Principi programskih jezikov

Andrej Bauer

25 maj, 2026

Kazalo

To so zapiski predavanj pri predmetu Principi programskih jezikov za 2. letnik Interdisciplinarnega študija računalništva in matematike na 1. stopnji. Predmet lahko izberejo tudi študenti Računalništva in informatike.

- **Predznanje:** Osnovno znanje programiranja, na stopnji, ki so ga pridobili po enem letu študija.
- **Opravljanje predmeta:** Predmet opravite s pisnim izpitom.
- **Urnik:**
  - **predavanja:** ponedeljek od 8h do 11h, učilnica P01.
  - **vaje:** po skupinah, glej urnik predavanj in vaj

Gradivo

Osnovno gradivo:

- Vpišite se v predmet Principi programskih jezikov na spletni učilnici, da vas bomo lahko obveščali.
- Video ponetki predavanj
- Zapiski s predavanj
- **Discord server**: na spletni učilnici najdete povezavo.

> **Namig**
>
> Ti zapiski so na voljo na GitHub repozitoriju `ppj-skripa`. Popravki so dobrodošli, prosim naredite pull request.

Dodatna gradivo:

- Benjamin C. Pierce: Types and Programming Languages
- Robert: Practical Foundations for Programming Languages, Cambridge University Press, 2016
- Ivan Bratko: *Prolog Programming for Artificial Intelligence* (4th edition), Addison-Wesley, 2012.
- Markus Triska: The Power of Prolog.
- Martin Lipovača: Learn You a Haskell for Great Good!.

POGLAVJE 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.

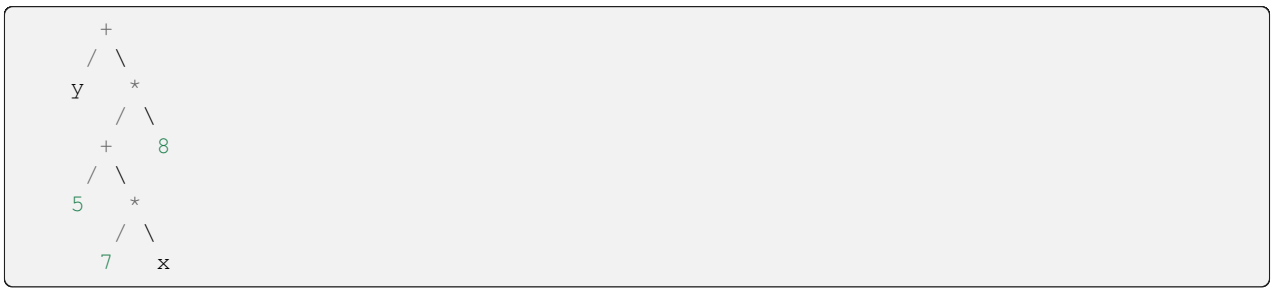
Sintaksa pove, kakšne izraze in programe lahko pišemo v programskem jeziku.

1.3.1 Konkretna sintaksa aritmetičnih izrazov

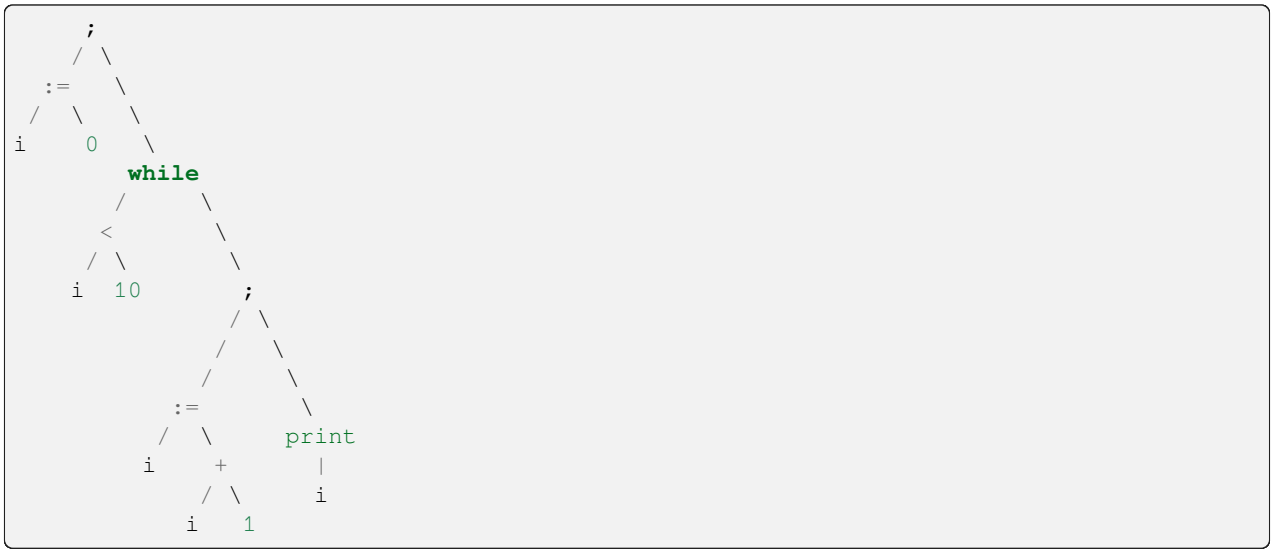
Programer zapiše program kot niz znakov, na primer:

```
"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 ⋯ P□
----------
    S
```

To preberemo: Če smo že dokazali predpostavke `P₁, P₂, ⋯, P□`, 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 $\cdot$, 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
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
      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.

POGLAVJE 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⟩
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
⟨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:

```
─────────────────
 η | true ↪ true


──────────────────
η | false ↪ false
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

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

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

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

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

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

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

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

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

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:

```
       η | e ↬ n
————————————————————————
(η, (x := e))  ↦  (η[x↦n], skip)


       (η, c₁) ↦ (η', c₁')
————————————————————————————
(η, (c₁ ; c₂))  ↦  (η', (c₁' ; c₂))


————————————————————————————
   (η, (skip ; c₂))  ↦  (η, c₂)


             η | b ↬ true
————————————————————————————————————
(η, (if b then c₁ else c₂ end)) ↦ (η, c₁)


             η | b ↬ false
————————————————————————————————————
(η, (if b then c₁ else c₂ end)) ↦ (η, c₂)


      η | b ↬ false
————————————————————————
(η, (while b do c done)) ↦ (η, skip)


                     η | b ↬ true
————————————————————————————————————————————————
 (η, (while b do c done))  ↦ (η, (c ; while b do c done))
```

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

```
(η₁, c₁) ↦ (η₂, c₂) ↦ (η₃, c₃) ↦ …
```

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` c_1 `else` c_2 `done`
- zanka `while b do c done`
- ukaz `skip`
- sestavljeni ukaz c_1 `;` c_2
- 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 $x \circ y$ 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` $\neq$ `0` in `y` $\neq$ `0`), sicer `0`.
- `OR` — logična disjunkcija: `1` iff (`x` $\neq$ `0` ali `y` $\neq$ `0`), sicer `0`.
- `NOT` — logična negacija vrha sklada: neničelen $\to$ `0`, `0` $\to$ `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
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
    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 ;
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
  }
  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

POGLAVJE 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 { P_1 } c_1 { P_2 } in velja { P_2 } c_2 { P_3 } in velja { P_3 } c_3 { P_4 } 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.

Pravilo za `skip`

Ukaz `skip` nima nikakršnega učinka na veljavnost izjave:

```
_________________
{ P } skip { P }
```

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 }
```

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 }
```

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`.

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

```
FV(P) ∩ FA(c) = ∅    [ R ] c [ Q ]
──────────────────────────────────
      [ R ∧ P ] c [ Q ∧ P ]
```

Pozor

Pravilo

```
FV(P) ∩ FA(c) = ∅
──────────────────
  [ P ] c [ P ]
```

ni veljavno. Če bi bilo, bi lahko dokazali

```
[ x > 0 ] 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 > ...
```

Tudi celoštevilska vrednost se lahko zmanjšuje v nedogled:

```
2 > 1 > 0 > -1 > -3 > -5 > -7 > ....
```

Pravilo za popolno pravilnost `while` se glasi:

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

```
[ P ∧ b ∧ e = z ] c [P ∧ e < z ]        z ∉ FA(c)
──────────────────────────────────────────────────
     [ P ] while b do c done [ ¬ b ∧ P ]
```

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

3.3 Primeri

Naloga

Dokaži pravilnost programa:

```
{ x ≤ 7 }
x := x + 3
{ x ≤ 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) ]
```

POGLAVJE 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 $\cdot$, $+$ 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

$$\int_0^1 (x^2 + b)\, dx$$

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

$$\int_0^1 (x^2 + x + a)\, dx$$

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

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

in šele nato za b vstavimo $x + a$:

$$\int_0^1 (t^2 + x + a)\, dt.$$

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 `λ 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 `λ 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 ↦ 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
```

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)))) ;
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
6  := ^ f x . f (f (f (f (f x)))) ;
7  := ^ f x . f (f (f (f (f (f x))))) ;
8  := ^ f x . f (f (f (f (f (f (f x)))))) ;
9  := ^ f x . f (f (f (f (f (f (f (f x))))))) ;
10 := ^ f x . 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)))
```

POGLAVJE 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:

> **Pomni**
>
> **Programski jezik naj programerju omogoči neposredno izražanje idej.**

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:

$$\pi_1(x, y) = x \qquad \pi_2(x, y) = y$$

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

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

Res: črn čevelj velikosti 42 je element $\iota_1(\mathsf{crna}, 42)$, 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

$$A \to B \to C = A \to (B \to C).$$

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**, e_1 in e_2 pa sta izraza. Vzorec je sestavljen iz konstruktorjev, imen in anonimnega vzorca _. Vrednost izraza e_1 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) ;;
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
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 $1 = \{\star\}$):

$$A \cong 1 \times A$$

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 komplicirane podatke. Na primer, podatke o trenutnem času bi lahko predstavili z naborom

$$(\text{leto}, \text{mesec}, \text{dan}, \text{ura}, \text{minuta}, \text{sekunda}, \text{milisekunda})$$

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 $\{\ell_1 = e_1; \ldots; \ell_n = e_n\}$ kot nabor $(e_1, \ldots, e_n)$, le da smo poimenovali njene komponente $\ell_1, \ldots, \ell_n$.

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
    ⋮
  | p□ -> e□
```

Tu so p_1, ..., p_n **vzorci**. Vrednost izraza `match ...` je prvi e_i, za katerega e zadošča vzorcu p_i. 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 _)
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
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>
```

POGLAVJE 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

$$x_0, h(x_0), h(h(x_0)), h(h(h(x_0))), \ldots$$

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

$$1.0, 0.5, 0.75, 0.6875, 0.714844, 0.703842, \ldots$$

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)
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
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 * n + 0` in s `Shl1 n` število `2 * 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:

```
      5
     / \
    3   8
     \   \
      4  10
        /  \
       9   20
```

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:

```haskell
-- 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
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
      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 ())
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
-- 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
```

POGLAVJE 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.

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`, $\tau = \alpha \times \beta$.

- 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 $\alpha \to \tau$ 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`, $\alpha = \tau$. Opomba: običajno na ta način definiramo rekurzivne funkcije, torej bo `x` v resnici funkcija.

Druga faza: združevanje

Imamo množico enačb `E`

$$
\begin{aligned}
l_1 &= d_1 \\
l_2 &= d_2 \\
l_3 &= d_3 \\
&\dots \\
l_? &= d_?
\end{aligned}
$$

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 $\alpha = d$, kjer je α neznanka:
     - če se α pojavi v d, postopek prekinemo, ker *ni rešitve*
     - sicer smo našli rešitev za α, namreč $\alpha \mapsto d$. Povsod v `r` in `E` zamenjamo α z d in v `r` dodamo rešitev $\alpha \mapsto d$
   - če je enačba oblike $l = \alpha$, 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))) ;;
```

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))))) ;;
```

POGLAVJE 8

Logično programiranje

Do sedaj smo spoznali *ukazno* in *deklarativno* programiranje:

- Pri ukaznem programiranju na izvajanje programa gledamo kot na zaporedje akcij, ki spreminjajo stanje sistema (vrednosti spremenljivk).
- Pri deklarativnem programiranju je program izraz, ki se preračuna v končno *vrednost*.

Logično programiranje izhaja iz ideje, da je izvajanje programa **iskanje dokaza**. Da bomo to razumeli, najprej ponovimo nekaj osnov logike.

8.1 Hornove formule

V logiki prvega reda lahko zapišemo formule sestavljene iz konstant $\bot$, $\top$, veznikov $\wedge$, $\vee$, $\Rightarrow$, $\neg$ in kvantifikatorjev $\forall$ (za vsak), $\exists$ (obstaja). Take formule so lahko precej zapletene in niso primerne za logično programiranje, zato se omejimo na tako imenovane **Hornove formule**, ki so oblike

$$\forall x_1, \ldots, x_m.(\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n \Rightarrow \psi).$$

Tu so $\phi_1, \ldots, \phi_n$ in ψ **osnovne formule**, se pravi vsaka od njih je oblike

$$p(t_1, \ldots, t_m)$$

kjer je p **relacijski simbol** in $t_1, \ldots, t_m$ **termi**. Nadalje je term izraz, ki ga lahko sestavimo iz konstant, funkcijskih simbolov in spremenljivk.

Primer

Denimo, da imamo relacijski simbol `less`, funkcijske simbole `plus`, `times`, `succ`, kostanto `zero` in spremenljivki `X` in `Y`. Tedaj sta `plus(times(X,X), times(Y, Y))` in `times(succ(succ(zero)), times(X, Y))` terma in

```
less(plus(times(X,X), times(Y, Y)), times(succ(succ(zero)), times(X, Y)))
```

osnovna formula. Običajno to formulo zapišemo $X \cdot X + Y \cdot Y < 2 \cdot X \cdot Y$.

Poseben primer Hornove formule je **dejstvo**, ki ga dobimo pri $n = 0$:

$$\forall x_1, \ldots, x_m.\psi.$$

Drugi primer je formula brez kvantifikatorjev (v kateri ni spremenljivk, samo konstante), ki ga dobimo pri $m = 0$:

$$\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n \Rightarrow \psi.$$

Primer

Hornova formula

$$\forall a.(\mathsf{pes}(a) \Rightarrow \mathsf{zival}(a))$$

pove, da so psi živali: "za vsak a, če je a pes, potem je a žival".

Primer

Hornova formula

$$\forall x\, y\, z.(\mathsf{otrok}(x, y) \wedge \mathsf{otrok}(y, z) \wedge \mathsf{zenska}(z) \Rightarrow \mathsf{babica}(x, z))$$

pravi: "za vse (osebe) x, y, z, če je x otrok od y in y otrok od z in je z ženska, potem je z babica od x".

Primer

Formula

$$\forall x\, y\, z.\mathsf{otrok}(x, z) \wedge \mathsf{otrok}(y, z) \wedge \mathsf{zenska}(x) \wedge \mathsf{zenska}(y) \Rightarrow \mathsf{sestra}(x, y).$$

ne pomeni "x in y sta sestri" ampak "x in y sta sestri ali polsestri, ali pa sta enaka".

8.2 Predstavitev funkcije z relacijo

S Hornovimi formulami lahko izrazimo tudi matematična dejstva. Peanova aksioma za seštevanje se glasita

$$\forall n.n + 0 = n$$
$$\forall k\, m.k + \mathsf{succ}(m) = \mathsf{succ}(k + m)$$

Pravzaprav v Prologu ni funkcij! Definirati moramo *relacijo*, ki predstavlja funkcijo:

Definicija

Pravimo, da relacija R **predstavlja** funkcijo f, če je $f(x) = y \Leftrightarrow R(x, y)$.

Za funkcijo seštevanja: namesto operacije $+$ definiramo relacijo vsota, da velja:

$$x + y = z \Leftrightarrow \mathsf{vsota}(x, y, z).$$

S Hornovima formulama ju zapišemo takole, pri čemer $\mathsf{vsota}(x, y, z)$ beremo "vsota x in y je z":

$$\forall n.\mathsf{vsota}(n, 0, n)$$
$$\forall k\, m\, n.\mathsf{vsota}(k, m, n) \Rightarrow \mathsf{vsota}(k, \mathsf{succ}(m), \mathsf{succ}(n))$$

Prva formula očitno ustreza prvemu aksiomu, druga pa je ekvivalentna

$$\forall k\, m\, n.k + m = n \Rightarrow k + \mathsf{succ}(m) = \mathsf{succ}(n),$$

kar je ekvivalentno drugemu aksiomu (premisli!).

Naloga

S Hornovimi formulami zapišite Peanova aksioma za množenje:

$$\forall n.n \cdot 0 = 0$$
$$\forall k\, m.k \cdot \mathsf{succ}(m) = k + k \cdot m$$

Uporabi relacijo vsota iz prejšnjega primera, ter relacijo $\mathsf{zmnozek}(x, y, z)$, ki ga beremo "zmnožek x in y je z".

Primer

Nekaterih dejstev s Hornovimi formulami ne moremo izraziti, na primer negacije $\neg\phi$ in eksistenčnih formul $\exists x.\phi$.

8.3 Sistematično iskanje dokaza

Denimo, da imamo Hornove formule in želimo vedeti, ali iz njih sledi dana izjava. Kako bi *sistematično* poiskali dokaz?

Primer

Najprej poglejmo primer brez kvantifikatorjev. Ali iz Hornovih formul

1. $X \wedge Y \Rightarrow C$
2. $A \wedge B \Rightarrow C$
3. $X \Rightarrow B$
4. $A \Rightarrow B$
5. A

sledi C?

Iskanja dokaza se lotimo sistematično. Katera od formul bi lahko pripeljala do dokaza izjave C? Prva ali druga. Poskusimo obe:

- če uporabimo $X \wedge Y \Rightarrow C$, C prevedemo na *podnalogi* X in Y. A tu se zatakne, ker o X in Y ne vemo nič pametnega.

- če uporabimo $A \wedge B \Rightarrow C$, C prevedemo na *podnalogi* A in B:
  - dokažimo A: to velja zaradi 5. formule
  - dokažimo B: uporabimo lahko 3. ali 4. formulo. Tretja ne deluje, četrta pa dokazovanje prevede na podnalogo A, ki velja.

Primer

Ali iz

1. $\forall x\, y.\mathsf{otrok}(x, y) \Rightarrow \mathsf{mlajsi}(x, y)$
2. $\mathsf{otrok}(\mathsf{miha}, \mathsf{mojca})$

sledi $\mathsf{mlajsi}(\mathsf{miha}, \mathsf{mojca})$? Če v prvi formuli vzamemo $x = \mathsf{miha}$ in $y = \mathsf{mojca}$, lahko nalogo prevedemo na $\mathsf{otrok}(\mathsf{miha}, \mathsf{mojca})$. To pa velja zaradi druge formule.

Primer

Ali iz

1. $\forall x.\mathsf{sodo}(x) \Rightarrow \mathsf{liho}(\mathsf{succ}(x))$
2. $\forall y.\mathsf{liho}(y) \Rightarrow \mathsf{sodo}(\mathsf{succ}(y))$
3. $\mathsf{sodo}(\mathsf{zero})$

sledi $\mathsf{sodo}(\mathsf{succ}(\mathsf{succ}(\mathsf{zero})))$? Tokrat zapišimo bolj sistematično postopek iskanja:

- dokaži $\mathsf{sodo}(\mathsf{succ}(\mathsf{succ}(\mathsf{zero})))$
- uporabimo drugo formulo, za y vstavimo $\mathsf{succ}(\mathsf{zero})$ in dobimo nalogo
- dokaži $\mathsf{liho}(\mathsf{succ}(\mathsf{zero}))$
- uporabimo prvo formulo, za x vstavimo zero in dobimo nalogo
- dokaži $\mathsf{sodo}(\mathsf{zero})$
- to velja zaradi tretje formule.

V splošnem bomo morali rešiti nalogo **združevanja**: poišči take vrednosti spremenljivk, da sta dani formuli enaki*. Podobno nalogo smo že reševali, ko smo obravnavali parametrični polimorfizem, kjer smo izenačevali tipe.

8.4 Logično programiranje

V logičnem programiranju je program podan s

- seznamom **pravil** $H_1, \ldots, H_k$, ki so Hornove formule
- **poizvedbo** G, ki je formula oblike $\exists y_1, \ldots, y_n.p(t_1, \ldots, t_m)$.

Zanima nas, ali poizvedba sledi iz pravil. Poizvedbo G predelamo na poizvedbe in **iščemo v globino**, takole:

V seznamu $H_1, \ldots, H_m$ poišči prvo formulo, ki je oblike

$$\forall x_1, \ldots, x_n.\phi_1 \wedge \cdots \wedge \phi_m \Rightarrow \psi,$$

katere sklep ψ je **združljiv** s $p(t1, \dots, t_m)$. To pomeni da lahko za $y_1, \dots, y_n$ vstavimo take vrednosti $u_1, \dots, u_n$ in za $x_1, \dots, x_n$ take vrednosti $v_1, \dots, v_m$, da sta formuli

$$p(u_1, \dots, u_n)$$

in

$$\psi(v_1, \dots, v_m)$$

enaki. Možno je, da izbira vrednosti $u_1, \dots, u_n$ in $v_1, \dots, v_m$ ni enolična. V tem primeru izberemo *najbolj splošne vrednosti*, ki jih najdemo s postopkom združevanja, ki smo ga spoznali v poglavju o izpeljavi tipov.

S tem smo poizvedbo predelali na poizvedbe $\phi_1(v_1, \dots, v_m), \dots, \phi_n(v_1, \dots, v_m)$, ki jih rešujemo po vrsti rekurzivno. (Če se v teh poizvedbah pojavljajo spremenljivke, jih obravnavamo, kot da smo jih kvantificirali z $\exists$.)

Primer

Poglejmo si še enkrat primer, ko imamo Hornove formule

1. $\mathsf{sodo}(\mathsf{zero})$
2. $\forall x.\mathsf{sodo}(x) \Rightarrow \mathsf{liho}(\mathsf{succ}(x))$
3. $\forall y.\mathsf{liho}(y) \Rightarrow \mathsf{sodo}(\mathsf{succ}(y))$

in poizvedbo $\exists z.\mathsf{liho}(z)$.

Ali lahko združimo $\mathsf{sodo}(\mathsf{zero})$ in $\mathsf{liho}(z)$? Ne.

Ali lahko združimo $\mathsf{liho}(\mathsf{succ}(x))$ in $\mathsf{liho}(z)$? Poskusimo s postopkom združevanja: Enačbo

$$\mathsf{liho}(\mathsf{succ}(x)) = \mathsf{liho}(z)$$

prevedemo na enačbo

$$\mathsf{succ}(x) = z$$

Dobili smo rešitev za z in ni več enačb, torej je x poljuben. Torej uporabimo drugo pravilo, ki prevede nalogo na

$$\exists x.\mathsf{sodo}(x)$$

Ali lahko to združimo s prvo formulo? Poskusimo rešiti

$$\mathsf{sodo}(\mathsf{zero}) = \mathsf{sodo}(x)$$

Rešitev je $x = \mathsf{zero}$. Ker je prva formula dejstvo, ni nove podnaloge.

Rešitev se glasi: $x = \mathsf{zero}$, $z = \mathsf{succ}(x)$. Končna rešitev je torej

$$z = \mathsf{succ}(\mathsf{zero})$$

Dokazali smo, da res obstaja liho število, namreč $\mathsf{succ}(\mathsf{zero})$.

Primer

Če v prejšnjem primeru zamenjamo vrstni red pravil,

1. $\forall x.\mathsf{sodo}(x) \Rightarrow \mathsf{liho}(\mathsf{succ}(x))$
2. $\forall y.\mathsf{liho}(y) \Rightarrow \mathsf{sodo}(\mathsf{succ}(y))$
3. $\mathsf{sodo}(\mathsf{zero})$

potem poizvedba $\exists z.\mathsf{liho}(z)$ privede do neskončne zanke (ker iščemo v globino in vedno uporabimo prvo pravilo, ki deluje). Namreč, z uporabo prvega pravila dobimo poizvedbo

$$\exists x.\mathsf{sodo}(x)$$

nato z uporabo drugega pravila ($x = \mathsf{succ}(y)$)

$$\exists y.\mathsf{liho}(y)$$

nato z uporabo prvega pravila

$$\exists u.\mathsf{sodo}(u)$$

in tako naprej. Tretje pravilo nikoli ne pride na vrsto!

8.5 Prolog

Prolog je programski jezik, v katerem logično programiramo. Ima nekoliko nenavadno sintakso:

- namesto $A \wedge B$ pišemo `A, B`
- namesto $A \vee B$ pišemo `A ; B`
- namesto $A \Rightarrow B$ pišemo `B :- A` (pozor, zamenjal se je vrstni red, $B \Leftarrow A$!)
- kvantifikatorjev $\forall$ in $\exists$ ne pišemo, ampak **kvantificirane spremenljivke pišemo z velikimi črkami**
- **konstante, predikate in funkcije pišemo z malimi črkami**.

Na koncu vsake formule zapišemo piko.

Predelajmo primer iz prejšnjega razdelka v Prolog. Najprej v datoteko `even_odd.pl` spravimo pravila (pri čemer pravila za `sodo` zložimo skupaj, da se ne pritožuje):

