrime = 0 ;
        } else { }
        i = i + 1 ;
    }
    if (isPrime == 1) {
      s = s + k;
    }
    else { }
    k = k + 1 ;
  }
  printf("%ld\n", s);
}
```

Rezultati zelo nestrokovno izvedene meritve, ki ji ne moremo zaupati:

Programski jezik	Čas (s)
C	0.22
Python	16.73
comm	44.88

3. Dokazovanje pravilnosti programov

Kako vemo, ali program deluje pravilno? Kako vemo, kakšen program želimo sestaviti?

Ločimo med **specifikacijo** in **implementacijo** programa:

- **Specifikacija** je opis, kaj naj želeni program počne.
- **Implementacija** je program, ki počne to, kar zahteva specifikacija.

Specifikacija je lahko podana bolj ali manj natančno, v človeškem jeziku ali zapisana v formalnem matematičnem jeziku. Specifikacije imajo več namenov:

- da pridobimo opis programa, ki naj bi ga sestavili
- da lahko preverimo, ali je implementacija pravilna
- zagotovimo kompatibilnost med različnimi deli programske opreme

Danes bomo spoznali "specifikacije v malem", s katerimi povemo, kaj naj počne konkreten košček izvorne kode. Kasneje bomo govorili tudi o specifikaciji in implementaciji večjih programskih enot, ki zajemajo zbirko podatkovnih tipov in funkcij.

3.1 Hoarova logika

V matematiki običajno uporabljamo samo eno zvrst logike, namreč klasično logiko prvega reda. V računalništvu pa je različnih logik veliko, saj ena sama ne zadošča vsem potrebam. Danes bomo spoznali logiko, ki jo je zasnoval britanski računalničar Tony Hoare.

V Hoarovi logiki delovanje programa opišemo s **Hoarovimi trojicami**

`{ P } c { Q }`

Tu sta `P` in `Q` logični formuli in `c` ukaz. Formuli `P` pravimo **predpogoj** (angl. *precondition*), formuli `Q` pravimo **končni pogoj** (ang. *postcondition*). Skupaj tvorita specifikacijo, program `c` pa je implementacija.

Izkaže se, da je pravilnost programa lažje obravnavati v dveh korakih:

1. Ali program deluje pravilno, če se ustavi?
2. Ali se program ustavi?

V ta namen uvedemo dve vrsti Hoarovih trojic:

Definicija (delna pravilnost)

`{ P } c { Q }` pomeni "*Če velja `P` in če se bo ukaz `c` končal, potem bo veljal `Q`.*"

Definicija (Popolna pravilnost)

`[ P ] c [ Q ]` pomeni: "*Če velja `P`, potem se bo `c` končal in veljal bo `Q`.*"

Zapomnimo si: delna pravilnost ne zagotavlja, da se bo `c` končal, popolna pravilnost to zagotavlja.

Kaj lahko počnemo s specifikacijami? Če imam dano specifikacijo, lahko poiščemo program, ki ji ustreza.

Primer

Poiščite program `c`, v katerem se `m` in `n` ne pojavita, in ki zamenja vrednosti spremenljivk `x` in `y`:

`{ x = m ∧ y = n } c { x = n ∧ y = m }`

Če ne bi dovolili, da se `m` in `n` pojavita v `c`, bi lahko zapisali program `x := n ; y := m`, to pa ni bil naš namen, saj želimo do vrednosti `m` in `n` dostopati izključno preko spremenljivk `x` in `y`.

Kadar zahtevamo, da se kakšna spremenljivka v programu ne pojavi, rečemo, da je **duh** (ang. ghost variable).

Primer

Kako bi zapisali speficikacijo za program `c`, ki uredi `x` in `y` po velikosti, se pravi v `x` shranimo manjšega od `x`, `y` in v `y` shranimo večjega.

Na prvi pogled je ustrezna specifikacija

```
{ true } c { x ≤ y }
```

Predpogoj `true` pomeni, da ne predpostavljamo nič posebnega, končni pogoj pa, da naj bosta `x` in `y` urejena po velikosti. A ta specifikacija ne zahteva, da moramo ohraniti vrednosti spremenljivk, zato ji zadošča tudi program

```
x := 0 ; y := 1
```

Če uporabimo duhova `m` in `n`, lahko zahtevamo, da se `x`

```
{ x = m ∧ y = n } c { x = min(m, n) ∧ y = max(m, n) }
```

Hoarove trojice običajno pišemo navpično, takole:

```
{ x = m ∧ y = n }
c
{ x = min(m, n) ∧ y = max(m, n) }
```

Če imamo opravka z večimi vrsticami kode, vrivamo predpogoje med vrstice

```
{ P₁ }
c₁
{ P₂ }
c₂
{ P₃ }
c₃
⋯
```

in jih beremo kot zaporedje Hoarovih trojic: velja `{ P₁ } c₁ { P₂ }` in velja `{ P₂ } c₂ { P₃ }` in velja `{ P₃ } c₃ { P₄ }` in tako naprej.

3.2 Pravila sklepanja

Vsaka logika ima svoja pravila sklepanja, s katerimi izpeljujemo dokaze. Za Hoarovo logiko veljajo naslednja pravila sklepanja.

3.2.1 Splošna pravila

Vedno smemo uporabiti veljavno logično in matematično sklepanje, na primer:

```
P' ⇒ P    { P } c { Q }      Q ⇒ Q'
──────────────────────────────────
          { P' } c { Q' }


{ P₁ } c { Q₁ }     { P₂ } c { Q₂ }
──────────────────────────────────
     { P₁ ∧ P₂ } c { Q₁ ∧ Q₂ }
```

Naj bodo `FV(P)` vse spremenljivke, ki se pojavljajo v formuli `P` (free variables) in `FA(c)` vse spremenljivke, ki jih `c` nastavlja (assigned variables). Na primer:

`FV(x ≤ y ∨ x > 0) = {x, y}`

`FA(if x < y then x := y + 3 else skip end) = { x }`

Velja pravilo:

```
FV(P) ∩ FA(c) = ∅
──────────────────
  { P } c { P }
```

To pravilo pove, da ukaz `c` ne vpliva na izjavo `P`, če nimata skupnih spremenljivk. Tako pravilo ne bi veljalo v programskem jeziku s kazalci, saj bo lahko `c` spreminjal vrednosti, ki so dosegljive iz `P`, čeprav jih `P` ne omenja.

3.2.1.1 Pravilo za `skip`

Ukaz `skip` nima nikakršnega učinka na veljavnost izjave:

```
──────────────────
{ P } skip { P }
```

3.2.1.2 Pravilo za pogojni stavek

Pri pogojnem stavku obravnavamo dva primera, enega za `then` in drugega za `else`.

```
{ P ∧ b } c₁ { Q }       { P ∧ ¬b } c₂ { Q }
──────────────────────────────────────────────
    { P } if b then c₁ else c₂ end { Q }
```

3.2.1.3 Pravilo za c_1 ; c_2

Pravilo za `;` veriži končni pogoj prvega ukaza s predpogojem drugega:

```
{ P } c₁ { Q }      { Q } c₂ { R }
──────────────────────────────────
       { P } c₁ ; c₂ { R }
```

3.2.1.4 Pravilo za zanko `while`

```
        { P ∧ b } c { P }
──────────────────────────────
{ P } while b do c done { ¬ b ∧ P }
```

Formuli `P` pravimo *invarianta* zanke `while`. Izmed vseh pravil, je tega najtežje uporabljati, ker je treba imeti nekaj izkušenj, da najdemo ustrezni `P`.

3.2.1.5 Pravilo za prirejanje

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

Zapis `P[x ↦ e]` pomeni “v izjavi `P` zamenjaj `x` z `e`”.

3.2.2 Popolna pravilnost

Vsa zgornja pravila, razen dveh, lahko predelamo v popolno pravilnost, na primer:

```
[ P ∧ b ] c₁ [ Q ]       [ P ∧ ¬b ] c₂ [ Q ]
──────────────────────────────────────────────
    [ P ] if b then c₁ else c₂ end [ Q ]
```

Prva izjema je pravilo

```
FV(P) ∩ FA(c) = ∅
──────────────────
  { P } c { P }
```

ki ga predelamo takole

$$\frac{FV(P) \cap FA(c) = \varnothing \qquad [\ R\ ]\ c\ [\ Q\ ]}{[\ R \wedge P\ ]\ c\ [\ Q \wedge P\ ]}$$

Pozor

Pravilo

$$\frac{FV(P) \cap FA(c) = \varnothing}{[\ P\ ]\ c\ [\ P\ ]}$$

ni veljavno. Če bi bilo, bi lahko dokazali

$$[\ x > 0\ ]\ \texttt{while true do skip done}\ [x > 0]$$

kar pa ne velja.

Pri zanki `while` zagotovimo, da se bo končala, tako da poiščemo količino, ki se zmanjšuje, a se ne more zmanjševati v nedogled. Na primer, to je lahko celoštevilska pozitivna vrednost.

Pozor

Realna pozitivna vrednost se lahko zmanjšuje v nedogled:

$$0.1 > 0.01 > 0.001 > 0.0001 > \ldots$$

Tudi celoštevilska vrednost se lahko zmanjšuje v nedogled:

$$2 > 1 > 0 > -1 > -3 > -5 > -7 > \ldots.$$

Pravilo za popolno pravilnost `while` se glasi:

Naj bo `e` količina, ki se ne more v nedogled zmanjševati (na primer naravno število):

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

V tem pravilu je `z` duh, kar formalno napišemo kot $z \notin FA(c)$. Kako pa ta pravila v praksi uporabljamo? Poglejmo nekaj primerov.

3.3 Primeri

Naloga

Dokaži pravilnost programa:

$$\{\ x \leq 7\ \}$$
$$x := x + 3$$
$$\{\ x \leq 10\ \}$$

Naloga

Zapiši s Hoarovo logiko:

1. Program `c` se ne ustavi.
2. Program `c` se ustavi.

Naloga

Dokaži pravilnost programa, kjer predpostavimo, da v spremenljivkah hranimo realna števila (da ni težav z deljenjem z 2):

```
{ x ≤ y }
s := (x + y) / 2
{ x ≤ s ≤ y }
```

Naloga

Dogovor: v predpogojih in končnih pogojih namesto `P ∧ Q` pišemo tudi `P, Q`.

Dokaži pravilnost programa:

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

Rešitev:

```
{ b ≥ 0 }
i := 0 ;
{ b ≥ 0, i = 0 }
p := 1 ;
{ b ≥ 0, i = 0, p = 1 } # logično sklepamo, je zelo easy
{ p = a ^ i, i ≤ b }
while i < b do
   { i < b, p = a ^ i, i ≤ b }
   # iz p = a^i sledi p·a = a^(i+1)
   # iz i < b sledi i+1 < b+1 sledi i+1 ≤ b (ker i, b celi števili)
   { p · a = a ^ (i + 1), (i + 1) ≤ b }
   p := p * a ;
   { p = a ^ (i + 1), (i + 1) ≤ b }
   i := i + 1
   { p = a ^ i, i ≤ b }
done
{ i ≥ b, p = a ^ i, i ≤ b } # očitno
{ i = b, p = a ^ i } # očitno
{ p = a ^ b }
```

Popolna pravilnost: zmanjšuje se celoštevilska količina `e = b - i ≥ 0`. Imamo invarianto `Q ≡ (p = a ^ i ∧ i ≤ b)`

```
while i < b do
   [ Q, i < b, b - i = z ]
   [ i < b, b - i = z ]
   p := p * a ;
   [ i < b, b - i = z ]
   ⇒
   [ b - i = z ]
   ⇔
   [ b = z + i ]
   ⇒
   [ b < z + i + 1 ]
   ⇔
   [ b - i - 1 < z ]
   ⇔
   [ b - (i+1) < z ]
   i := i + 1
   [ b - i < z ]
done
```

Naloga

Dokaži pravilnost programa:

```
[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) ]
```

Rešitev:

```
[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
   [x ≤ y, x = m ∧ y = n]
   [m ≤ n, x = m ∧ y = n]
   skip
   [ m ≤ n, x = m ∧ y = n ] # očitno sledi
   [ x = min(m, n) ∧ y = max(m, n) ]
end
[ x = min(m, n) ∧ y = max(m, n) ]
```

4. λ-račun

V začetku 20. stoletja so se logiki spraševali, kako bi natančno opredelili pojem »računski postopek«. Leta 1936 je Alan Turing podal pojem stroja, ki še danes velja za standard. A pred Turingom je Alonzo Church podal svojo razlago računanja, ki jo je poimenoval **λ-račun**. Kasneje se je izkazalo, da sta oba pojma ekvivalentna, vsaj kar se tiče računanja s števili.

Kasneje je λ-račun pomembno vplival na razvoj programskih jezikov, zato je prav da ga spoznamo bolj podrobno. Poleg tega je programiranje v λ-računu dobra vaja iz razumevanja osnovnih principov funkcijskega programiranja. Seveda čistega λ-računa, ki ga bomo uporabljati, nihče ne uporablja v praksi, tako kot tudi ne Turingovih strojev.

Danes bomo spoznali **λ-račun brez tipov**, v poglavju o deklarativnem programiranju pa še λ-račun s tipi.

4.1 Funkcijski predpis

V matematiki poznamo **funkcijski predpis**:

```
x ↦ e
```

To preberemo »`x` se slika v `e`«, pri čemer je `e` izraz, ki lahko vsebuje `x`. Primer:

```
x ↦ x² + 3·x + 7
```

Funkcijske predpise včasih imenujemo tudi **anonimne funkcije**, ker se razlikujejo od običajnih definicij funkcij, ki le-te poimenujejo:

```
f(x) := x² + 3·x + 7
```

Hkrati smo podali funkcijo in jo poimenovali `f`. Če poimenovanje in podajanje predpisa razstavimo, lahko zgornjo definicijo podamo tudi takole:

```
f := (x ↦ x² + 3·x + 7)
```

Torej so funkcijski predpisi *bolj splošni* kot imenovane funkcije.

> **Opomba**
>
> Funkcijski predpis sam po sebi ne določa funkcije, saj moramo opredeliti še domeno in kodomeno funkcije.
>
> λ-račun sestoji samo iz funkcijskih predpisov, je *račun* (sistem za računanje s simboli).

Funkcijski predpis lahko **uporabimo** na argumentu. Na primer, zgornji `f` lahko uporabimo na `3` in dobimo izraz `f(3)`, ki mu pravimo **aplikacija**.

Dejstvo, da je funkcija poimenovana `f`, nima nobene zveze z aplikacijo. Prav lahko bi pisali neposredno

```
(x ↦ x² + 3·x + 7)(3)
```

To se morda zdi nenavadno, a je lahko koristno v programiranju (kot bomo videli kasneje). Nekateri programski jeziki imajo funkcijske predpise:

- Python: `lambda x: x**2 + 3*x + 7`
- Haskell: `\x -> x**2 + 3*x + 7`
- OCaml: `fun x -> x*x + 3*x + 7`
- Racket: `(lambda (x) (+ (* x x) (* 3 x) 7))`
- Mathematica: `#² + 3*# + 7 &` ali `Function[x, x² + 3*x + 7]`

4.1.1 Vezane in proste spremenljivke

V funkcijskem predpisu

`x ↦ x² + 3·x + 7`

se `x` imenuje **vezana spremenljivka**. S tem želimo povedati, da je `x` veljavna samo znotraj funkcijskega predpisa, je kot neke vrste lokalna spremenljivka. Če jo preimenujemo, se funkcijski zapis ne spremeni:

`a ↦ a² + 3·a + 7`

Poudarimo, da štejemo funkcijska predpisa za enaka, če se razlikujeta le po tem, kateri simbol je uporabljen za vezano spremenljivko.

V funkcijskem predpisu lahko nastopa tudi kaka dodatna spremenljivka, ki ni vezana. Pravimo ji **prosta spremenljivka**, na primer:

`x ↦ a·x² + b·x + c`

Tu so `a`, `b` in `c` proste spremenljivke. Teh ne smemo preimenovati, ker bi se pomen izraza spremenil, če bi to storili. (Pravzaprav bi lahko rekli, da imamo še dodatne proste spremenljivke `·`, `+` in `²`.)

Vezane in proste spremenljivke se pojavljajo tudi drugje v matematiki in računalništvu:

- v integralu $\int x^2 + a \cdot x \, dx$ je x vezana spremenljivka in a prosta
- v vsoti $\sum_{i=1}^{n} i \cdot (i + k)$ je i vezana spremenljivka, n in k sta prosti
- v limiti $\lim_{x \to a} (x - a)/(x + a)$ je x vezana spremenljivka, a je prosta
- v formuli $\exists x \in \mathbb{R} \,.\, x^3 = y$ je x vezana spremenljivka, y je prosta
- v programu

```
for (int i = 0; i < 10; i++) {
    s += i;
}
```

je `i` vezana spremenljivka, `s` je prosta.

- v programu

```
if (false) {
   int s = 0 ;
   for (int i = 0; i < 10; i++) {
      s += i;
   }
}
```

sta `s` in `i` vezani spremenljivki.

Pomembno je, katere spremenljivke so proste in katere vezane:

- `x ↦ a · x + b` pomeni »pomnoži z `a` in prištej `b`«.
- `a ↦ a · x + b` pomeni »pomnoži z `x` in prištej `b`«.
- `b ↦ a · x + b` pomeni »prištej `a · x`«.

4.1.2 Substitucija ali zamenjava

Operacija, ki v izrazu prosto spremenljivko zamenja z izrazom, se imenuje **zamenjava** ali **substitucija**. Zapišemo jo

`e [e'/x]`

in preberemo »v `e` zamenjaj `x` z `e'`«.

Primeri:

- `(x² + 3·x + 7)[3/x]` je enako `3² + 3·3 + 7`,
- `f(a + b)[(b + 1)/a]` je enako `f((b + 1) + b)`,
- `f(a + b)[(x ↦ x²)/f]` je enako `(x ↦ x²)(a + b)`.

Ko napravimo substitucijo, moramo paziti, da se prosta spremenljivka ne »ujame«. S tem želimo povedati, da bi prosto spremenljivko vstavili v podizraz, v katerem je že veljavna

enako poimenovana vezana spremenljivka, s čimer bi prišlo do zmede med obema spremenljivkama. Na primer, če v integralu

$$\int_0^1 (x^2 + b)\, dx \tag{4.1}$$

prosto spremenljivo b naivno zamenjamo z izrazom $x + a$, dobimo

$$\int_0^1 (x^2 + x + a)\, dx \tag{4.2}$$

To ni prav, saj je je x iz $x + a$ ujel v integralu. Da dobimo pravilen rezultat, moramo vezano spremenljivko v integralu najprej preimenovati,

$$\int_0^1 (x^2 + b)\, dx = \int_0^1 (t^2 + b)\, dt, \tag{4.3}$$

in šele nato za b vstavimo $x + a$:

$$\int_0^1 (t^2 + x + a)\, dt. \tag{4.4}$$

4.1.3 Računsko pravilo ali β-redukcija

V λ-računu poznamo eno samo računsko pravilo, ki se imenuje tudi **β-redukcija** in se glasi

`(x ↦ e₁)(e₂)  =  e₁[e₂/x]`

To preberemo

> *Če uporabimo funkcijski predpis* `x ↦ e₁` *na argumentu* `e₂`*, dobimo izraz* `e₁`*, v katerem* `x` *zamenjamo z* `e₂`*.*

Pravzaprav je to pravilo, ki ga vsi uporabljamo, ko računamo pri matematičnih predmetih, le da imamo običajno opravka s poimenovanimi funkcijami.

Primer

Če je `f(x) = x² + 3 · x + 7`, potem je `f(a + 3) = (a + 3)² + 3 · (a + 3) + 7`.

Uporabili smo β-redukcijo, saj smo v funkcijskem predpisu `f` vezano spremenljivko `x` zamenjali z `a + 3`. Taisti primer z λ-računom:

`(x ↦ x² + 3 · x + 7)(a + 3)  =  (a + 3)² + 3 · (a + 3) + 7`

Opozorilo

Pozor, pravilo za funkcijski zapis *ne* trdi `(x ↦ x² + 3·x + 7)(a + 3) = a² + 9 · a + 25`, ampak le `(x ↦ x² + 3 · x + 7)(a + 3) = (a + 3)² + 3 · (a + 3) + 7`.

Da bi iz `(a + 3)² + 3 · (a + 3) + 7` dobili `a² + 9 · a + 25`, bi morali uporabiti še dodatna pravila algebre in aritmetike, ki jih λ-račun ne zajema. Tu števila in aritmetične operacije uporabljamo kot primitivne simbole in se pretvarjamo, da ne poznamo njihovega običajnega pomena.

4.1.4 Gnezdeni funkcijski predpisi

Funkcijske predpise lahko gnezdimo, ali jih uporabljamo kot argumente. Primeri:

1. `(x ↦ (y ↦ x · x + y))(42)  =  (y ↦ 42 · 42 + y)`
2. `((x ↦ (y ↦ x · x + y))(42))(1)  =  (y ↦ 42 · 42 + y)(1) =  42 · 42 + 1`

3. `(f ↦ f (f (3))) (n ↦ n · n + 1)  =  (n ↦ n · n + 1) ((n ↦ n · n + 1) (3))`
   `= (n ↦ n · n + 1) (3 · 3 + 1) = (3 · 3 + 1) · (3 · 3 + 1) + 1`

Podobno kot pri integralih, je treba pred vstavljanjem izraza v funkcijski predpis po potrebi preimenovati vezano spremenljivko:

- pravilno: `(x ↦ (y ↦ x · y²)) (z + 1) = (y ↦ (z + 1) · y²)`
- narobe: `(x ↦ (y ↦ x · y²)) (y + 1) = (y ↦ (y + 1) · y²)`
- pravilno: `(x ↦ (y ↦ x · y²)) (y + 1) = (x ↦ (a ↦ x · a²)) (y + 1) = (a ↦ (y + 1) · a²)`

4.2 λ-račun

Zapis `x ↦ e` postane dolgovezen, ko funkcijske zapise gnezdimo, zato bomo uporabili starejši zapis

```
λ x . e
```

To je prvotni zapis funkcijskih predpisov, kot ga je zapisal Alonzo Church, vaš akademski praded! Temu zapisu pravimo **abstrakcija** izraza `e` glede na spremenljivko `x`.

Poleg tega bomo aplikacijo `f(x)` pisali brez oklepajev `f x`. Seveda pa oklepaje dodamo, kadar bi lahko prišlo do zmede. Dogovorimo se, da je aplikacija *levo asociativna*, torej

```
e₁ e₂ e₃  =  (e₁ e₂) e₃
```

V abstrakciji `λ` vedno veže največ, kolikor lahko. Torej je `λ x . e₁ e₂ e₃` je enako `λ x . (e₁ e₂ e₃)` in ni enako `(λ x . e₁) e₂ e₃`.

Abstraktna sintaksa λ-računa je nadvse preprosta:

```
⟨izraz⟩ ::= ⟨spremenljivka⟩
          | ⟨izraz⟩ ⟨izraz⟩
          | λ ⟨spremenljivka⟩ . ⟨izraz⟩
```

Kadar imamo gnezdene abstrakcije

```
λ x . λ y . λ z . e
```

jih gnezdimo `λ x . (λ y . (λ z . e))`. Dogovorimo se še, da lahko tako gnezdeno abstrakcijo krajše zapišemo

```
λ x y z . e
```

4.2.1 Evalvacijske strategije

Pravilo za računanje lahko uporabimo na različne načine. Primer:

```
(λ x . (λ f . f x) (λ y . y)) ((λ z . g z) u)
```

je enak

```
(λ x . (λ f . f x) (λ y . y)) (g u)
```

in prav tako

```
(λ x . (λ y . y) x) ((λ z . g z) u)
```

Vendar pa je λ-račun **konfluenten**, kar pomeni, da vrstni red računanja ni pomemben. Natančneje, če ima `e` dva možna računska koraka, `e ↦ e₁` in `e ↦ e₂`, potem lahko v `e₁` in v `e₂` izvedemo take računske korake, da se bosta pretvorila v isti izraz.

V zgornjem primeru:

```
(λ x . (λ f . f x) (λ y . y)) (g u) =
(λ x . (λ y . y) x) (g u)  =
(λ x . x) (g u) =
g u
```

in

```
(λ x . (λ y . y) x) ((λ z . g z) u) =
(λ x . x) ((λ z . g z) u) =
(λ z . g z) u =
g u
```

Dobili smo izraz, v katerem ne moremo več narediti računskega koraka. Pravimo, da je tak izraz v **normalni obliki**.

Postavi se vprašanje, kako sistematično računati. Poznamo nekaj strategij:

- **Neučakana (eager evaluation):** v izrazu $e_1\ e_2$ najprej do konca izračunamo e_1 da dobimo $\lambda\ x\ .\ e$, nato do konca izračunamo e_2, da dobimo e_2' in šele nato vstavimo e_2' v e.
- **Lena (lazy evaluation):** v izrazu $e_1\ e_2$ najprej izračunamo e_1, da dobimo $\lambda\ x\ .\ e$, nato pa takoj vstavimo e_2 v e.

Poleg tega lahko računamo znotraj abstrakcij ali ne. Programski jeziki znotraj abstrakcij ne računajo (to bi pomenilo, da se računa telo funkcije, še preden smo funkcijo poklicali).

Primer

Izračunajmo `(λ x . (λ y . x) z) ((λ t . t) u)` na različne načine.

Neučakano (argument izračunamo, preden ga vstavimo):

```
(λ x . (λ y . x) z) ((λ t . t) u) =
(λ x . (λ y . x) z) u =
(λ y . u) z =
u
```

Leno (argument vstavimo takoj):

```
(λ x . (λ y . x) z) ((λ t . t) u) =
(λ y . ((λ t . t) u)) z =
(λ t . t) u =
u
```

Računamo tudi znotraj λ-abstrakcij neučakano:

```
(λ x . (λ y . x) z) ((λ t . t) u) =
(λ x . x) u =
u
```

Opomba

Obstajajo izrazi, ki nimajo normalne oblike in jih ne moremo »izračunati do konca«. Primer je `(λ x . x x) (λ x . x x)`, ki ima natanko en možen računski korak, a ta pripelje spet do istega izraza:

```
(λ x . x x) (λ x . x x) =
(λ x . x x) (λ x . x x) =
(λ x . x x) (λ x . x x) =
…
```

4.3 Programiranje v λ-računu

Na prvi pogled se zdi, da se v λ-računu ne da izračunati nič koristnega. A velja ravno obratno, λ-račun je po računski moči ekvivalenten Turingovim strojem – je splošen programski jezik.

4.3.1 Identiteta, kompozicija in konstantna preslikava

Začnimo z osnovnimi preslikavami. **Identiteta** je preslikava $x \mapsto x$, ki jo zapišemo tudi kot

```
id := λ x . x
```

Bolj zanimiva je **kompozicija** preslikav:

```
compose := λ f g x . f (g x)
```

Tudi konstantne funkcije ni težko definirati:

```
const := λ c x . c
```

Izraz `const e` je funkcija, ki vedno vrne `e`. Običajno se namesto `const` piše `K`.

4.3.2 Boolove vrednosti in pogojni stavek

Kako pa lahko dobimo Boolove vrednosti in pogojni stavek? Iščemo λ-izraze `true`, `false`, in `if`, za katere velja

```
if true a b  = a
if false a b = b
```

V λ-računu ustrezne izraze definiramo takole:

```
true := λ x y . x
false := λ x y . y
if := λ b t e . b t e
```

Preverimo, da imajo ustrezne lastnosti:

```
if true a b =
(λ b t e . b t e) true a b =
(λ t e . true t e) a b =
(λ e . true a e) b =
true a b =
(λ x y . x) a b =
(λ y . a) b =
a
```

Sami preverite, da velja `if false a b = b`.

4.3.3 Urejeni pari

Da bomo lahko programirali z večimi vrednostmi hkrati, potrebujemo urejene pare, ki jih lahko gnezdimo, da dobimo urejene trojice, četverice itd. Potrebujemo izraze `pair`, `first`, in `second`, ki zadoščajo enačbam:

```
first (pair a b) = a
second (pair a b) = b
```

Naslednji programi delujejo:

```
pair := λ x y . λ f . f x y
first := λ p . p (λ x y . x)
second := λ p . p (λ x y. y)
```

Preverimo, da velja druga enačba:

```
second (pair a b) =
second ((λ x y . λ p . p x y) a b) =
second (λ p . p a b) =
(λ q . q (λ x y. y)) (λ p . p a b) =
(λ p . p a b) (λ x y. y) =
(λ x y. y) a b =
b
```

4.3.3.1 Churcheva števila

Tudi naravna števila lahko predstavimo z λ-izrazi: število `n` predstavimo z izrazom, ki sprejme funkcijo in jo `n`-krat gnezdi:

```
0 := λ f x . x
1 := λ f x . f x
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))))
6 := λ f x . f (f (f (f (f (f x)))))
7 := λ f x . f (f (f (f (f (f (f x))))))
8 := λ f x . f (f (f (f (f (f (f (f x)))))))
9 := λ f x . f (f (f (f (f (f (f (f (f x))))))))
```

Na primer `3 foo bar = foo (foo (foo bar))`.

S Churchevimi števili lahko računamo. Ali razumete, kako delujejo naslednik, vsota in množenje?

```
succ := λ n f x . f (n f x)

+ := λ n m f x . (n f) ((m f) x)

* := λ m n f x . m (n f) x
```

Kako izračunamo predhodnik števila `n`? Vse kar lahko naredimo z `n` je, da `n`-krat uporabimo neko funkcijo. Poglejmo, kaj dobimo, če trikrat uporabimo `f (x, y) := (x + 1, x)` na paru `(0, 0)`:

```
f (f (f (0, 0))) =
f (f (1, 0)) =
f (2, 1) =
(3, 2)
```

Dobili smo število `3` in njegov predhodnik `2`, kar pripelje do programa

```
pred := λ n . second (n (λ p. pair (succ (first p)) (first p)) (pair 0 0))
```

Še nekaj programov, s katerimi primerjamo števila:

```
iszero := λ n . n (K false) true

<= := λ m n . iszero (n pred m)

>= := λ m n . iszero (m pred n)

< := λ m n . <= (succ m) n

> := λ m n . >= m (succ n)
```

4.3.4 Implementacija λ-računa

Ročno računanje z λ-računom je mukotrpno. V PL Zoo najdete programski jezik `lambda`, ki olajša delo. Na voljo je tudi spletni vmesnik za `lambda`. (Kogar zanima, kako se tak vmesnik naredi, si lahko ogleda `repl-in-browser`).

> **Nasvet**
>
> Na izpitu boste lahko uporabljali računalnik, a brez spletne povezave. S seboj prinesite `lambda.zip`, da boste lahko uporabljali `lambda` lokalno v brskalniku.

Da ohranimo kompatibilnost z računalniki iz leta 1968, se izognemo simbolu `λ` in ga nadomestimo z `^`.

```
-- Urejeni pari

pair := ^ a b . ^p . p a b ;
first := ^ p . p (^ x y . x) ;
second := ^ p . p (^ x y. y) ;
```

```
-- Konstantna funkcija

K := ^ x y . x ;

-- Boolove vrednosti

true  := ^ x y . x ;
false := ^ x y . y ;
if    := ^ p x y . p x y ;

and := ^ x y . if x y false ;

-- Churchova števila

0  := ^ f x . x ;
1  := ^ f x . f x ;
2  := ^ f x . f (f x) ;
3  := ^ f x . f (f (f x)) ;
4  := ^ f x . f (f (f (f x))) ;
5  := ^ f x . f (f (f (f (f x)))) ;
6  := ^ f x . f (f (f (f (f (f x))))) ;
7  := ^ f x . f (f (f (f (f (f (f x)))))) ;
8  := ^ f x . f (f (f (f (f (f (f (f x))))))) ;
9  := ^ f x . f (f (f (f (f (f (f (f (f x)))))))) ;
10 := ^ f x . f (f (f (f (f (f (f (f (f (f x)))))))))) ;

succ := ^n f x . f (n f x) ;

+ := ^n m f x . (n f) ((m f) x) ;

iszero := ^n . n (K false) true ;

pred := ^n . second (n (^p. pair (succ (first p)) (first p)) (pair 0 0)) ;

<= := ^m n . iszero (n pred m) ;

>= := ^m n . iszero (m pred n) ;

< := ^m n . <= (succ m) n ;

> := ^m n . >= m (succ n) ;

-- Church-Scottova števila

0' := ^ f x . x ;
1' := ^ f x . f 0' x ;
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. Deklarativno programiranje

Z λ-računom smo spoznali uporabno vrednost funkcij in dejstvo, da lahko z njimi programiramo na nove in zanimive načine. A kot programski jezik λ-račun ni primeren, saj je zelo neučinkovit, poleg tega pa se programer večino časa ukvarja s kodiranjem podatkov s pomočjo funkcij. Da ne omenjamo grozne sintakse.

Obdržimo, kar smo se od λ-račun naučili:

1. **Funkcije so podatki.** V programskem jeziku lahko funkcije obravnavamo enakovredno vsem ostalim podatkom. To pomeni, da lahko funkcije sprejmejo druge funkcije kot argumente, ali jih vrnejo kot rezultat, da lahko tvorimo podatkovne strukture, ki vsebujejo funkcije ipd.
2. **Program ni nujno zaporedje ukazov.** V λ-računu program *ni* navodilo, ki pove, kako naj se izvede zaporedje ukazov. Kot smo videli, je vrstni red računanja nedoločen, saj je v splošnem možno izraz v λ-računu poenostaviti na več načinov (ki pa vsi vodijo do istega odgovora).

Kakšne vrste programiranje pa potemtakem je λ-račun, če ni ukazno? Nekateri uporabljajo izraz **funkcijsko programiranje**, mi pa bomo raje rekli **deklarativno programiranje**. S tem izrazom želimo poudariti, da s programom izrazimo (najavimo, deklariramo) strukturo podatka, ki ga želimo imeti, ne pa nujno kako se izračuna. Postopek, s katerim pridemo do rezultata je nato v večji ali manjši meri prepuščen programskemu jeziku.