```
liho(succ(X)) :- sodo(X).
sodo(succ(Y)) :- liho(Y).
sodo(zero).
```

Datoteko naložimo v interaktivno zanko. Ta nam omogoča, da vpišemo poizvedbo in dobimo odgovor:

```
?- liho(Z).
Z = succ(zero) ;
Z = succ(succ(succ(zero))) ;
Z = succ(succ(succ(succ(succ(zero))))) .
```

Ko nam prolog poda odgovor, lahko z znakom `;` zahtevamo, da išče še naprej. Z znakom `.` zaključimo iskanje.

Naloga

Ali se prolog res spusti v neskončno zanko, če zamenjamo vrsti red pravil za `sodo`?

Naloga

Na svoj računalnik si namesti SWI Prolog in poženi zgornji program.

8.5.1 Seznami

Kako bi v Prologu naredili sezname? V Ocamlu smo jih definirali kot induktivni tip:

```
type 'a list = Nil | Cons of 'a * 'a list
```

Na primer, `Cons(a, Cons(b, Cons(c, Nil)))` je seznam z elementi `a`, `b` in `c`.

V Prologu ni tipov, lahko pa uporabljamo poljubne konstante in konstruktorje, le z malimi črkami jih je treba pisati. Torej lahko sezname še vedno predstavljamo z `nil` in `cons`.

Seznamov ni treba v naprej definirati, se pravi, ni treba razlagati, kaj sta `nil` in `cons`. Prolog ju obravnava kot simbola, s katerimi lahko tvorimo izraze. Seznam `[a; b; c]` zapišemo `cons (a, cons (b, cons (c, nil)))`.

Opomba: Prolog ima tudi vgrajene sezname, glej spodaj.

8.5.2 Relacija `elem`

Da bomo dobili občutek za moč logičnega programiranja, definirajmo nekaj funkcij za delo s seznami.

Naš prvi program je relacija, ki ugotovi, ali je dani `X` pripada danemu seznamu `L`:

```
elem(X, cons(X, _)).
elem(X, cons(_, L)) :- elem(X, L).
```

V datoteko `list.pl` zapišemo:

```
elem(X, cons(X, _)).
elem(X, cons(_, L)) :- elem(X, L).

join(nil, Y, Y).
join(cons(A, X), Y, cons(A, Z)) :- join(X, Y, Z).

:- use_module(library(lists)).
```

Poskusimo:

```
?- elem(a, cons(b, cons(a, cons(c, cons(d, cons(a, nil)))))).
true ;
true ;
false.
```

Zakaj smo dvakrat dobili `true` in nato `false`?

Vprašamo lahko tudi, kateri so elementi danega seznama:

```
?- elem(X, cons(a, cons(b, cons(a, cons(c, nil))))).
X = a ;
X = b ;
X = a ;
X = c ;
false.
```

In celo, kateri seznami vsebujejo dani element!

```
?- elem(a, L).
L = cons(a, _3234) ;
L = cons(_3232, cons(a, _3240)) ;
L = cons(_3232, cons(_3238, cons(a, _3246))) ;
L = cons(_3232, cons(_3238, cons(_3244, cons(a, _3252)))) ;
L = cons(_3232, cons(_3238, cons(_3244, cons(_3250, cons(a, _3258))))) ;
L = cons(_3232, cons(_3238, cons(_3244, cons(_3250, cons(_3256, cons(a, _3264)))))) .
```

Prolog je ustvaril pomožne spremenljivke `_XYZW`, s katerimi označi poljubne terme.

8.5.3 Relacija `join`

Funkcijo, ki stakne seznama predstavimo s trimestno relacijo `join`:

`join(X, Y, Z)` pomeni, da je `Z` enak stiku seznamov `X` in `Y`.

Zapišimo pravila zanjo:

```
join(nil, Y, Y).
join(cons(A, X), Y, cons(A, Z)) :- join(X, Y, Z).
```

To je podobno funkciji, ki bi jo definirali v OCamlu:

```
let rec join x y=
  match (x, y) with
  | (Nil, y) -> y
  | (Cons (a, x), y) ->
      let z = join x y in
      Cons (a, z)
```

Takole izračunamo stik seznamov `cons(a, cons(b, nil))` in `cons(x, cons(y, cons(z, nil)))`:

```
?- join(cons(a, cons(b, nil)), cons(x, cons(y, cons(z, nil))), Z).
Z = cons(a, cons(b, cons(x, cons(y, cons(z, nil))))).
```

Vgrajeni seznami

Prolog že ima vgrajene sezname:

- `[e_1, e_2, ..., e_m]` je seznam elementov `e_1`, `e_2`, ..., `e_m`.
- `[e | ℓ]` je seznam z glavo `e` in repom ℓ
- `[e_1, e_2, ..., e_m | ℓ]` je seznam, ki se začne z elementi `e_1`, `e_2`, ..., `e_m` in ima rep ℓ.

Za delo s seznami je na voljo knjižnica `lists`, ki jo naložimo z ukazom

```
:- use_module(library(lists)).
```

Ta že vsebuje relaciji `member` (ki smo jo zgoraj imenovali `elem`) in `append` (ki smo jo zgoraj imenovali `join`). Preizkusimo:

```
?- append([a,b,c], [d,e,f], Z).
Z = [a, b, c, d, e, f].
```

Lahko pa tudi vprašamo, kako razbiti seznam `[a,b,c,d,e,f]` na dva podseznama:

```
?- append(X, Y, [a,b,c,d,e,f]).
X = [],
Y = [a, b, c, d, e, f] ;
X = [a],
Y = [b, c, d, e, f] ;
X = [a, b],
Y = [c, d, e, f] ;
X = [a, b, c],
Y = [d, e, f] ;
X = [a, b, c, d],
Y = [e, f] ;
X = [a, b, c, d, e],
Y = [f] ;
X = [a, b, c, d, e, f],
Y = [] ;
false.
```

8.5.4 Enakost in neenakost

Včasih v Prologu potrebujemo enakost in neenakost. Enakost pišemo `s = t` in neenakost `s \= t`.

8.6 Primer: ukazni programski jezik

Če bo čas, si bomo ogledali, kako v Prologu implementiramo tolmač za preprost ukazni programski jezik:

```
/* Tolmač za preprost ukazni jezik v 100 vrsticah (s komentarji). */

/* Sintaksa jezika:

   Celoštevilski izrazi:
   - celo število k pišemo "int(k)"
   - spremenljivko x pišemo "var(x)"
   - "e₁ + e₂" pišemo "plus(e₁, e₂)"
   - "e₁ * e₂" pišemo "times(e₁, e₂)"

   Boolovi izrazi:
   - "e₁ < e₂" pišemo "less(e1, e2)"
   - "b₁ ∨ b₂" pišemo "or(b1, b2)"
   - "b₁ ∧ b₂" pišemo "and(b1, b2)"
   - "¬ b" pišemo "not(b)"

   Ukazi:
   - "skip" pišemo "skip"
   - "x := e" pišemo "let(x, e)"
   - "c₁ ; c₂" pišemo "seq(c1, c2)"
   - "if b then c₁ else c₂ end" pišemo "if(b, c₁, c₂)"
   - "while b do c done" pišemo "while(b, c)"
*/

/* Okolje predstavimo s seznamom seznamov [[x₁,v₁], …, [x₂,v₂]] */

/* get(X, Env, V) velja, če ima spremenljivka X v okolju Env vrednost V. */
get(X, [[X,V] | _], V).
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
get(X, [_ | Env], V) :- get(X, Env, V).

/* put(X, V, Env1, Env2) velja, če dobimo Env2 iz Env1, ko X nastavimo na V */
put(X, V, [[X,_] | L], [[X,V] | L]).
put(X, V, [[Y,W] | L], [[Y,W] | M]) :- put(X, V, L, M).

/* eval(Env, E, V) velja, če v okolju Env izraz E evaluira v vrednost V */

/* Aritmetični izrazi */
eval(_, int(V), V).

eval(Env, var(X), V) :- get(X, Env, V).

eval(Env, plus(E1, E2), V) :-
    eval(Env, E1, V1),
    eval(Env, E2, V2),
    V is V1 + V2.

eval(Env, times(E1, E2), V) :-
    eval(Env, E1, V1),
    eval(Env, E2, V2),
    V is V1 * V2.

/* Boolovi izrazi */
eval(_, false, false).

eval(_, true, true).

eval(Env, less(E1, E2), true) :-
    eval(Env, E1, V1),
    eval(Env, E2, V2),
    V1 < V2.

eval(Env, less(E1, E2), false) :-
    eval(Env, E1, V1),
    eval(Env, E2, V2),
    V1 >= V2.

eval(Env, or(B1, _), true) :- eval(Env, B1, true).
eval(Env, or(B1, B2), V) :- eval(Env, B1, false), eval(Env, B2, V).

eval(Env, and(B1, _), false) :- eval(Env, B1, false).
eval(Env, and(B1, B2), V) :- eval(Env, B1, true), eval(Env, B2, V).

eval(Env, not(B), false) :- eval(Env, B, true).
eval(Env, not(B), true) :- eval(Env, B, false).

/* finish(C, Env1, Env2) velja, če ukaz C v okolju Env1 v enem koraku konča v
   okolju Env2 */
finish(skip, Env, Env).

finish(let(X, E), Env1, Env2) :-
    eval(Env1, E, V),
    put(X, V, Env1, Env2).

/* step(C1, Env1, C2, Env2) velja, če ukaz C1 v okolju Env1 v enem koraku
   spremeni okolje v Env2 in program se nadaljuje z ukazom C2 */
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
step(if(B, C1, _), Env, C1, Env) :- eval(Env, B, true).
step(if(B, _, C2), Env, C2, Env) :- eval(Env, B, false).

step(seq(C1, C2), Env1, C2, Env2) :- finish(C1, Env1, Env2).
step(seq(C1, C3), Env1, seq(C2, C3), Env2) :- step(C1, Env1, C2, Env2).

/* Upoštevamo enačbo:

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

*/
step(while(B, C), Env, if(B, seq(C, while(B, C)), skip), Env).


/* run(C, Env1, Env2) velja, če ukaz C v okolju Env1 po končno mnogo korakih
   konča v okolju Env2 */
run(C, Env1, Env2) :- finish(C, Env1, Env2).
run(C, Env1, Env3) :- step(C, Env1, C2, Env2), run(C2, Env2, Env3).

/* Primeri */

/* Minimum:

   if x < y then z := x else z := y end
*/
minimum(X, Y, Z) :-
  run(
    if(less(var(x), var(y)), let(z, var(x)), let(z, var(y))),
    [[x,X], [y,Y], [z,42]],
    Env),
  get(z, Env, Z).

/* S števcem i štej od 0 do N:

   i = 0 ;
   while (i < N) do
     i := i + 1
   done

   Opomba: spremenljivke j ne potrebujemo, v okolje jo postavimo za hec.
*/
count(N, Env) :-
  run(
    seq(let(i,int(0)),
        while(less(var(i), int(N)),
              let(i, plus(var(i), int(1)))
    )),
    [[i,42], [j,100]],
    Env).

/* Faktoriela N!:

   i := 1 ;
   p := 1 ;
   while (i < N + 1) do
     p := p * i ;
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
     i := i + 1
   done
 */

faktoriela(N, F) :-
  run(
    seq(let(i, int(1)),
    seq(let(p, int(1)),
        while(less(var(i), plus(int(N), int(1))),
              seq(let(p, times(var(p), var(i))),
                  let(i, plus(var(i), int(1))))
    ))),
    [[i,0],[p,0]],
    Env2
  ),
  get(p, Env2, F).

/* Celoštevilski kvadratni koren N, zaokrožen navzgor:

   i := 0
   while i * i < N do
     i := i + 1
   done
*/
sqrt(N, S) :-
  run(
    seq(let(i, int(0)),
        while(less(times(var(i), var(i)), int(N)),
              let(i, plus(var(i), int(1)))
        )
    ),
   [[i,42]],
   Env),
   get(i, Env, S).

/* Ali lahko računamo nazaj? */
```

POGLAVJE 9

Logično programiranje z omejitvami

V logičnem programiranju je program spisek logičnih izjav, ki opisujejo rešitev (seveda morajo biti izjave izražene s Hornovimi formulami). V istem duhu bi lahko računanje s števili opisali z *enačbami* (in *neenačbami*) namesto z zaporedjem aritmetičnih operacij in primerjav. Če bi prologu dodali algoritme za reševanje sistemov enačb (in neenačb), bi lahko takšne opise uporabili za računanje z aritmetičnimi izrazi.

9.1 Aritmetika v Prologu

V prologu računamo s števili takole:

```
?- X is (10 + 4) * 3.
X = 42.

?- X is 2 + 3, Y is 10 * X.
X = 5,
Y = 50.
```

Operator `is` sprejme spremenljivko `X` in aritmetični izraz `E`, izračuna vrednost izraza `E` in dobljeno vrednost priredi spremenljivki `X`.

Števila lahko tudi primerjamo z operatorji za primerjavo števil:

```
?- 221 * 19 =:= 247 * 17.
true.

?- 23 < 42.
true.
```

Takole bi implementirali funkcijo faktoriela:

```
faktoriela(0, 1).
faktoriela(N, F) :-
    N > 0,
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
    M is N - 1,
    faktoriela(M, G),
    F is N * G.
```

Preizkusimo:

```
?- faktoriela(7, F).
F = 5040.
```

V duhu prologa bi pričakovali, da `faktoriela` deluje v obse smeri:

```
?- faktoriela(N, 6).
ERROR: Arguments are not sufficiently instantiated
ERROR: In:
ERROR:     [9] _13020>0
ERROR:     [8] faktoriela(_13046,6) at /var/folders/tw/2p25pzx951n_ht9607h10kkc0000gn/
 ↪T/prolcomp13972QVt.pl:5
ERROR:     [7] <user>
```

Napako se pojavi, ker `is` in < ne delujeta kot običajna predikata. V izrazu `X is E`, aritmetični izraz `E` ne sme vsebovati spremenljivk z neznano vrednostjo. Na primer, če poskusimo izračunati `Y - Y`

```
?- Z is Y - Y.
ERROR: Arguments are not sufficiently instantiated
ERROR: In:
ERROR:     [8] _3132 is _3138-_3140
ERROR:     [7] <user>
```

dobimo napako, ker izraz na desni nima točno določene vrednosti. Tudi < ne deluje, če mu damo neznane vrednosti:

```
?- X < X + 1.
ERROR: Arguments are not sufficiently instantiated
ERROR: In:
ERROR:     [8] _5844<_5850+1
ERROR:     [7] <user>
```

Naj povemo, da predikat = ni uporaben pri računanju rezultatov, saj samo uporablja postopek združevanja:

```
?- X = 2 + 3.
X = 2+3.

?- 2 + X = 2 + 3.
X = 3.

?- X + 2 = 2 + 3.
false.

?- Z = Y - Y.
Z = Y-Y.
```

Danes bomo spoznali **logično programiranje z omejitvami** (angl. logic *constraint programming*), ki omogoča dosti bolj fleksibilno obravnavo aritmetičnih izrazov.

9.2 Logično programiranje z omejitvami

V SWI prologu programiranje z omejitvami omogoča knjižnica `clpfd` (constraint logic programming on finite domains):

```
?- use_module(library(clpfd)).
true.

?- Z #= Y - Y.
Z = 0,
Y in inf..sup.

?- 3 + X #= 3 + 2.
X = 2.

?- X + 2 #= 3 + 2.
X = 3.

?- 3 #< X, 4 * X #< 25.
X in 4..6.
```

Kot vidimo, je treba namesto operatorjev =, <, > uporabljati aritmetične omejitve `#=`, `#<`, `#>`, … Ali lahko rešujemo tudi kvadratne enačbe?

```
?- X * X #= 1.
X in -1\/1.
```

Prolog je odgovoril, da je vrednost `X` bodisi `-1` bodisi `1`. Poskusimo še eno kvadratno enačbo:

```
?- X * X - 5 * X + 6 #= 0.
X in 2..sup,
5*X#=_30476,
X^2#=_30500,
_30476 in 10..sup,
-6+_30476#=_30500,
_30500 in 4..sup.
```

Tokrat smo dobili čuden odgovor. Poglejmo pobližje, kako deluje programiranje z omejitvami.

9.2.1 Domene

Programiranje z omejitvami vedno deluje na neki **domeni**, se pravi na množici vrednosti z dano strukturo. Tipične domene so:

- cela števila (domena `fd`)
- realna in racionalna števila (domena `qr`)
- Boolova algebra (domena `pb`)

Vsaka od teh domen zahteva specialne algoritme, ki znajo poenostavljati in združevati omejitve, ki so sestavni del programa. Mi bomo spoznali programiranje z omejitvami za cela števila, ki se imenuje tudi **končne domene** (angl. finite domain), ker vedno obravnavamo cela števila iz neke končne množice vrednosti.

9.2.2 Omejitve za domeno `fd`

V logičnem programiranju z omejitvami je program podan s Hornovimi formulami in **omejitvami**, ki predpisujejo dovoljene vrednosti spremenljivk. Prolog omejitve zbira in jih poenostavlja, z nekaj sreče pa jih kar razreši. Za domeno `fd` imamo več vrst omejitev.

Aritmetične omejitve

Aritmetične omejitve zapišemo z osnovnimi aritmetičnimi operacijami

```
+   -   *   ^   min   max   mod   rem   abs   //   div
```

in operatorji za primerjavo

```
#=   #\=   #>=   #=<   #>   #<
```

Intervalske omejitve

Intervalska omejitev

```
X in A..B
```

določi, da mora veljati $A \leq X \leq B$. Če želimo nastaviti samo zgornjo ali spodnjo mejo za `X`, lahko namesto `A` pišemo `inf` (kar pomeni $-\infty$) in za `B` pišemo `sup` (kar pomeni $+\infty$). Na primer,

```
X in inf..5
```

pomeni, da velja $X \leq 5$. Pogosto želimo z intervalom omejiti več spremenljivk hkrati:

```
X in 1..5, Y in 1..5, Z in 1..5.
```

V tem primeru lahko uporabimo `ins`:

```
[X,Y,Z] ins 1..5.
```

V splošnem `L ins A..B` pomeni, da mora veljati $A \leq X \leq B$ za vse elemente $X \in L$ seznama `L`.

Kombinatorne in globalne omejitve

Omejitev `all_distinct([X1, ..., X?])` zagotovi, da imajo spremenljivke X_1, …, `X?` različne vrednosti. Primer uporabe bomo videli kasneje.

Ostale kombinatorne in globalne omejitve so opisane v priročniku za SWI prolog.

9.2.3 Naštevanje

Program z omejitvami napišemo v dveh delih:

1. Podamo omejitve.
2. Podamo zahtevo, da naj prolog našteje vse rešitve, glede na dane omejitve.

Drugi korak izvedemo s predikatom `label` (ter `indomain` in `labeling`, preberite sami). Če podamo samo omejitve, jih prolog izpiše, a ne pokaže konkretnih rešitev. Denimo, da želimo `X, Y ∈ {1, 2, 3, 4}` in `X < Y`:

```
?- [X,Y] ins 1..4, X #< Y.
X in 1..3,
X#=<Y+ -1,
Y in 2..4.
```

Prolog je omejitve poenostavil:

- `X ∈ {1, 2, 3}`
- `X ≤ Y - 1`
- `Y ∈ {2, 3, 4}`

Če dodamo še `label([X,Y])`, našteje konkretne rešitve:

```
?- [X,Y] ins 1..4, X #< Y, label([X,Y]).
X = 1,
Y = 2 ;
X = 1,
Y = 3 ;
X = 1,
Y = 4 ;
X = 2,
Y = 3 ;
X = 2,
Y = 4 ;
X = 3,
Y = 4.
```

Poglejmo si primere, s katerimi bomo najbolje spoznali, kako deluje programiranje z omejitvami.

Primer

Vrnimo se k funkciji faktoriela:

```
faktoriela(0, 1).
faktoriela(N, F) :-
    N > 0,
    M is N - 1,
    faktoriela(M, G),
    F is N * G.
```

Operatorje `is` in `>` zamenjajmo z omejitvami (in funkcijo preimenujmo, da ne bo zmede):

```
:- use_module(library(clpfd)).

fakulteta(0, 1).
fakulteta(N, F) :-
    N #> 0,
    M #= N - 1,
    F #= N * G,
    fakulteta(M, G).
```

Opomba: obrnili smo vrstni red zadnjih dveh pogojev. S tem smo poskrbeli, da se *najprej* zabeležijo vse omejitve, šele nato pa se izvede rekurzivni klic.

Preizkusimo:

```
?- fakulteta(7, F).
F = 5040 .

?- fakulteta(N, 6).
N = 3.

?- fakulteta(N, 1).
N = 0 ;
N = 1 ;
false.
```

Primer

Pitagorejska trojica je trojica celih števil `(A, B, C)`, za katero velja $A^2 + B^2 = C^2$. Poleg tega smemo zaradi simetrije med `A` in `B` predpostaviti $A \leq B$.

```
:- use_module(library(clpfd)).

pitagora(A, B, C) :-
    A #=< B,
    A * A + B * B #= C * C.

pitagora_do([A,B,C], N) :-
    pitagora(A,B,C),
    [A, B, C] ins 1..N,
    label([A,B,C]).
```

Primer

Na vajah boste programirali permutacije seznamov v navadnem prologu. Permutacije števil lahko elegantno zapišemo z omejitvami:

```
:- use_module(library(clpfd)).

permutacija(N, P) :-
    length(P, N),
    P ins 1..N,
    all_distinct(P).
```

V poizvedbi zahtevamo, da se rešitve dejansko naštejejo:

```
?- permutacija(6,P), label(P).
```

Pojasnimo vsako od zahtev:

- `length(P, N)` pove, da je `P` seznam dolžine `N`,
- `P ins 1..N` pove, da so vsi elementi seznama `P` števila med `1` in `N`
- `all_distinct(P)` pove, da so vsi elementi seznama `P` med seboj različni.
- `label(P)` je zahteva, da je treba našteti vse sezname `P`, ki zadoščajo navedenim trditvam.

Poskusimo:

```
?- permutacije(3,P).
P = [1, 2, 3] ;
P = [1, 3, 2] ;
P = [2, 1, 3] ;
P = [2, 3, 1] ;
P = [3, 1, 2] ;
P = [3, 2, 1].
```

Ker smo uporabili programiranje z omejitvami, zlahka računamo tudi „nazaj“,

```
?- permutacije(N,[1,2,3]).
N = 3.
```

in preverimo, ali dani seznam je permutacija:

```
?- permutacije(N,[1,2,3,3]).
false.
```

9.2.4 Primer: Sudoku

Skupaj implementirajmo Sudoku. Zasnovati moramo ustrezno predstavitev igre in zapisati pogoje, ki opisujejo pravila. Ostalo bo prolog naredil sam.

Načrt:

1. Igralno polje 9×9 predstavimo se seznamom vrstic:
   - imamo 9 vrstic,
   - vsaka vrstica ima dolžino 9.
2. Vsi elementi, ki se pojavljajo v vrsticah so števila med 1 in 9
3. Vsaka vrstica je permutacija
4. Vsak stolpec je permutacija.
5. Vsak podkvadrat 3×3 je permutacija.

Vsako od zgornjih omejitev zapišemo v prologu.

`Rows` je seznam dolžine 9

Pogoj lahko izrazimo kot

```
Rows = [X1,X2,X3,X4,X5,X6,X7,X8,X9]
```

ali kot

```
length(Rows, 9)
```

Vsak element `Rows` je dolžine 9

Prvi poskus:

```
Rows = [X1,X2,X3,X4,X5,X6,X7,X8,X9],
length(X1,9),
length(X2,9),
length(X3,9),
length(X4,9),
length(X5,9),
length(X6,9),
length(X7,9),
length(X8,9),
length(X9,9).
```

Drugi poskus: uporabimo `maplist`.

```
length9(L) :- length(L,9).

maplist(length9, Rows).
```

Vsi elementi elementov `Rows` so med 1 in 9

Prvi poskus:

```
Rows = [X1,X2,X3,X4,X5,X6,X7,X8,X9],
X1 ins 1..9,
X2 ins 1..9,
X3 ins 1..9,
X4 ins 1..9,
X5 ins 1..9,
X6 ins 1..9,
X7 ins 1..9,
X8 ins 1..9,
X9 ins 1..9.
```

Z uporabo `maplist`:

```
vsi_med_1_9(L) :- L ins 1..9.

maplist(vsi_med_1_9, Rows).
```

Z uporabo `append`:

```
append(Rows, Elementi), Elementi ins 1..9.
```

Vsaka vrstica je permutacija

```
maplist(all_distinct, Rows).
```

Vsak stolpec je permutacija

Ideja: matriko `Rows` transponiramo in zahtevamo, da so vrstice transponiranke permutacije:

```
transpose(Rows, Columns), map_list(all_distinct, Columns).
```

Vsak podkvadrat 3 × 3 je permutacija

Ideja: če imamo vrstice `P`, `Q`, `R`, lahko iz njih izluščimo kvadrat na levi:

```
P = [P1, P2, P3 | Ps]
Q = [Q1, Q2, Q3 | Qs]
R = [R1, R2, R3 | Rs]
K = [[P1, P2, P3],
     [Q1, Q2, Q3],
     [R1, R2, R3]]
```

Iz tega dobimo predikat, ki za vrstice `P`, `Q` in `R` pove, da so vsi trije kvadrati, ki tvorijo `P`, `Q`, `R`, permutacije:

```
kvadrati_permutacija([],[],[]).
kvadrati_permutacija(
   [P1, P2, P3 | Ps],
   [Q1, Q2, Q3 | Qs],
   [R1, R2, R3 | Rs]) :-
   all_distinct([P1, P2, P3, Q1, Q2, Q3, R1, R2, R3]),
   kvadrati_permutacija(Ps, Qs, Rs).
```

9.2.5 Končni izdelek

```
:- use_module(library(clpfd)).

% seznam L ima 9 elementov
length9(L) :- length(L, 9).

% ali Rows je pravilno izpolnjen Sudoku
sudoku(Rows) :-
        length9(Rows), % imamo devet vrstic
        maplist(length9, Rows), % vsaka vrstica ima 9 elementov
        append(Rows, Vs), Vs ins 1..9, % vsi elementi so med 1 in 9
        maplist(all_distinct, Rows), % vsaka vrstica je permutacija
        transpose(Rows, Columns), maplist(all_distinct, Columns), % vsak stolpec je␣
 ↪permutacija
        Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],
        blocks(As, Bs, Cs), % podkvadrati v vrsticah 1, 2, 3
        blocks(Ds, Es, Fs), % podkvadrati v vrsticah 4, 5, 6
        blocks(Gs, Hs, Is). % podkvadrati v vrsticah 7, 8, 9

% preveri kvadratke v treh danih vrsticah
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
blocks([], [], []).
blocks([N1,N2,N3|Ns1],
       [N4,N5,N6|Ns2],
       [N7,N8,N9|Ns3]) :-
        all_distinct([N1,N2,N3,N4,N5,N6,N7,N8,N9]),
        blocks(Ns1, Ns2, Ns3).

% pomožni predikat, ki naredi label na vseh elementih matrike
find(Rows) :-
    append(Rows, Vs), label(Vs).

example1([[_,_,_,_,_,_,_,_,_],
          [_,_,_,_,_,_,_,8,5],
          [_,_,1,_,2,_,_,_,_],
          [_,_,_,5,_,7,_,_,_],
          [_,_,4,_,_,_,1,_,_],
          [_,9,_,_,_,_,_,_,_],
          [5,_,_,_,_,_,_,7,3],
          [_,_,2,_,1,_,_,_,_],
          [_,_,_,_,4,_,_,_,9]]).

example2([[5,3,_,_,7,_,_,_,_],
          [6,_,_,1,9,5,_,_,_],
          [_,9,8,_,_,_,_,6,_],
          [8,_,_,_,6,_,_,_,3],
          [4,_,_,8,_,3,_,_,1],
          [7,_,_,_,2,_,_,_,6],
          [_,6,_,_,_,_,2,8,_],
          [_,_,_,4,1,9,_,_,5],
          [_,_,_,_,8,_,_,7,9]]).

example3([[1,2,3,4,5,6,7,8,9],
          [_,_,_,_,_,_,_,_,_],
          [_,_,_,_,_,_,_,_,_],
          [_,_,_,_,_,_,_,_,_],
          [_,_,_,_,_,_,_,_,_],
          [_,_,_,_,_,_,_,_,_],
          [_,_,_,_,_,_,_,_,_],
          [_,_,_,_,_,_,_,_,_],
          [_,_,_,_,_,_,_,_,_]]).