5.1 Podatki

V λ-računu moramo vse podatke predstaviti, ali *kodirati*, s funkcijami. Tako opravilo je zamudno in podvrženo napakam, ker krši načelo:

Če mora programer neki koncept v programu izraziti tako, da ga simulira s pomočjo drugih konceptov, je večja možnost napake. Poleg tega prevajalnik ne bo imel informacije o tem, kaj programer počne, zato bo prepoznal manj napak in imel manj možnosti za optimizacijo.

Ponazorimo to načelo z idejo. Denimo, da želimo računati s seznami. Od programskega jezika pričakujemo *neposredno* podporo za sezname. Pričakujemo, da lahko seznam preprosto naredimo, dostopamo do njegovih sestavnih delov, analiziramo, podamo kot argument funkciji itd. Ali programski jeziki, ki jih že poznamo, podpirajo sezname? Poglejmo:

- **C**: sezname moramo simulirati s pomočjo struktur (`struct`) in kazalcev
- **Java**: sezname moramo simulirati z objekti
- **Python**: seznami so vgrajeni, z njimi lahko delamo neposredno

Python težavo torej reši tako, da ima sezname kar vgrajene. To je prikladna rešitev, vendar pa ne moremo pričakovati, da bomo lahko z vgrajenimi podatkovnimi strukturami zadovoljili vse potrebe. Programerju moramo dodatno omogočiti, da definira *nove* strukture in *nove* načine organiziranja idej, ki jih načrtovalec jezika ni vnaprej predvidel. Različni programski jeziki to omogočajo na različne načine:

- **C**: definiramo lahko strukture (`struct`), unije (`union`), uporabljamo kazalce, itd.
- **Java**: definiramo razrede in podatke organiziramo kot objekte
- **Python**: definiramo razrede in podatke organiziramo kot objekte

Zdi se, da se novejši jeziki vsi zanašajo na objekte. A to še zdaleč ni edina rešitev za predstavitev podatkov – in tudi ne najboljša. Spoznali bomo *neposredne* konstrukcije podatkovnih tipov, ki *niso* simulacije s pomočjo kazalcev ali objektov. (Seveda prevajalnik podatke v pomnilniku predstavi s kazalci, a programerju tega ni treba vedeti, ali o tem razmišljati.)

Navdih bomo vzeli iz matematike, kjer namesto podatkovnih tipov delamo z množicami.

5.2 Konstrukcije množic

V matematiki gradimo nove množice z nekaterimi osnovnimi operacijami, ki jih večinoma že poznamo, a jih vseeno ponovimo.

5.2.1 Zmnožek ali kartezični produkt

Zmnožek ali **kartezični produkt** množic A in B je množica, katere elementi so urejeni pari:

- za vsak $x \in A$ in $y \in B$ lahko tvorimo **urejeni par** $(x, y) \in A \times B$

Če imamo element $p \in A \times B$, lahko dobimo njegovo **prvo komponento** $\pi_1(p) \in A$ in **drugo komponento** $\pi_2(p) \in B$. Pri tem velja:

$$\pi_1(x, y) = x \qquad \pi_2(x, y) = y \tag{5.1}$$

Operacijama π_1 in π_2 pravimo **projekciji**.

Tvorimo lahko tudi zmnožek več množic, na primer $A \times B \times C \times D$, v tem primeru imamo urejene četverice (x, y, z, t) in štiri projekcije, π_1, π_2, π_3 in π_4.

5.2.2 Vsota ali disjunktna unija

Vsota množic $A + B$ je množica, ki vsebuje dve vrsti elementov:

- za vsak $x \in A$ lahko tvorimo element $\iota_1(x) \in A + B$
- za vsak $y \in A$ lahko tvorimo element $\iota_2(x) \in A + B$

Predstavljamo si, da je vsota $A + B$ sestavljena iz dveh ločenih kosov A in B. Simbola ι_1 in ι_2 sta *oznaki*, ki povesta, iz katerega kosa je element. To je pomembno, kadar tvorimo vsoto $A + A$. Če je $x \in A$, potem sta $\iota_1(x)$ in $\iota_2(x)$ *različna* elementa vsote $A + A$.

Operacijama ι_1 in ι_2 pravimo **injekciji**.

Vsoti pravimo tudi **disjunktna unija**. Ločiti jo moramo od običajne unije. V vsoti $A + B$ se A in B nikoli ne prekrivata, ker elemente razločimo z oznakama ι_1 in ι_2. V uniji $A \cup B$ so lahko nekateri elementi *hkrati* v A in v B. V skrajnem primeru imamo celo $A \cup A = A$, tako da je vsak element v obeh kosih.

Če želimo uporabiti element $u \in A + B$ v neki konstrukciji ali dokazu, **obravnavamo primera**:

1. $u = \iota_1(x)$ za neki $x \in A$ ali
2. $u = \iota_2(y)$ za neki $y \in B$.

To je matematična zasnova konstrukcij za obravnavanje primerov v programskih jezikih (`match` v OCamlu, `case` v C/C++).

Matematiki ne poznajo prikladnega zapisa za obravnavanje primerov. Nasploh matematiki vsoto množic slabo poznajo in jo neradi uporabljajo (kdo bi vedel, zakaj). V programiranju so vsote izjemno koristne, a na žalost jih pogosto programski jeziki bodisi ne podpirajo bodisi implementirajo narobe.

Poglejmo si primer uporabe vsot v programiranju. Na primer, da v spletni trgovini prodajamo čevlje, palice in posode. Čevelj ima barvo in velikost, palica velikost in posoda prostornino. Če je B množica vseh barv in $\mathbb{N}$ množica naravnih števil, lahko izdelek predstavimo kot element množice

$$(B \times \mathbb{N}) + \mathbb{N} + \mathbb{N} \tag{5.2}$$

Res: črn čevelj velikosti 42 je element $\iota_1(\texttt{črna}, 42)$, palica dolžine 7 je $\iota_2(7)$, posoda s prostornino 7 pa je $\iota_3(7)$. Oznaki ι_2 in ι_3 ločita med palicami in posodami. Seveda je tak zapis s programerskega stališča nepraktičen, zato ga bomo v programskem jeziku izboljšali.

5.2.3 Eksponent ali množica funkcij

Eksponent B^A, ki ga pišemo tudi $A \to B$, je množica vseh funkcij iz A v B. Če je $f \in B^A$, pravimo, da je A **domena** in B **kodomena** funkcije f.

Dogovorimo se, da je $\to$ asociira desno, se pravi

$$A \to B \to C = A \to (B \to C). \tag{5.3}$$

Primeri:

1. $\mathbb{R} \to \mathbb{R}$ je množica realnih funkcij ene spremenljivke. Primeri: sin, cos, exp in $x \mapsto 2x + 3$,
2. $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$ je množica realnih funkcija dveh spremenljivk. Primeri: $+$, $\times$, in $(x, y) \mapsto x^2 + y^3$.
3. $\mathbb{R} \to \mathbb{R} \to \mathbb{R}$ je množica funkcij, ki sprejmejo eno realno število in vrnejo funkcijo, ki sprejme še eno realno število in vrne realno število, na primer $x \mapsto (y \mapsto x^2 + y^3)$.
4. $(\mathbb{R} \to \mathbb{R}) \to \mathbb{R}$ je množica funkcij, ki sprejmejo realno funkcijo in vrnejo realno število, na primer $f \mapsto \int_0^1 f(x)\,dx$ (določeni integral od 0 do 1).

Več bomo o eksponentih povedali kasneje, ko jih bomo obravnavali kot podatkovne tipe v programskem jeziku.

5.3 Podatkovni tipi

V programskem jeziku ne govorimo o množicah, ampak o **tipih**, ki so na prvi pogled podobni množicam, a z njimi ne izražamo le »skupkov stvari«, ampak *načine konstrukcij* matematičnih objektov in podatkov. Tipi so zelo splošen in uporaben koncept, ki presega meje programiranja in celo računalništva. Uporabljajo se tudi v logiki in drugih vejah matematike.

Programski jeziki lahko podpirajo tipe v večji ali manjši meri. V λ-računu ni nobenih tipov, C jih ima, prav tako Java. Če je `e` izraz tipa `T`, to zapišemo

```
e : T
```

Zapis spominja na $e \in T$ iz teorije množic. Poudarimo še enkrat, da na tip `T` ne gledamo kot na zbirko elementov, ampak kot na informacijo o tem, kakšen podatek je `e` in kakšna je njegova struktura.

5.3.1 OCaml

Konstrukcije množic, ki smo jih spoznali, bomo predelali v konstrukcije tipov. V ta namen potrebujemo primer programskega jezika, ki le-te neposredno podpira. Izbrali bomo **OCaml**. Lahko bi uporabili tudi SML, Haskell, Idris, Elm in še marsikaterega drugega. C/C++, Java, Python in Javascript ne podpirajo konstrukcij, ki jih bomo obravnavali, lahko jih le bolj ali manj uspešno simuliramo.

V OCaml se imena tipov piše z malo začetnico, zato bomo za imena tipov uporabljali male črke.

Spletni viri za OCaml:

- Uradna spletna stran za OCaml
- REPL za OCaml na glot.io (kaj pomeni REPL?)
- Poglavje Funkcijsko programiranje v zapiskih Matije Pretnarja
- Učbenik OCaml from the ground up

5.3.2 Zmnožek tipov

Zmnožek tipov ali **kartezični produkt** `a * b` tipov `a` in `b` vsebuje urejene pare, ki jih v OCaml zapišemo enako, kot v matematiki:

```
OCaml version 4.12.0
```

```
# (1 + 2, "banana") ;;
- : int * string = (3, "banana")
```

Zapisali smo urejeni par `(1 + 2, "banana")`. OCaml je ugotovil, da je tip tega urejenega para `int * string` in to izpisal, skupaj z izračunano vrednostjo.

Na koncu vsakega ukaza moramo napisati `;;`. Človek se sčasoma navadi na vse.

Tvorimo lahko urejene n-terice, za poljuben $n \geq 0$:

```
# (1, "banana", false, 2) ;;
- : int * string * bool * int = (1, "banana", false, 2)
```

Projekciji π_1 in π_2 v OCamlu zapišemo `fst` in `snd` (okrajšavi za »first« in »second«):

```
# fst (1, "banana") ;;
- : int = 1
# snd (1, "banana") ;;
- : string = "banana"
```

Ti dve projekciji delujeta samo na urejenih parih! Na primer, s `fst` ne moremo projecirati prve komponente urejene trojice:

```
# fst (1, "banana", false) ;;
Error: This expression has type 'a * 'b * 'c
       but an expression was expected of type 'd * 'e
```

To je tako, ker v OCamlu redko uporabljamo projekcije, saj so dosti bolj prikladno in bolj splošno prirejanje z vzorci, ki ga bomo kmalu obravnavali.

5.3.3 Definicije vrednosti

Če želimo vpeljati definicijo, to naredimo z `let`:

```
# let i = 10 + 3 ;;
val i : int = 13

# let j = 100 + i * i ;;
val j : int = 269
```

Pozor! Zgoraj *nismo* definirali »spremenljivke `i`« ampak **vrednost** `i`, ki je *ne moremo spreminjati*. OCaml izračuna vrednost `13` in jo priredi `i`, ki se ga ne da spremeniti. To je tako, kot če bi v Javi povsod pisali `final` pred deklaracije spremenljivk in je zelo dobra ideja.

Če naredimo tole:

```
# let x = 2 ;;
val x : int = 2
# let x = 3 ;;
val x : int = 3
```

nismo spremenili `x`, ampak smo definirali *nov* `x`, ki je *prekril* staro definicijo. Če želimo pravo spremenljivko, v ta namen uporabimo referenco (o tem več kasneje).

> **Naloga**
>
> Predavatelja poskusite prepričati, da so »ne-spremenljivke« slaba ideja.

Poznamo tudi **lokalne definicije** vrednosti, ki jih pišemo

```
let p = e₁ in e₂
```

Tu je `p` **vzorec**, `e₁` in `e₂` pa sta izraza. Vzorec je sestavljen iz konstruktorjev, imen in anonimnega vzorca `_`. Vrednost izraza `e₁` se primerja z vzorcem `p` in sestavni deli vrednosti se priredijo simbolom. Primeri:

```
# let x = 3 * 14 ;;
val x : int = 42

# let (a, b) = (1 + 2, 3 + 4) ;;
val a : int = 3
val b : int = 7

# let (x, (y, z)) = (1, (2, 3)) ;;
val x : int = 1
val y : int = 2
val z : int = 3

# let (r, _, q) = (false, "foo", 8) ;;
val r : bool = false
val q : int = 8
```

S pomočjo vzorcev lahko definiramo tudi ostale projekcije (tretja, četrta, peta, ...), a te v praksi malokdaj pridejo prav, saj so vzorci dosti bolj uporabni:

```
# let thd (_, _, z) = z ;;
val thd : 'a * 'b * 'c -> 'c = <fun>
# thd (1, "banana", false) ;;
- : bool = false
```

5.3.4 Enotski tip

Če smemo pisati urejene pare, trojice, četverice, ..., ali smemo zapisati tudi »urejeno ničterico«? Seveda!

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

Dobili smo **enotski tip** `unit`. To je tip, ki ima en sam element, namreč urejeno ničterico `()`, ki ji pravimo **enota**. Zakaj se mu reče »enotski«? Ker je množica z enim elementom »enota za množenje« (matematiki namesto `unit` pišejo kar $1 = \{\star\}$):

$$A \cong 1 \times A \tag{5.4}$$

Morda se zdi enotski tip neuporaben, a to ni res. V C in Java so ta tip poimenovali `void` (»prazen“) in se ga uporablja za funkcije, ki ne vračajo rezultata. Tip `void` sploh ni prazen, ampak ima en sam element, ki pa ga programer nikoli ne vidi (in ga tudi ne more). Če namreč funkcija vrača v naprej predpisan element, potem vemo, kaj bo vrnila, in tega ni treba razlagati.

Zapomnimo si torej, da funkcija, ki »ne vrne ničesar« v resnici vrne `()`. V OCaml se to dejansko vidi, v Javi in C pa ne.

Kaj pa funkcija, ki »ne sprejme ničesar“? Če funkcija sprejme argumente `x`, `y` in `z`, potem sprejme urejeno trojico. Če ne sprejme ničesar, potem v resnici sprejme urejeno ničterico `()`, torej spet enoto.

Pa še to: morda ste si kdaj želeli, da bi lahko v C ali Java brez velikih muk napisali funkcijo, ki vrne dva rezultata? Jezik, ki ima zmnožke, to omogoča sam od sebe: preprosto vrnete urejeni par!

5.3.5 Zapisi

Urejeni pari včasih niso prikladni, ker si moramo zapomniti vrstni red komponent. Na primer, polno ime osebe bi lahko predstavili z urejenim parom `("Mojca", "Novak")`, a potem moramo vedno paziti, da ne zapišemo pomotoma `("Novak", "Mojca")`. Težava nastopi tudi, ko imamo kompliciranе podatke. Na primer, podatke o trenutnem času bi lahko predstavili z naborom

$$(\mathit{leto}, \mathit{mesec}, \mathit{dan}, \mathit{ura}, \mathit{minuta}, \mathit{sekunda}, \mathit{milisekunda}) \tag{5.5}$$

Kdo si bo zapomnil, da so minute peto polje in milisekunde sedmo?

Težavo razrešimo tako, da komponent ne štejemo po vrsti, ampak jih poimenujemo. Dobimo tako imenovani tip *zapis* (angl. *record*). Najprej ga definiramo z deklaracijo `type`:

```
type oseba = { ime : string; priimek : string; }
```

S tem smo uvedli nov tip `oseba`, ki je zapis z dvema poljema. Sedaj lahko namesto urejenega para tvorimo zapis:

```
# { ime = "Mojca"; priimek = "Pokraculja" } ;;
- : oseba = {ime = "Mojca"; priimek = "Pokraculja"}
```

Torej je $\{\ell_1{=}e_1; \ldots; \ell_i{=}e_i\}$ kot nabor $(e_1, \ldots, e_i)$, le da smo poimenovali njene komponente $\ell_1, \ldots, \ell_i$.

Težave z vrstnim redom izginejo, ker je v zapisu pomembno ime komponente in ne vrstni red:

```
# { priimek = "Pokraculja"; ime = "Mojca" } ;;
- : oseba = {ime = "Mojca"; priimek = "Pokraculja"}
```

Z zapisom lahko zapišemo tudi urejeno »enerico«:

```
# type zajec = { masa : int } ;;
type zajec = { masa : int; }
```

Sedaj lahko tvorimo zapis z enim samim poljem:

```
# { masa = 42 } ;;
- : zajec = {masa = 42}
```

V Pythonu se to zapiše `("42",)`.

Do polja z imenom `foo` v zapisu `s` dostopamo s `s.foo`:

```
# let mati = { ime = "Neza"; priimek = "Cankar" } ;;
val mati : oseba = {ime = "Neza"; priimek = "Cankar"}
# mati.ime ;;
- : string = "Neza"
# mati.priimek ;;
- : string = "Cankar"
```

Do polj zapisa lahko dostopamo tudi z vzorci:

```
# let {ime = i; priimek = p} = mati ;;
val i : string = "Neza"
val p : string = "Cankar"
```

Polj, ki nas ne zanimajo, v vzorcu ni treba omenjati, lahko le uporabimo _:

```
# let {ime = i; priimek = _} = mati ;;
val i : string = "Neza"
```

Če ignoriramo več polj, lahko za vse skupaj uporabimo _:

```
# let {ime = i; _} = mati ;;
val i : string = "Neza"
```

Pogosto poimenujemo vrednosti enako kot polja:

```
# let {ime = ime; priimek = priimek} = mati ;;
val ime : string = "Neza"
val priimek : string = "Cankar"
```

Za take primere OCaml podpira sintaktični sladkorček:

```
# let {ime; priimek} = mati ;;
val ime : string = "Neza"
val priimek : string = "Cankar"
```

```
# let {ime; _} = mati ;;
val ime : string = "Neza"
```

5.3.6 Definicije tipov

Videli smo že, da lahko s `type a = ...` definiramo zapise:

```
type complex = { re : float; im : float }

type datetime = { year : int
                ; month : int
                ; hour : int
                ; minute : int
                ; second : int
                ; millisecond : int
                }

type color = { red : float; green : float; blue : float }
```

Definiramo lahko tudi okrajšave za tipe, na primer:

```
type krneki = int * bool * string
```

Sedaj lahko namesto `int * bool * string` pišemo `krneki`.

5.3.7 Vsota tipov

Elemente vsote množic $A + B$ smo označevali z ι_1 in ι_2. Izbor oznak je z matematičnega stališča nepomemben, namesto ι_1 in ι_2 bi lahko pisali tudi kaj drugega. V programiranju bomo to seveda izkoristili: tako kot smo uvedli zapise, ki so pravzaprav zmnožki s poimenovanimi komponentami, bomo uvedli vsote tipov, pri katerih si oznake izbere programer.

Če želimo imeti vsoto, jo moramo v OCaml najprej definirati s `type`, tako kot zapise. Zgornji primer izdelkov v spletni trgovini, bi zapisali takole:

```
type barva = { blue : float; green : float;  red : float }

type izdelek =
  | Cevelj of barva * int
  | Palica of int
  | Posoda of int
```

Ta definicija pravi, da je `izdelek` vsota treh tipov: prvi tip je zmnožek tipov `barva` in `int`. Drugi in tretji tip sta oba `int`. Za oznake smo izbrali `Cevelj`, `Palica` in `Posoda`. Tem oznakam v Ocaml pravimo **konstruktorji** (angl. constructor).

Črn čevelj velikosti `42` zapišemo

```
Cevelj ({blue=0.0; green=0.0; red=0.0}, 42)
```

palico velikosti `7`

```
Palica 7
```

in posodo s prostornino `7`

```
Posoda 7
```

5.3.8 Razločevanje primerov

Kot smo omenili, potrebujemo zapis za *razločevanje primerov*. Nadaljujmo s primerom. Denimo, da je cena izdelka `z` določena takole:

- čevelj stane 15 evrov, če je številka manjša od 25, sicer stane 20 evrov
- palica dolžine `x` stane `1 + 2 * x` evrov
- posoda stane 7 evrov ne glede na prostornino

To v Ocaml zapišemo z `match`:

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

Splošna oblika stavka `match` je

```
match e with
  | p₁ -> e₁
  | p₂ -> e₂
  | p₃ -> e₃
    ⋮
  | pᵢ -> eᵢ
```

Tu so p_1, ..., p_i **vzorci**. Vrednost izraza `match ...` je prvi e_j, za katerega `e` zadošča vzorcu p_j. V OCaml je `match` dosti bolj uporaben kot `switch` v C in Javi ali `if … elif … elif …` v Pythnu, ker OCaml izračuna, ali smo pozabili obravnavati kakšno možnost. Primer:

```
# match (Palica 7) with
         | Cevelj (b, v) -> if v < 35 then 15 else 25
         | Posoda y -> 7 ;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Palica _
Exception: Match_failure ("//toplevel//", 29, -65).
```

Včasih želimo uvesti tip, ki sestoji iz končnega števila konstant. To lahko naredimo z vsoto takole:

```
type t = Foo | Bar | Baz | Qux
```

V C je to tip `enum`, podobno v Javi. Imena konstruktorjev se morajo pisati z veliko začetnico. V OCaml bi lahko `bool` definirali sami, če ga še ne bi bilo:

```
type bool = False | True
```

Vzorci v stavku `match` so lahko poljubno gnezdeni. Denimo, da bi želeli ceno izračunati takole:

- čevelj stane 15 evrov, če je številka manjša od 25, sicer stane 20 evrov
- palica dolžine 42 stane 1000 evrov
- palica dolžine `x` $\neq$ 42 stane `1 + 2 * x` evrov
- posoda stane 7 evrov ne glede na prostornino

Pripadajoči stavek `match` se glasi:

```
match z with
  | Cevelj (b, v) -> if v < 35 then 15 else 25
  | Palica 42 -> 1000
  | Palica x -> 1 + 2 * x
  | Posoda y -> 7
```

Vzorce lahko uporabljamo tudi v definicijah vrednosti `let`:

```
# let (Posoda p) = Posoda 10 ;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Palica _
val p : int = 10

# let Cevelj (x, y) = Cevelj ({red=1.0; green=0.5; blue=0.0}, 43) ;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
(Palica _|Posoda _)
val x : barva = {blue = 0.; green = 0.5; red = 1.}
```

```
val y : int = 43

# let Cevelj ({red=r;green=_;blue=b},v) = Cevelj ({red=1.0; green=0.5;
blue=0.0}, 43) ;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
(Palica _|Posoda _)
val b : float = 0.
val r : float = 1.
val v : int = 43
```

5.3.9 Tip funkcij

Tip funkcij `a -> b` zajema funkcije, ki sprejmejo argument tipa `a` in vrnejo rezultat tipa `b`. V OCaml λ-abstrakcijo $\lambda x\,.\,e$ zapišemo kot `fun x -> e`:

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

Izračunana vrednost je funkcija, ki jo OCaml izpiše kot `<fun>`. (Kaj pa bi lahko naredil drugega, izpisal prevedeno kodo, ki predstavlja funkcijo?)