% Usage: ?- example1(Rows), sudoku(Rows), find(Rows), maplist(portray_clause, Rows).
```

POGLAVJE 10

Specifikacija, implementacija, abstrakcija

10.1 Specifikacija & implementacija

Specifikacija (angl. specification) `S` je *zahteva*, ki opisuje, kakšen izdelek želimo.

Implementacija (angl. implementation) `I` je izdelek. Implementacija `I` *zadošča* specifikaciji `S`, če ustreza zahtevam iz `S`.

V programiranju je implementacija programska koda. Specifikacije podajamo na različne načine in jih pogosto razvijemo postopoma:

- pogovor s stranko in analiza potreb
- dokumentacija, ki jo razume stranka
- tehnična dokumentacija za programerje

Brez specifikacije ne vemo, kaj je treba naprogramirati. Danes si bomo ogledali, kako v programskih jezikih poskrbimo za zapis specifikacij in kako programski jezik preveri, ali dana koda (implementacija) zadošča dani specifikaciji.

10.1.1 Signature in aksiomi

Omenimo še povezavo z algebro. V algebri poznamo *algebraične strukture*, na primer vektorske prostore, grupe, monoide, kolobarje, Boolove algebre, … Definicija takih struktur poteka v dveh korakih:

- **signatura** pove, kakšne množice, konstante in operacije imamo
- **aksiomi** povedo, kakšnim zakonom morajo zadoščati operacije

Primer

Matematično strukturo **grupa** opišemo takole:

- signatura:

  - množica G
  - operacija $\cdot : G \times G \to G$
  - operacija $^{-1} : G \to G$
  - konstanta $e : G$
- aksiomi:

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z$$
$$x \cdot e = x$$
$$e \cdot x = x$$
$$x \cdot x^{-1} = e$$
$$x^{-1} \cdot x = e$$

Primer

Matematično strukturo **usmerjen graf** opišemo takole:

- signatura:
  - množica V (vozlišča)
  - množica E (povezave)
  - operacija $\mathtt{src} : E \to V$ (začetno vozlišče povezave)
  - operacija $\mathtt{trg} : E \to V$ (končno vozlišče povezave)
- aksiomi: ni aksiomov

Zakaj vse to razlagamo? Ker programski jeziki ponavadi omogočajo zapis *signature* v programskem jeziku, ne pa tudi aksiomov, saj jih prevajalnik ne more preveriti.

10.2 Vmesniki

Specifikaciji včasih rečemo tudi **vmesnik (angl. interface)**, ker jo lahko razumemo kot opis, ki pove, kako se uporablja neko programsko kodo. Na primer, avtor programske knjižnice običajno objavi **API (Application Programming Interface)**, ki ni nič drugega kot specifikacija, ki pove, kako deluje knjižnica.

Torej imamo (vsaj) dve uporabi specifikacij:

- zahtevek za programsko kodo (specifikacija)
- protokol za uporabo programske kode (vmesnik)

10.2.1 Vmesniki v Javi

V Javi je specifikacija `S` podana z vmesnikom

```
public interface S {
   ...
}
```

v katerem lahko naštejemo metode. Tipe, ki nastopajo v specifikaciji, podamo kot generične razrede. Na primer, vmesnik za grupo bi zapisali takole:

```
public interface Group<G> {
    public G getUnit();
    public G multiply(G x, G y);
    public G inverse(G x);
}
```

Vmesnik za usmerjeni graf:

```
public interface Graph<V, E> {
   public V src(E e);
   public V trg(E e);
}
```

Seveda v praksi nihče ne piše takih vmesnikov, tu samo razmišljamo o zvezi med matematičnimi signaturami in vmesniki v programskih jezikih. Kasneje bomo videli bolj uporabne vmesnike, ki opisujejo abstraktne podatkovne strukture.

10.2.2 Vmesniki v OCamlu

V OCamlu lahko podamo poljubno signaturo (tipe in vrednost), ne moremo pa zapisati aksiomov, ki jim zadoščajo. Takole zapišemo signaturo za grupo:

```
module type GROUP =
sig
  type g
  val mul : g * g -> g
  val inv : g -> g
  val e : g
end
```

In takole za usmerjeni graf:

```
module type DIRECTED_GRAPH =
sig
  type v
  type e
  val src : e -> v
  val trg : e -> v
end
```

10.3 Implementacija

Programski jeziki seveda omogočajo implementacijo, kar ni nič drugega kot pisanje kode, ki ustreza dani specifikaciji. Zanimivo pa je vprašanje, kako so posamezne enote implementacije organizirane v različnih jezikih.

10.3.1 Implementacija v Javi

V Javi implementiramo vmesnik `I` tako, da definiramo razred `C`, ki mu zadošča:

```
public class C implements I {
   ...
}
```

Razred lahko hkrati zadošča več vmesnikom. (Opomba: podrazredi so mehanizem, ki se *ne* uporablja za specifikacijo.)

10.3.2 Implementacija v OCamlu

Implementacija v OCamlu se imenuje **modul (angl. module)**. Modul je skupek definicij tipov in vrednosti, lahko pa vsebuje tudi še nadaljnje podmodule.

Nekaj primerov (nepraktičnih) implementacij grup podajmo tu, kasneje pa bomo videli bolj uporabne primere:

```
(* Najprej definiramo signature. *)

module type GROUP =
sig
    type g
    val mul : g * g -> g
    val inv : g -> g
    val e : g
end

module type DIRECTED_GRAPH =
sig
    type v
    type e
    val src : e -> v
    val trg : e -> v
end

(* Signaturo implementiramo z modulom ali strukturo.
   Dana signatura ima lahko več implementacij. *)

module Z3 : GROUP =
struct

  type g = Zero | One | Two

  let e = Zero

  let plus = function
    | (Zero, y) -> y
    | (x, Zero) -> x
    | (One, One) -> Two
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
    | (One, Two) -> Zero
    | (Two, One) -> Zero
    | (Two, Two) -> One

  let mul = plus

  let inv = function
    | Zero -> Zero
    | One -> Two
    | Two -> One
end

module Z3' : GROUP =
struct

    type g = int

    let mul (x, y) = (x + y) mod 3
    let inv x = (3 - x) mod 3
    let e = 0
end

module K4 : DIRECTED_GRAPH =
struct

    type v = V0 | V1 | V2 | V3
    type e = E0 | E1 | E2 | E3 | E4 | E5

    let src = function
      | E0 -> V0
      | E1 -> V1
      | E2 -> V2
      | E3 -> V3
      | E4 -> V0
      | E5 -> V1

    let trg = function
      | E0 -> V1
      | E1 -> V2
      | E2 -> V3
      | E3 -> V0
      | E4 -> V2
      | E5 -> V3
end

module Cycle3 : DIRECTED_GRAPH =
struct
  type v = int (* uporabimo 0, 1, 2 *)
  type e = int (* uporabimo 0, 1, 2 *)
  let src e = e
  let trg e = (e + 1) mod 3

  (* Graf z vozlišči 0, 1, 2 in povezavami 0, 1, 2:
        src    trg
     0 : 0 --> 1
     1 : 1 --> 2
     2 : 2 --> 0
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
  *)
end

(* Takole pa naredimo modul, ki je parametriziran s
   strukturo. Kasneje bomo videli bolj uporabne primere. *)
module Cycle (S : sig val n : int end) : DIRECTED_GRAPH =
struct
  type v = int
  type e = int
  let src k = k
  let trg k = (k + 1) mod S.n
end

module C5 = Cycle(struct let n = 5 end)
module C15 = Cycle(struct let n = 15 end)
```

10.4 Abstrakcija

Ko gradimo večje programske sisteme, so ti sestavljeni iz enot, ki jih povezujemo med seboj. Za vsako enoto je lahko zadolžena ločena ekipa programerjev. Programerji opišejo programske enote z *vmesniki*, da vedo, kaj kdo počne in kako uporabljati kodo ostalih ekip.

A to je le del zgodbe. Denimo, da prva ekipa razvija programsko enoto `E`, ki zadošča vmesniku `S` in da druga ekipa uporablja enoto `E` pri izdelavi svoje programske enote. Dobra programska praksa pravi, da se druga ekipa ne sme zanašati na podrobnosti implementacije `E`, ampak samo na to, kar je zapisano v specifikaciji `S`. Na primer, če `E` vsebuje pomožno funkcijo `f`, ki je `S` ne omenja, potem je druga ekipa ne sme uporabljati, saj je `f` namenjena *notranji* uporabi `E`. Prva ekipa lahko `f` spremeni ali zbriše, saj `f` ni del specifikacije `S`.

Če sledimo načelu, da mora programski jezik neposredno podpirati aktivnosti programerjev, potem bi želeli *skriti* podrobnosti implementacije `E` tako, da bi lahko programerji druge ekipe imeli dostop *samo* do tistih delov `E`, ki so našteti v `S`.

Kadar *skrijemo* podrobnosti implementacije, pravimo, da je implementacija **abstraktna**.

Programski jeziki omogočajo abstrakcijo v večji ali manjši meri:

- Java nadzoruje dostopnost do komponent z določili `private`, `public` in `protected`
- Python omogoča skrivanje s poimenovanjem `__xyz` (glej name mangling).
- OCaml omogoča abstrakcijo z določilom `M : S`, kjer je `M` module in `S` signatura. S tem skrijemo vsebino modula `M`, razen tistih komponent, ki so naštete v `S`.

10.5 Generično programiranje

Z izrazom *generično programiranje* razumemo kodo, ki jo lahko uporabimo večkrat na različne načine. Na primer, če napišemo knjižnico za 3D grafiko, bi jo želeli uporabljati na več različnih grafičnih karticah. Ali bomo za vsako grafično kartico napisali novo različico knjižnice? Ne! Želimo **generično** implementacijo, ki bo preko ustreznega *vmesnika* dostopala do grafične kartice. Proizvajalci grafičnih kartic bodo implementirali *gonilnike*, ki bodo zadoščali temu vmesniku.

10.5.1 Generično programiranje v Javi

Java podpira generično programiranje. Ko definiramo razred, je ta lahko odvisen od kakega drugega razreda:

```
public class Knjiznica3D<Driver extends GraphicsDriver> {
  ...
}
```

10.5.2 Generično programiranje v OCamlu

V OCamlu je generično programiranje omogočeno s **funktorji (angl. functor)** (opomba: v matematiki poznamo funktorje v teoriji kategorij, ki nimajo nič skupnega s funktorji v OCamlu).

Funktor je preslikava iz struktur v strukture in je bolj splošen kot generični razredi v Javi (ker lahko struktura vsebuje podstrukture in definicije več tipov, razred pa ne more vsebovati definicij podrazredov).

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

```
module F(A : S) =
struct
  (* definicija strukture B *)
end
```

Zgoraj smo videli primer preprostega funktorja `Cycle`, ki sprejme strukturo s številom `n` in vrne usmerjeni cikel na `n` vozliščih. Bolj uporaben primer sledi.

Primer

Kot primer uporabe modulov in funktorjev v OCamlu priporočamo MirageOS. To je sistem za gradnjo unijeder (angl unikernel) – minuaturinih operacijskih sistemov, ki opravljajo samo eno opravilo, npr. odgovorijo na DNS request. So zelo majhni in za zagon potrebujejo le nekaj milisekund, MirageOS pa jih gradi z intenzivno uporabo OCaml modulov in funktorjev.

10.6 Primer: prioritetne vrste

Prioritetna vrsta je podatkovna struktura, v katero dodajamo elemente, ven pa jih jemljemo glede na njihovo *prioriteto*. Zapišimo specifikacijo:

Signatura:

- podatkovni tip `element`
- operacija `priority : element → int`
- podatkovni tip `queue`
- konstanta `empty : queue`
- operacija `put : element → queue → queue`
- operacija `get : queue → element option * queue`

Aksiomov ne bomo pisali, ker bi morali v tem primeru spoznati bolj zahtevne jezike za specifikacijo, ki presegajo okvir te lekcije. Neformalno pa lahko opišemo zahteve za prioritetno vrsto:

- `element` je tip elementov, ki jih hranimo v vrsti

- `priority x` vrne prioriteto elementa `x`, ki je celo število. Manjše število pomeni »prej na vrsti«
- `queue` je tip prioritetnih vrst
- `empty` je prazna prioritetna vrsta, ki ne vsebuje elementov
- `put x q` vstavi element `x` v vrsto `q` glede na njegovo prioriteto in vrne tako dobljeno vrsto
- `get q` vrne `(Some x, q')` kjer je `x` element iz `q` z najnižjo prioriteto in `q'` vrsta `q` brez `x` Operacija `get` vrne `(None, q)`, če je `q` prazna vrsta.

10.6.1 Implementacija v OCamlu

Oglejmo si implementacijo v OCamlu.