Veljajo podobna pravila kot v λ-računu. Na primer funkcije lahko gnezdimo:

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

OCaml je izračunal tip funkcije `int -> int -> int`. Operator `->` je *desno asociativen* , `a -> b -> c` je enako `a -> (b -> c)`.

Tip `int -> int -> int` torej opisuje funkcije, ki sprejmejo `int` in vrnejo `int -> int`. Funkcije lahko tudi uporabljamo:

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

Ste razumeli, kaj naredi zgornji primer? Kaj pa tale:

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

Funkcijo lahko poimenujemo:

```
# let f = fun x -> x * x + 1 ;;
val f : int -> int = <fun>
# f 10 ;;
- : int = 101
```

Namesto `let f = fun x -> ⋯` lahko pišemo tudi `let f x = ⋯`

```
# let g x = x * x + 1 ;;
val g : int -> int = <fun>
# g 10 ;;
- : int = 101
```

Definicija funkcije je rekurzivna, če to naznanimo `let rec`:

```
# let rec fact n = (if n = 0 then 1 else n * fact (n - 1)) ;;
val fact : int -> int = <fun>
# fact 10 ;;
- : int = 3628800
```

V telesu funkcije smo uporabili pogojni stavek, a se v OCamlu bolje obnese `match`:

```
let rec fact n =
  match n with
  | 0 -> 1
  | n -> n * fact (n - 1)
```

Sintagma `fun x -> match x with` p_1 `->` e_1 `| ...` je pogosta in ju lahko nadomestimo s `function` p_1 `->` e_1 `| ..`:

```
let rec fact = function
  | 0 -> 1
  | n -> n * fact (n - 1)
```

Kot vidimo, OCaml sam izračuna tip funkcije. Pravzaprav vedno sam izračuna vse tipe. Pravimo, da tipe *izpelje* in s tem se bomo še posebej ukvarjali. Včasih kak tip ostane nedoločen, na primer:

```
# fun (x, y) -> (y, x) ;;
- : 'a * 'b -> 'b * 'a = <fun>
```

Tip `x` je poljuben, prav tako tip `y`. Ocaml ju zapiše z `'a` in `'b`. Znak apostrof označuje dejstvo, da sta to *poljubna* tipa, ali *parametra*. Še en primer:

```
# fun (x, y, z) -> (x, y + z, x) ;;
- : 'a * int * int -> 'a * int * 'a = <fun>
```

Ko zapišemo funkcijo, lahko podamo tip njenih argumentov:

```
# fun (x : string) -> x ;;
- : string -> string = <fun>
```

Brez oznake tipa OCaml izpelje najbolj splošen tip:

```
# fun x -> x ;;
- : 'a -> 'a = <fun>
```

6. Rekurzija in rekurzivni tipi

Rekurzija je osnovni koncept v logiki, matematiki in računalništvu. V tej lekciji bomo spoznali vlogo rekurzije v programiranju in programskih jezikih.

6.1 Rekurzija in negibne točke

Pravimo, da je funkcija *rekurzivna*, kadar kliče sama sebe. Kot primer vzemimo funkcijo `f`, ki računa faktorielo. V Javi bi jo zapisali takole:

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

Ekvivalentna definicija v Pythonu:

```
def f(n):
  if n == 0:
      return 1
  else:
      return n * f(n -1)
```

Ekvivalentna definicija v OCamlu:

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

Ekvivalenta definicija v Haskellu:

```
f :: Integer -> Integer
f n = if n == 0 then 1 else n * f (n - 1)
```

Obravnavajmo verzijo v Haskellu. Funkcijo prepišemo takole:

```
f :: Integer -> Integer
f = \n -> if n == 0 then 1 else n * f (n - 1)
```

Sedaj definicijo razstavimo na dva dela: na *telo* rekurzije, ki samo po sebi ni rekurzivno, in na *rekurzivni sklic* funkcije `f` same nase:

```
telo :: (Integer -> Integer) -> (Integer -> Integer)
telo self =  \n -> if n == 0 then 1 else n * self (n - 1)

f :: Integer -> Integer
f = telo f
```

Pravimo, da smo **razprli** rekurzivno zanko.

Rekurzivno funkcijo `f` smo zapisali kot negibno točko funkcije `telo`.

Definicija (negibna točka)

Negibna točka funkcije $h : X \to X$ je tak $x \in X$, da velja $x = h(x)$.

V našem primeru je h funkcija `telo`, X je tip `Integer -> Integer` in x je `f`. Negibne točke so pomembne tudi na drugih področjih matematike in o njih matematiki veliko vedo.

Primer

V numeričnih metodah enačbo oblike $x = h(x)$ poiščemo z zaporedjem približkov

$$x_0, h(x_0), h(h(x_0)), h(h(h(x_0))), \ldots \tag{6.1}$$

Če imamo srečo (kar pomeni, da je absolutna vrednost odvoda h v okolici negibne točke manjša od 1 in je x_0 v taki okolici), zaporedje konvergira k negibni točki. Na primer, enačbo $x^2 = 1/2$ prepišemo v obliko $x = 1/2 - x^2 + x$, se pravi $x = h(x)$, kjer je $h(x) = 1/2 - x^2 + x$. Če vzamemo $x_0 = 1$, dobimo zaporedje

$$1.0, 0.5, 0.75, 0.6875, 0.714844, 0.703842, \ldots \tag{6.2}$$

ki konvergira k rešitvi enačbe $1/\sqrt{2} = 0.70710678118654752440$.

Ali je tudi rekurzivna funkcija dveh argumentov negibna točka, na primer funkcija, ki računa binomske koeficiente?

```
binom :: (Integer, Integer) -> Integer
binom (n, k) = if k == 0 || k == n then 1 else binom (n - 1, k - 1) + binom (n
- 1, k)
```

Seveda, saj lahko `binom` še vedno razstavimo na telo funkcije in sklic samega nase:

```
telo :: ((Integer, Integer) -> Integer) -> ((Integer, Integer) -> Integer)
telo self = \(n, k) -> if k == 0 || k == n then 1 else self (n - 1, k - 1) +
self (n - 1, k)

binom :: (Integer, Integer) -> Integer
binom = telo binom
```

Kaj pa definicija rekurzivnih funkcij `f` in `g`, ki kličeta druga drugo?

Na primer, obravnavajmo funkcijo `f`, ki kliče `f` in `g`, ter funkcijo `g`, ki kliče `f`:

```
f x = if x == 0 then 1 + f (x - 1) else 2 + g (x - 1)
g y = if y == 0 then 1 else 3 * f (y - 1)
```

Če ju združimo v urejeni par `(f, g)` in ju zapišemo z λ-računom, dobimo

```
(f, g) = ((λ x . if x == 0 then 1 + f (x - 1) else 2 + g (x - 1)),
          (λ y . if y == 0 then 1 else 3 * f (y - 1)))
```

To je *rekurzivna definicija urejenega para (funkcij)*, kar prepišemo v

```
(f, g) = t (f, g)
```

kjer je

```
t = λ (f', g') . ((λ x . if x = 0 then 1 + f' (x - 1) else 2 + g' (x - 1)),
                  (λ y . if y = 0 then 1 else 3 * f' (y - 1)))
```

Torej tudi za hkratne rekurzivne definicije velja, da so to negibne točke.

6.2 Operator `fix`

Recept za rekurzivno funkcijo je vedno isti:

1. Zapišemo telo `t` funkcije.
2. Definiramo negibno točko `f = t f`.

Pri tem je samo drugi korak rekurziven in vedno enak. Zapišemo ga lahko kot funkcijo `fix`, ki izračuna negibno točko dane funkcije:

```
fix :: (a -> a) -> a
fix t = t (fix t)
```

V Haskellu tip `(a -> a) -> a` pomeni »funkcija, ki sprejme funkcjo tipa `a -> a` in vrne vrednost tipa `a`. Pri tem je tip `a` poljuben, pravimo, da je *parameter*.

Vso rekurzijo smo "spravili" v `fix`. Od tu naprej bi lahko rekurzivne funkcije definirali s pomočjo `fix`:

```
f :: Integer -> Integer
f = fix (\ self n -> if n == 0 then 1 else n * self (n - 1))
```

Naloga

Katero vrednost zavzame tip `a` iz definicije `fix` v zgornji definiciji `f`?

Poglejmo postopek še enkrat, tokrat zapisan z λ-računom:

1. Prvotna definicija `f` se glasi: `f n = if n = 0 then 1 else n * f (n - 1)`
2. Zapišemo s pomočjo λ-abstrakcije: `f = λ n . if n = 0 then 1 else n * f (n - 1)`
3. Ločimo rekurzijo in telo funkcije: `f = t f` kjer je `t = (λ g n . if n = 0 then 1 else n * g (n - 1))`
4. S pomočjo `fix` definiramo `f`: `f = fix t`

6.3 Iteracija je poseben primer rekurzije

V ukaznem programiranju poznamo zanke, na primer zanko `while`:

```
while b do c done
```

Tudi ta je negibna točka! Res, taka zanka je ekvivalentna svojemu »odvitju«

```
if b then (c ; while b do c done) else skip
```

Če pišemo `W` za našo zanko, dobimo:

```
W ≡ (if b then (c ; W) else skip)
```

Torej je `W` negibna točka funkcije

```
t = (λ V . if b then (c ; V) else skip)
```

saj velja

```
(while b do c done) = t (while b do c done)
```

Primer

Zanko `while` lahko na zgornji način »odvijamo v nedogled«:

Odvitje 0:

```
while b do c done
```

Odvitje 1:

```
if b
then
  (c ; while b do c done)
else skip
```

Odvitje 2:

```
if b
then
  (c ;
   if b
   then
     (c ; while b do c done)
   else skip
  )
else skip
```

Faza 3:

```
if b
then
  (c ;
   if b
   then
     (c ;
      if b
      then
        (c ; while b do c done)
      else skip
     )
   else skip
  )
else skip
```

In tako naprej. Če bi lahko imeli neskončno programsko kodo, ne bi potrebovali zank!

6.4 Rekurzivni seznami

Rekurzivno lahko definiramo tudi razne druge strukture, ne samo funkcij. Na primer, neskončni seznam

```
ℓ = [1, 2, 1, 2, 1, 2, ...]
```

lahko v Haskellu definiramo rekurzivno:

```
ℓ = 1 : 2 : ℓ
```

Primer

Še bolj zanimiv primer rekurzivno definiranega seznama:

```
fibs = 0 : 1 : zipWith (+) fibs (drop 1 fibs)
```

Ugotovite, kaj počneta `zipWith` in `drop` in nato razložite, kakšen seznam je to.

6.5 Rekurzivni tipi

V tem razdelku bomo spet delali z OCamlom, kasneje pa v Haskellu. Do sedaj smo spoznali podatkovne tipe:

- produkt `a * b` in zapisi
- vsota `a + b`
- eksponent `a → b`

S temi konstrukcijami ne moremo dobro predstaviti bolj naprednih podatkovnih tipov, kot so seznami in drevesa. Poglejmo na primer, kako se tvori sezname celih števil:

- prazen seznam: `[]` je seznam
- sestavljen seznam: če je `x` celo število in `ℓ` seznam, je tudi `x :: ℓ` seznam

Zapis `[1; 2; 3]` je okrajšava za `1 :: (2 :: (3 :: []))`.

Seznami so **rekurzivni podatkovni tip**, saj gradimo sezname iz seznamov. Brez uporabe posebnih oznak `[]` in `::` bi zgornjo definicijo zapisali takole (oznaki `Nil` in `Cons` izhajata iz programskega jezika LISP, kjer pišemo `nil` in `(cons x ℓ)`):

- prazen seznam: `Nil` je seznam
- sestavljen seznam: če je `x` celo število in `ℓ` seznam, je tudi `Cons (x, ℓ)` seznam

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

V OCamlu se tako definicijo zapiše takole:

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

Spet imamo opravka z rekurzijo. Tipi, ki se sklicujejo sami nase v svoji definiciji, se imenujejo **rekurzivni tipi**.

In spet vidimo, da je rekurzija negibna točka. Podatkovni tipi `seznam` je negibna točka za preslikavo `T`, ki slika tipe v tipe:

```
seznam = T (seznam)
```

kjer je `T` definiran kot

```
T a = (Nil | Cons of int * a)
```

Z besedami: `T` je funkcija, ki sprejme poljuben tip `a` in vrne vsoto tipov `Nil | Cons of int * a`.

6.5.1 Induktivni tipi

Izhajamo iz definicije seznama:

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

Vprašajmo se: ali ta definicija zajema neskončne sezname? Na primer:

```
Cons (1, Cons (2, Cons (3, Cons (4, Cons (5, ...)))))
```

Ali se mora to kdaj zaključiti z `Nil`? Možna sta dva odgovora. Če zahtevamo, da morajo biti elementi rekurzivnega tipa končni, govorimo o *induktivnih* tipih. Če pa dovolimo neskončne elemente, govorimo o *koinduktivnih* tipih.

Poglejmo najprej **induktivne podatkovne tipe**. To so rekurzivni tipi, v katerih vrednosti sestavljamo začenši z osnovnimi s pomočjo konstruktorjev in neskončne vrednosti niso dovoljene. Primeri:

1. naravna števila
2. končni seznami
3. končna drevesa
4. abstraktna sintaksa jezika:
   - programski jeziki
   - jeziki za označevanje podatkov
5. hierarhija elementov v uporabniškem vmesniku

Primer

Definicija naravnega števila:

- `0` je naravno število
- če je `n` naravno število, je tudi n^+ naravno število (ki mu rečemo »naslednik `n`«)

Definicija podatkovnega tipa:

```
type stevilo = Nic | Naslednik of stevilo
```

Ta definicija ni učinkovita, ker predstavi naravna števila z naslednikom, torej v »eniškem« sistemu. Naravna števila bi lahko definirali tudi takole:

- `0` je naravno število
- če je `n` naravno število, je tudi `Shl0 n` naravno število
- če je `n` naravno število, je tudi `Shl1 n` naravno število

Oznaka `Shl` je okrajšava za »shift left«. S `Shl0 n` predstavimo število $2 \cdot n + 0$ in s `Shl1 n` število $2 \cdot n + 1$. Na primer

```
Shl0 (Shl1 (Shl0 (Shl1 0)))
```

je število `10`. Kot podatkovni tip:

```
type stevilo = Zero | Shl0 of stevilo | Shl1 of stevilo
```

Vendar to še vedno ni optimalna rešitev, ker lahko število nič predstavimo na neskončno načinov:

```
0 = Shl0 0 = Shl0 (Shl0 0) = Shl0 (Shl0 (Shl0 0)) = ⋯
```

Naloga

Poiščite predstavitev dvojiških števil z induktivnimi tipi (lahko jih je več), da bo imelo vsako nenegativno celo število natanko enega predstavnika.

Primer

Standard za predstavitev podatkov JSON se kot podatkovni tip glasi takole:

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

6.5.2 Splošni rekurzivni tipi

Rekurzivni tipi so lahko zelo nenavadni:

```
type d = Foo of (d -> bool)
```

Vrednost tipa `d` je oblike `Foo f`, kjer je `f` funkcija iz `d` v `bool`. Ali znate zapisati kako tako vrednost?

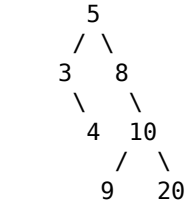
6.5.3 Strukturna rekurzija

Ker so induktivni podatkovni tipi definirani rekurzivno, jih običajno obdelujemo z rekurzivnimi funkcijami. Kot primer si oglejmo, kako bi implementirali iskalna drevesa.

Obravnavajmo preprosta **iskalna drevesa**, v katerih hranimo cela števila. Iskalno drevo je

- bodisi **prazno**
- bodisi **sestavljeno** iz korena, ki je označen s številom `x`, ter dveh poddreves `l` in `r` pri čemer velja:
  - vsa števila v vozliščih `l` so manjša od `x`,
  - vsa števila v vozliščih `r` so večja od `x`

Primer drevesa:



Podatkovni tip v OCaml se glasi:

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

V tipu *nismo* shranili informacije o tem, da je iskalno drevo urejeno! Če bo programer ustvaril iskalno drevo, ki ni pravilno urejeno, prevajalnik tega ne bo zaznal.

> **Naloga**
>
> Sestavite funkcije za iskanje, vstavljanje in brisanje elementov v iskalnem drevesu.

6.6 Koinduktivni tipi

Poznamo še en pomembno vrsto rekurzivnih tipov, to so **koinduktivni tipi**. Pojavljajo se v računskih postopkih, ki so po svoji naravi lahko neskončni.

Tipičen primer koinduktivnega tipa je **komunikacijski tok podatkov:**

- bodisi je tok podatkov prazen (komunikacije je konec)
- bodisi je na voljo sporočilo `x` in preostanek toka

Primere take komunikacije najdemo povsod, kjer program komunicira z okoljem ali z drugim programom: odjemalec s strežnikom, dva programa med seboj, program z uporabnikom ipd.

Če preberemo zgornjo definicijo kot induktivni tip, se ne razlikuje od definicije seznamov. To bi pomenilo, da bi moral biti komunikacijski tok vedno končen, kar je nespametna predpostavka. V praksi seveda komunikacija ni *dejansko* neskončna, a je *potencialno* neskončna, kar pomeni, da lahko dva procesa komunicirata v nedogled in brez vnaprej postavljene omejitve.

Koinduktivni tipi so rekurzivni tipi, ki dovoljujejo tudi neskončne vrednosti. Vendar pozor, kadar imamo opravka z neskončno velikimi seznami, drevesi itd., moramo paziti, kako z njimi računamo. Izogniti se moramo temu, da bi neskončno veliko drevo ali komunikacijski tok poskušali izračunati v celoti do konca.

Haskell ima koinduktivne podatkovne tipe.

6.6.1 Tokovi

Poglejmo različico tokov, ki so vedno neskončni, ker pri njih koinduktivna narava pride še bolj do izraza. Tok je:

- sestavljen iz sporočila in preostanka toka

Če to definicijo preberemo induktivno, dobimo *prazen* tip, saj ne moremo začeti. Res, če zapišemo v OCaml

```
type 'a stream = Cons of 'a * 'a stream
```

dobimo podatkovni tip, ki nima nobene končne vrednosti. Vrednost bi bila nujno neskončna, na primer:

- `Cons (1, Cons (2, Cons (3, Cons (4, …))))`
- `Cons (1, Cons (1, Cons (1, Cons (1, …))))`

OCaml sicer dopušča *nekatere* take vrednosti z definicijo `let rec`:

```
# let rec s = Cons (1, Cons (2, s)) ;;
val s : int stream = Cons (1, Cons (2, <cycle>))
```

A te vrednosti le pokvarijo induktivno naravo tipov, hkrati pa ne dovoljujejo poljubnih neskončnih vrednosti. Če bi imele v praksi uporabno vrednost, bi jih morda tolerirali, ker pa jih redkokdaj uporabimo, smo lahko nekoliko razočarani nad odločitvijo snovalcev OCamla, da jih dovolijo.

6.6.2 Tokovi v Haskellu

Ista definicija v Haskellu deluje, ker ima Haskell koinduktivne tipe.

V Haskellu podatkovne tipe pišemo nekoliko drugače:

- imena tipov se piše z velikimi začetnicami: `Bool`, `Integer`, ...
- produkt tipov `a` in `b` zapišemo `(a,b)`, se pravi tako kot urejene pare. Na primer, elementi tipa `(Bool, Int)` so `(False, 0)`, `(False, 42)`, `(True, 23)` itd.
- enotski tip pišemo `()`, torej tako kot njegovo edino vrednost.
- zapis `e :: t` pomeni »`e` ima tip `t`«, zapis `e : t` pa seznam z glavo `e` in repom `t` (ravno obratno, kot v OCamlu)
- podatkovni tip uvedemo z določilom `data` in parameter pišemo za ime tipa z malimi črkami. Torej namesto `'a stream` zapišemo `Stream a`.

A to so le podrobnosti konkretne sintakse.

Poglejmo definicijo tokov in njihovo uporavo na preprostih primerih:

```
-- neskončen podatkovni tok elementov tipa a
data Stream a = Cons a (Stream a)

-- Prvih n elementov toka pretvori v seznam
to_list :: Integer -> Stream a -> [a]
to_list 0 _ = []
to_list n (Cons x s) = x : (to_list (n-1) s)

-- rep podatkovnega toka
rest :: Stream a -> Stream a
rest (Cons _ s) = s

-- konstantni podatkovni tok, ki vedno vrača x
constant :: a -> Stream a
constant x = Cons x (constant x)

-- uporabi funkcijo na vseh elementih toka
streamMap :: (a -> b) -> Stream a -> Stream b
streamMap f (Cons x s) = Cons (f x) (streamMap f s)

-- spni dva tokova z dano funkcijo
streamZip :: (a -> b -> c) -> Stream a -> Stream b -> Stream c
streamZip f (Cons x s) (Cons y t) = Cons (f x y) (streamZip f s t)

-- filtriraj tok z dano Boolovo funkcijo
streamFilter :: (a -> Bool) -> Stream a -> Stream a
streamFilter p (Cons x s) | p x       = Cons x $ streamFilter p s
                          | otherwise = streamFilter p s

-- Tok naravnih števil
natural = Cons 0 $ streamMap (+ 1) natural

-- Tok števil od n naprej
naturalFrom :: Integer -> Stream Integer
naturalFrom n = Cons n $ naturalFrom (n + 1)

-- Kvadrati naravnih števil
squares = streamMap (\ n -> n * n) natural

-- Fibonaccijeva števila
fibonacci = Cons 0 $ Cons 1 $ streamZip (+) fibonacci (rest fibonacci)

-- Eratostenovo sito in praštevila
erastoten :: Stream Integer -> Stream Integer
erastoten (Cons k s) =
    Cons k $ erastoten (streamFilter (\ n -> n `mod` k /= 0) s)

primes :: Stream Integer
primes = erastoten $ naturalFrom 2
```

6.6.3 Tokovi v OCamlu

V OCaml lahko *simuliramo* tokove z uporabo tehnike *zavlačevanja* (angl. »thunk«).