```
module type PRIORITY_QUEUE =
  sig
    type element
    val priority : element -> int
    type queue
    val empty : queue
    val put : element -> queue -> queue
    val get : queue -> element option * queue
  end

module MyFirstQueue : PRIORITY_QUEUE with type element = int * int =
  struct
    type element = int * int

    let priority (a, b) = a

    type queue = element list

    let empty = []

    let rec put x = function
      | [] -> [x]
      | y :: ys ->
         if priority x <= priority y then
           x :: y :: ys
         else
           y :: put x ys

    let get = function
      | [] -> (None, [])
      | x :: xs -> (Some x, xs)
  end

module type PRIORITY =
sig
  type t
  val priority : t -> int
end


(* Implementacija prioritetne vrste s seznami. To je funktor, ki
   sprejme tip elemetov in prioritetno funkcijo. *)
module ListQueue (M : PRIORITY) : PRIORITY_QUEUE with type element = M.t
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
  =
  struct
    type element = M.t

    let priority = M.priority

    type queue = element list

    let fortytwo = 42

    let empty = []

    let rec put x = function
      | [] -> [x]
      | y :: ys ->
         if priority x <= priority y then
           x :: y :: ys
         else
           y :: put x ys

    let get = function
      | [] -> (None, [])
      | x :: xs -> (Some x, xs)
  end

(* Naredimo prioritetno vrsto nizov, prioriteta je dolžina niza. *)
module A =
  ListQueue(
      struct
        type t = string
        let priority = String.length
      end)

(* Preizkus. *)
let example1 =
  A.get (A.put "kiwi" (A.put "jabolko" (A.put "banana" A.empty)))

(* Naredimo prioritetno vrsto nizov, prioriteta je dolžina niza. *)
module A' =
  ListQueue(
      struct
        type t = string
        let priority s = - String.length s
      end)

(* Preizkus. *)
let example1' =
  A'.get (A'.put "kiwi" (A'.put "jabolko" (A'.put "banana" A'.empty)))

(* Naredimo prioritetno vrsto parov števil. *)
module B =
  ListQueue(
      struct
        type t = int * int
        let priority (a,b) = a
      end
    )
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
module IntQueue =
  ListQueue(
    struct
      type t = int
      let priority k = k
    end
  )


(* Učinkovita implementacija z levičarskimi kopicami,
   glej https://en.wikipedia.org/wiki/Leftist_tree.
   Implementacija je abstraktna, ker uporabimo :,
   vendar dodamo določilo, da je tip element enak tipu t.
 *)
module LeftistHeapQueue (M : PRIORITY)
       : PRIORITY_QUEUE with type element = M.t =
  struct
    type element = M.t

    let priority = M.priority

    type queue = Leaf | Node of int * element * queue * queue

    let rank = function
      | Leaf -> 0
      | Node (r, _, _, _) -> r

    let node (x, a, b) =
      if rank a < rank b then
        Node (1 + rank a, x, b, a)
      else
        Node (1 + rank b, x, a, b)

    let rec meld a b =
      match (a, b) with
      | (_, Leaf) -> a
      | (Leaf, _) -> b
      | (Node (_, ka, la, ra), Node (_, kb, lb, rb)) ->
         if priority ka < priority kb then
           node (ka, la, meld ra b)
         else
           node (kb, lb, meld a rb)

    let singleton x = Node (1, x, Leaf, Leaf)

    let empty = Leaf

    let put x q = meld q (singleton x)

    let get = function
      | Leaf -> (None, Leaf)
      | Node (_, y, l, r) -> (Some y, meld l r)
  end


module D = LeftistHeapQueue(
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
              struct
                type t = int * int
                let priority (x, y) = x + y
              end)

let example2 =
  let rec loop q = function
    | 0 -> D.put (0, 0) q
    | k -> loop (D.put ((47 * k * k + 13) mod 1000, k) q) (k - 1)
  in
  loop D.empty 300000

let rec to_list q =
  match D.get q with
  | (None, _) -> []
  | (Some x, q) -> x :: to_list q
```

10.6.2 Implementacija v Javi

Oglejmo si še implementacijo v Javi. V tem jeziku je bolj naravno narediti vrste kot objekte, ki se spreminjajo. Torej spremenimo specifikacijo.

Signatura:

- podatkovni tip `Element`
- metoda `priority : element → int`
- podatkovni tip `queue`
- operacija `empty : unit → queue`
- operacija `is_empty : queue → bool`
- operacija `put : element → queue → unit`
- operacija `get : queue → element option`

Zahteve so podobne kot prej, le da operacije `empty`, `put` in `get` delujejo nekoliko drugače:

- `empty ()` vrne nov primerek (objekt) prazne vrste
- `put x q` vstavi `x` v vrsti `q` in s tem *spremeni* `q`
- `get q` vrne prvi `x` v vrsti `q` in s tem *spremeni* `q`

Zgornjo specifikacijo predelamo v dva vmesnika. Prvi je vmesnik za razrede, ki imajo metodo `priority`:

```
public interface Priority {
        public int priority();
}
```

Vmesnik `PriorityQueue` pa podaja specifikacijo za prioritetno vrsto:

```
public interface PriorityQueue<Element extends Priority> {
    public PriorityQueue<Element> emptyQueue();
    public boolean isEmpty();
    public void put(Element x);
    public Element get();
}
```

Še primer implementacije prioritetnih vrst s seznami:

```
import java.util.LinkedList;

public class ListQueue<Element extends Priority> implements PriorityQueue<Element> {
    private LinkedList<Element> elements;

    public ListQueue() {
        elements = new LinkedList<Element>();
    }

    @Override
    public boolean isEmpty() {
        return elements.isEmpty();
    }

    @Override
    public void put(Element x) {
        int i = 0;
        for (Element y : elements) {
            if (x.priority() < y.priority()) {
                break;
            } else {
                i += 1;
            }
        }
        elements.add(i, x);
    }

    @Override
    public Element get() {
        return elements.removeFirst();
    }

    @Override
    public PriorityQueue<Element> emptyQueue() {
        return new ListQueue<Element>();
    }
}
```

10.7 Primer: množice

Na vajah boste na dva načina implementirali končne množice, opisane z naslednjo signaturo: signaturo.

```
type order = Less | Equal | Greater

module type SET =
sig
    type element
    val cmp : element -> element -> order
    type set
    val empty : set
    val member : element -> set -> bool
    val add : element -> set -> set
    val remove : element -> set -> set
end
```

POGLAVJE 11

Podtipi

Spoznali smo že *polimorfizem*, to je lastnost, da ima lahko izraz več kot en tip. Na primer, v OCamlu ima preslikava

```
fun (x, y) -> (y, x)
```

tip `α × β → β × α`, kjer sta `α` in `β` poljubna tipa. Danes bomo spoznali **podtipe**, ki delujejo podobno: če ima izraz `e` tip `t`, ga lahko uporabljamo, kot da bi imel tudi vse nadtipe tipa `t`.

Na primer, v nekaterih programskih jezikih lahko izraz `e` tipa `int` vedno uporabimo, kot da bi ime tip `float`, saj bo jezik samo pretvoril `e`.

V Javi poznamo *podrazrede*, ki so tudi vrsta podtipov, a jih bomo obravnavali prihodnjič.

11.1 Relacija podtip `A ≤ B`

Najprej se dogovorimo za zapis relacije „podtip“. Za tipa A in B zapis

$$A \leq B$$

preberemo »A je podtip B« ali »B je nadtip A«. Sledimo naslednjemu osnovnemu načelu:

Pomembno

A je podtip B, če lahko vrednosti tipa A uporabljamo, kot da bi imele tip B.

Pravilo sklepanja, ki izraža zgornje načelo je

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

in preberemo: »Če ima e tip A in je A podtip B, potem ima e tudi tip B.«

> **Opozorilo**
>
> Zmotno bi si bilo predstavljati, da je $A \leq B$ isto kot $A \subseteq B$, se pravi, da je podtip ista reč kot podmnožica. V razdelku o podtipih zapisov bomo namreč spoznali podtipe, ki se razlikujejo od relacije podmnožica.

Zapišimo pravila, ki izražajo pričakovane lastnosti relacije $\leq$. Vsekakor je relacija *refleksivna* in *tranzitivna* (taki relaciji pravimo *šibka urejenost*):

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

Kaj pa antisimetričnost: če $A \leq B$ in $B \leq A$, potem $A = B$? Ne, kasneje bomo videli protiprimer, torej $\leq$ ni *delna urejenost*.

Nadalje zapišemo še pravila, ki opredeljujejo, kako se obnašajo podtipi za razne operacije. Kartezični produkti delujejo po pričakovanjih:

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

Pravilo za funkcijske tipe je bolj zanimivo, pozorno ga preberite:

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

Obrnili smo vrsti red v domeni! Pravimo, da je $\to$ **kontravarianten** (obrača smer) v prvem argumentu in **kovarianten** (ohranja smer) v drugem. Skupaj premislimo, zakaj pričakujemo tako pravilo.

Kaj pa osnovni tipi? Ali je na primer `int` $\leq$ `float`? To je odvisno od programskega jezika: Java samodejno pretvori celo število v realno, zato bi lahko rekli, da v javi velja `int` $\leq$ `float`. V OCamlu to ne drži, saj mora programer sam pretvoriti celoštevilsko vrednost v realno s funkcijo `float`.

Lahko se celo zgodi, da v programskem jeziku velja `int` $\leq$ `float` *in hkrati* `float` $\leq$ `int`, na primer, če jezik samodejno zaokroži realno vrednost, kadar potrebuje celoštevilsko. A takih jezikov ni veliko.

11.2 Podtipi zapisov

Spoznali smo že zapise, to so nabori polj s poimenovanimi komponentami. Na primer, točke v ravnini lahko predstavimo z vrednostmi tipa zapisov

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

Primer vrednosti tega tipa je zapis `{x = 3.14; y = 2.78}`.

V splošnem je tip zapisa