Denimo da imamo izraz `e` tipa `t`, ki ga zaenkrat še ne želimo izračunati. Tedaj ga lahko predelamo v funkcijo `fun () -> e` (po angleško se imenuje taka funkcija *thunk*), ki je tipa `unit -> t`. Ker je `e` znotraj telesa funkcije, se bo izračunal šele, ko funkcijo uporabimo na `()`. Od tod dobimo idejo, kako bi predstavili t.i. lene vrednosti v OCaml:

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

Primeri uporabe:

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

(* Neskončni tok enic *)
let enice =
  let rec e () = Cons (1, e) in
  e ()

(* Neskončni tok k, k+1, k+2, ... *)
let rec count k = Cons (k, (fun () -> count (k+1)))

(* Neskončni tok enic *)
let rec enice =
  let rec generate () = Cons (1, generate) in
  generate ()

(* Prvih n elementov toka pretvori v seznam *)
let rec to_list k s =
  match (k, s) with
  | (0, _) -> []
  | (n, Cons (x, s)) -> x :: to_list (n-1) (s ())

(* n-ti element toka *)
let rec elementAt k s =
  match (k, s) with
  | (0, Cons (x, _)) -> x
  | (n, Cons (_, s)) -> elementAt (n-1) (s ())

(* Potenčne vrste. Tok a₀, a₁, a₂, ... predstavlja potenčno vrsto

      a₀ + a₁ x + a₂ x² + a₃ x³ + ...
 *)

(* Izračunaj vrednost potenčne vrste s v točki x, uporabi prvih n členov *)
let rec eval n x s =
  let rec loop k xpow v (Cons (a, t)) =
    match k with
    | 0 -> v
    | k ->  loop (k-1) (xpow *. x) (v +. a *. xpow) (t ())
  in
  loop n 1.0 0.0 s

(* V pythonu:

   def eval (n, x, s):
     k = n
     xpow = 1.0
     v = 0
     while k > 0:
       a = s.getNext()
       v = v + a * xpow
       xpow = xpow * x
       k = k - 1
     return v
```

```
 *)

(* Potenčna vrsta za eksponentno funkcijo exp(x). Koeficienti so:

   1/0!, 1/1!, 1/2!, 1/3!, 1/4!, …
 *)
let exp =
  let rec exp k a = Cons (a, fun () -> exp (k+1) (a /. float k)) in
  exp 1 1.0

(* Potenčna vrsta za sinus *)
let sin =
  let rec sin k a = Cons (0.0, fun () -> Cons (a, fun () -> sin (k+2) (-. a /.
float (k * (k+1))))))
  in sin 2 1.0

(* Odvod vrste

      a₀ + a₁ x + a₂ x² + a₃ x³ + a₄ x⁴ + ...

   je

      a₁ + 2 a₂ x¹ + 3 a₃ x² + 4 a₄ x³ + ...

 *)
let diff (Cons (_, s)) =
      let rec diff' k (Cons (a, s)) = Cons (float k *. a, fun () ->
diff' (k+1) (s ()))
      in diff' 1 (s ())

let cos = diff sin
```

6.6.4 Vhod/izhod kot koinduktivni tip

Še en primer koinduktivnega tipa je vhod/izhod (input/output). Tokrat s koinduktivnim tipom izrazimo strukturo programa, ki izvaja operaciji `Read` in `Write`:

```
-- Program, ki simulira operaciji Read in Write s podatkovnim tipom

data InputOutput a =
    Read (String -> InputOutput a) -- program prebere niz in nadaljuje delo
  | Write (String, InputOutput a)  -- program izpiše niz in nadaljuje delo
  | Return a                       -- program konča z danim rezultatom

-- Primer: program, ki izpiše "Hello, world!" in konča z rezultatom ()
hello_world :: InputOutput ()
hello_world = Write ("Hello, world!", Return ())

-- Primer: greeter n uporabnika n-krat vpraša po imenu in ga pozdravi, nato
--         vrne 42
greeter :: Integer -> InputOutput Integer
greeter 0 = Return 42
greeter n = Write ("Your name?", Read (\ str -> Write ("Hello, " ++ str,
greeter (n-1))))

-- Lahko bi naredili tudi potencialno neskončni program?
annoyance :: InputOutput ()
annoyance = Write ("Should I stop? (Y/N)",
                   Read (\answer -> case answer of { "Y" -> Write ("Bye",
Return ()) ; _   -> annoyance }))

-- Ker simuliramo I/O, moramo simulirati tudi operacijski sistem.
-- To naredimo s funkcijo, ki sprejme seznam vhodnih nizov,
-- niz, v katerem hranimo do sedaj izpisane podatke, in program.
-- Funkcija vrne rezultat programa in skupni niz, ki ga je program izpisal.
os :: [String] -> String -> InputOutput a -> (a, String)
```

```
os _            output (Return v)       = (v, output)
os (str : input) output (Read p)        = os input output (p str)
os []           output (Read _)         = error "kernel panic: input not
available"
os input        output (Write (str, p)) = os input (output ++ str) p

primer1 = os [] "" hello_world
-- Dobimo: ((),"Hello, world!")

primer2 = os ["Janezek", "Micka", "Mojca", "Darko"] "" (greeter 3)
-- Dobimo: (42,"Your name?Hello, JanezekYour name?Hello, MickaYour name?Hello,
Mojca")

primer3 = os ["Janezek", "Micka", "Mojca", "Darko"] "" (greeter 5)
-- Dobimo izjemo: *** Exception: kernel panic: input not available

-- Neskončen seznam "N"
no = "N" : no
primer4 = os no "" annoyance
-- Dobimo: se zacikla
```

7. Izpeljava tipov

7.1 Kako programski jeziki uporabljajo tipe

Skoraj vsi programski jeziki imajo tipe, razlikujejo pa se po tem, kako se le-ti uporabljajo.

7.1.1 Kako striktni so tipi

Tipi so lahko bolj ali manj **striktni**. Če so popolnoma striktni, ima vsak izraz v veljavnem programu tip (OCaml, Standard ML, Haskell, Java, C++). Lahko se zgodi, da veljavni program nima tipa, ali vsaj ne takega, ki bi dobro opisal njegovo delovanje (Javascript, Python).

Primer: nabori v Pythonu imajo zelo ohlapen tip `tuple`, ki ne pove nič več kot to, da gre za urejeno večterico:

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

V OCamlu so tipi striktni. Tip urejene trojice je bolj informativen:

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

7.1.2 Dinamični in statični tipi

Poznamo delitev glede na *fazo*, v kateri se uporabijo tipi:

- Programski jezik ima **statične tipe**, če preveri ali izpelje tipe v *statični fazi*, se pravi ob prevajanju ali nalaganju kode, preden se koda požene. Primeri: C, C++, Java, C#, Standard ML, OCaml, Haskell, Swift, Scala.
- Programski jezik ima **dinamične tipe**, če preverja tipe v *dinamični fazi*, se pravi, ko se koda izvaja. Primeri: Scheme, Racket, Javascript, Python.

7.1.3 Preverjanje in izpeljevanje tipov

Programski jezik lahko tipe **preverja** ali **izpeljuje**:

- **preverja** jih, če programer v večji meri zapiše tipe spremenljivk, funkcij in atributov, programski jezik pa preveri, da so pravilno uporabljeni. Primeri: C, C++, Java, C#.
- **izpeljuje** jih, če programerju ni treba podajati tipov spremenljivk, funkcij in atributov (lahko pa jih, če to želi), programski jezik pa sam ugotovi, kakšnega tipa so. Primeri: OCaml, Standard ML, Haskell.

Večina jezikov dopušča kombinacijo obeh tehnik. V OCamlu in Haskellu lahko predpišemo tip, čeprav bi ga programski jezik lahko izpeljal tudi sam. C++, Java in C# omogočajo izpeljavo tipov v omejenem obsegu, na primer pri tipih lokalnih spremenljivk.

7.2 Monomorfni in polimorfni tipi

Tipi so lahko:

- **monomorfni**, če ima vsak izraz največ en tip
- **polimorfni**, če ima lahko izraz hkrati več različnih tipov

Poznamo več vrst polimorfizma, danes bomo obravnavali **parametrični polimorfizem**.

7.3 Izpeljava tipov

Programski jeziki kot so OCaml, Standard ML in Haskell imajo polimorfne tipe, ki jih izpeljejo z algoritmom, ki sta ga razvila Hindley in Milner.

Kakšen tip ima funkcija `λ x . x`, oziroma v OCamlu `fun x -> x`? Možnih je veliko odgovorov:

- `int → int`
- `bool → bool`

- `int * int → int * int`
- `α list → α list` za poljuben `α`
- `β → β` za poljuben `β`.

Od vseh je zadnji najbolj splošen, ker lahko vse ostale dobimo tako, da **parameter** `β` zamenjamo s kakim drugim tipom. Pravimo, da je `β → β` *glavni* tip funkcije `fun x -> x`.

Definicija: Tip izraza je **glavni**, če lahko vse njegove tipe dobimo tako, da v glavnem tipu parametre zamenjamo s tipi (ki lahko vsebujejo nadaljnje parametere).

OCaml je načrtovan tako, da ima vsak veljaven izraz glavni tip, ki ga OCaml izpelje sam. (Izjema so rekurzivne polimorfne funkcije, kjer mora programer sam opredeliti tip, saj algoritem za izračun glavnega tipa rekurzivne funkcije ne obstaja.)

7.3.1 Postopek izpeljave glavnega tipa

Glavni tip izraza `e` izpeljemo v dveh fazah:

1. Izračunamo kandidata za tip `e`, ki vsebuje neznanke, in enačbe, ki jim morajo neznanke zadoščati.
2. Rešimo enačbe s postopkom *združevanja*.

Druga faza se lahko zalomi, če se izkaže, da enačbe nimajo rešitve.

7.3.1.1 Prva faza

V prvi fazi izračunamo kandidata za tip in nabiramo enačbe, ki morajo veljati:

- `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, $\tau_1 = \tau_2 \to \alpha$

- prazen seznam `[]`: uvedemo nov parameter α, tip je α `list`
- sestavljen seznam 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 τ_1 `list`, z enačbami E_1, E_2 in $\tau_2 = \tau_1$ `list`

- 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.

7.3.1.2 Druga faza: združevanje

Imamo množico enačb E

$$
\begin{aligned}
l_1 &= d_1 \\
l_2 &= d_2 \\
l_3 &= d_3 \\
&\dots \\
l_i &= d_i
\end{aligned}
$$

v neznankah α, β, γ, δ, ... Rešujemo z naslednjim postopkom:

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 obravnavamo:
   - 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 α ↦ d. Povsod v r in E zamenjamo α z d in v r dodamo rešitev α ↦ 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*.

Kako to deluje, si poglejmo na primerih.

Primer

Izpelji glavni tip funkcije

```
fun x -> x + 3
```

Odgovor: `int -> int`

Primer

Izpelji glavni tip izraza

```
if 3 < 5 then (fun x -> x) else (fun y -> y + 3)
```

Odgovor: `int -> int`

Naloga

Kakšen je tip Churchevega numerala 3?

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

To naj izračuna OCaml:

```
let zero  = (fun f x -> x) ;;
let one   = (fun f x -> f x) ;;
let two   = (fun f x -> f (f x)) ;;
let three = (fun f x -> f (f (f x))) ;;
```

Important

Naloga*

Kakšen je tip Church-Scottovega numerala 3?

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

To naj izračuna OCaml:

```
 let zero'  = (fun f x -> x) ;;
 let one'   = (fun f x -> f zero' x) ;;
 let two'   = (fun f x -> f one' (f zero' x)) ;;
 let three' = (fun f x -> f two' (f one' (f zero' x))) ;;
 let four'  = (fun f x -> f three' (f two' (f one' (f zero' x)))) ;;
 let five'  = (fun f x -> f four' (f three' (f two' (f one' (f zero'
x))))) ;;
```