```
{ ℓ₁ : A₁; ℓ₂ : A₂; …; ℓ_n : A_n }
```

njegove vrednosti pa so oblike

```
{ ℓ₁ = e₁; ℓ₂ = e₂; …; ℓ_n = e_n }
```

kjer mora veljati $e_i : A_i$ za vse $1 \leq i \leq n$ Zahtevamo še, da so imena polj `ℓ₁`, …, `ℓ_n` med seboj paroma različna.

Če je `v` zapis tipa `{ ℓ₁ : A₁; ℓ₂ : A₂; …; ℓ_n : A_n }`, potem je `v.ℓ`⍰ vrednost njegove `i`-te komponente, ki ima tip `A`⍰.

V zapisu vrstni red polj ni pomemben, to je, `{x = 3.14; y = 2.78}` in `{y = 2.78; x = 3.14}` sta enaki vrednosti.

Zapisi so uporaben podatkovni tip, ne samo v programiranju, ampak tudi na drugih področjih računalništva, kjer želimo predstviti strukturirane podatke. Na primer, vrstica v tabeli podatkovne baze ni nič drugega kot zapis (in shema za tabelo tip zapisa).

Tudi objekti nas nekoliko spominjajo na zapise, le da vsebujejo poleg polj (atributov) še metode:

```
public class Point {
   float x;
   float y
   ...
}
```

11.2.1 Podtipi zapisov v širino

Zapisi tipov imajo zanimivo relacijo $\leq$. Obravnavajmo primer. Na primer, da imamo tipa zapisov

$$A = \{x : \texttt{float}; y : \texttt{float}\}$$

in

$$B = \{x : \texttt{float}; y : \texttt{float}; z : \texttt{float}\}$$

Ali morda velja $A \leq B$ in $B \leq A$?

Najprej, $A \leq B$ *ne* velja. Izraz $a = \{x = 3.14; y = 2.78\}$ ima tip A. Če bi ga lahko uporabljali, kot da ima tip B, potem bi smeli pisati $a.z$, kar seveda ne gre, saj a nima polja z.

Velja pa $B \leq A$! Izraz $b = \{x = 3.14; y = 2.78; z = 1.0\}$ ima tip B in res ga lahko uporabimo, kot da bi imel tip A, saj smemo pisati $b.x$ in $b.y$. Dejstvo, da $\texttt{b}$ vsebuje še polje $\texttt{z}$ nas pri tem nič ne moti.

Zapišimo pravilo za podtipe zapisov, ki izhaja iz zgornjega razmisleka:

$$\frac{\text{za vsaj } j \leq m \text{ obstaja } i \leq n, \text{ da je } \ell_i = k_j \text{ in } A_i = B_j}{\{\ell_1 : A_1; \ldots; \ell_n : A_n\} \leq \{k_1 : B_1; \ldots; k_m : B_m\}}$$

Povedano z besedami, prvi tip zapisa je podtip drugega, če se vsako polje $k_j : B_j$ iz drugega zapisa pojavi v prvem. Tej vrsti podtipov pravimo **podtip zapisa po širini**, ker je podtip »širši« (ima več polj) kot njegov podtip.

Velja torej:

$$\{z : \texttt{float}; x : \texttt{float}; y : \texttt{float}\} \leq \{x : \texttt{float}; y : \texttt{float}\}$$

Vaja

Ali obstaja tip zapisa, ki je podtip vseh ostalih tipov zapisov? Kaj pa tip zapisa, ki je nadtip vseh ostalih tipov zapisov?

11.2.2 Podtipi zapisov v globino

Poznamo še **podtipe zapisov v globino**. Spet poglejmo primer, pri čemer predpostavimo, da velja `int` $\leq$ `float`. Zapis

$$v = \{x = 3; y = 5\}$$

ima tip $\{x : \texttt{int}; y : \texttt{int}\}$. Ali ga lahko uporabljamo, kot da bi imel tudi tip $\{x : \texttt{float}; y : \texttt{float}\}$? Da, saj lahko njegovi komponenti $v.x$ in $v.y$ uporabljamo, kot da bi imeli tip `float`, zahvaljujoč `int` $\leq$ `float`.

Pravilo, za podtipe zapisov v globino se glasi:

$$\frac{A_1 \leq B_1 \quad A_2 \leq B_2 \quad \cdots \quad A_n \leq B_n}{\{\ell_1 : A_1; \ldots; \ell_n : A_n\} \leq \{\ell_1 : B_1; \ldots; \ell_n : B_n\}}$$

Obe pravili, za širino in globino, lahko združimo v eno kombinirano pravilo:

$$\frac{\text{za vsak } j \leq m \text{ obstaja } i \leq n, \text{ da je } \ell_i = k_j \text{ in } A_i \leq B_j}{\{\ell_1 : A_1; \ldots; \ell_n : A_n\} \leq \{k_1 : B_1; \ldots; k_m : B_m\}}$$

11.2.3 Zapisi s spremenljivimi polji

Katera pravila za podtipe zapisov pridejo v poštev, je odvisno od tega, kako lahko zapise uporabljamo. Denimo, če imamo zapise s *spremenljivimi* polji, se pravi, da lahko zapisu spremenjamo vrednosti polj, potem podtipi v globino niso več veljavni. Na primer, če imamo tipa

```
A = { mutable x : int; mutable y : int}
```

in

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

potem *ne* velja `A <= B`. Če bi to veljalo, bi lahko v zapis `v : A` vrednost polja `x` nastavili na `3.14`. Zapišimo tip `A` še v OCamlu:

```
type a = { mutable x : int; mutable y : int}
```

Takole naredimo vrednost in ji nastavimo atribut:

```
# let p = {x = 10; y = 20} ;;
val p : a = {x = 10; y = 20}
# p.x <- 42 ;;
- : unit = ()
# p ;;
- : a = {x = 42; y = 20}
```

11.2.4 Problem koherentnosti

Težave nastopijo, če lahko vrednosti tipa A pretvorimo v nadtip B na več načinov, se pravi, če imamo

$$A \leq B$$
$$A \leq C$$
$$B \leq D$$
$$C \leq D$$

Zaradi tranzitivnosti sledi $A \leq D$, kar lahko izpeljemo na *dva* načina:

1. $A \leq B \leq D$
2. $A \leq C \leq D$

To pa lahko vpliva na implicitne pretvorbe. Če imamo $e : A$, ga lahko pretvorimo v D preko pretvorb $A \to B \to D$ ali preko $A \to C \to D$. Kako vemo, da bomo obakrat dobili isto? Temu pravimo problem koherentnosti. Pojavi se vedno, ko imamo implicitna (ali samodejne) pretvorbe vrednosti iz enega tipa v drugega.

11.3 Podtipi na nivoju struktur

Strukture ali moduli so v resnici neke vrste zapisi na višjem nivoju. Zanje veljajo pravila za podtipe, kar preizkusimo na primerih.

```
type color = { r : float; g : float; b : float }

module type POINT =
sig
  val x : float
  val y : float
end

module type COLOR_POINT =
sig
  val x : float
  val y : float
  val c : color
end

module Pika : POINT =
struct
  let x = 1.2
  let y = 3.4
end

module BarvnaPika : COLOR_POINT =
struct
  let x = 1.2
  let y = 3.4
  let c = {r = 1.0; g = 0.5; b = 0.0}
end

module F (P : POINT) = struct
  let z = P.x +. P.y
end

module G (P : COLOR_POINT) = struct
  let c = P.c.r +. P.c.g
end

(* COLOR_POINT ≤ POINT *)

module A = F (Pika)
module B = F (BarvnaPika)
module C = G (BarvnaPika)
(* ne deluje: module C = G (Pika) *)
```

POGLAVJE 12

Objektno programiranje

12.1 Objekti

Objekti so podobni zapisom, le da imajo atribute in metode. Metode se lahko sklicujejo na atribute, kakor tudi na druge metode (z uporabo `this` ali `self`).

Torej so objekti neke vrste **rekurzivni zapisi**. Rekurzivni, ker se lahko objekt skliče sam nase.

12.1.1 Objekti v OCamlu

V OCamlu in nekaterih drugih lahko objekte naredimo neposredno

```
object (this)
   val x = ...
   val y = ...
   method f = ...
   method g = ...
end
```

Funkcionalnost objekta (njegove metode) opišemo s tipom zapisa. S podtipi lahko objekte umeščamo v hierarhijo.

Več primerov bomo obdelali na vajah, poglejmo primere:

```
(* Objekt lahko definiramo neposredno tako, da podamo njegove
   atribute in metode. *)
let p =
  object
    (* atributi, v OCamlu se imenujejo "instance variable" *)
    val mutable x = 10
    val mutable y = 20

    (* metode *)
    method get_x = x
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
    method get_y = y

    method move dx dy =
      x <- x + dx ;
      y <- y + dy
  end

(* Če želimo klicati eno eno metodo iz druge, moramo podati tudi ime,
   s katerim se objekt sklicuje sam nase. *)
let q =
  object (this) (* namesto "this" bi lahko uporabili kako drugo ime *)
    val mutable x = 10
    val mutable y = 20

    method get_x = x

    method get_y = y

    method move dx dy =
      x <- this#get_x + dx ;  (* tu se sklicujemo na metodo get_x *)
      y <- y + dy
  end

(* Funkcija, ki sprejme začetni vrednosti atributov x in y ter
   vrne ustrezno inicializiran objekt. To je zelo podobno konstruktorjem
   v Javi, a je le funkcija. *)
let make_point x0 y0 =
  object
    val mutable x = x0
    val mutable y = y0

    method get_x = x

    method get_y = y

    method move dx dy =
      x <- x + dx ;
      y <- y + dy
  end

(* S tako funkcijo naredimo nov objekt. *)
let r = make_point 1 3

(* OCaml pozna tudi razrede, ki jih definiramo, kot da bi bili funkcije, ki
   vračajo objekte (vendar to niso). *)
class point x0 y0 =
  object
    val mutable x = x0
    val mutable y = y0

    method get_x = x

    method get_y = y

    method move dx dy =
      x <- x + dx ;
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
      y <- y + dy
  end

(* Z "new" naredimo nov objekt danega razreda. *)
let s = new point 1 3

(* Primerjajmo tipa objektov r in s:

   r : < get_x : int; get_y : int; move : int -> int -> unit >
   s : point

   Tip r je podoben tipu zapisa, saj našteje metode (ne pa atributov, ki so
   popolnoma skriti). Tip s je kar ime razreda, ki mu s pripada.

   Ker razred point definira natanko iste metode, kot tip r, sta r in s
   pravzaprav istega tipa, "point" je samo okrajšava.
*)

(* Poglejmo še, kako deluje dedovanje in podrazredi. *)

type color = { r : float; g : float ; b : float }

let pink = { r = 1.0; g = 0.75; b = 0.75 }

(* Razred color_point podeduje razred point. *)
class color_point x0 y0 (c0 : color) =
object
  inherit point x0 y0 (* dedovanje *)

  val color = c0

  method get_color = color
end
```

Poglej Tudi

OCaml ima zelo bogat nabor konstrukcij za objektno programiranje, več o njih preberite v dokumentaciji Objects in OCaml.

12.2 Objektno programiranje z razredi

Mnogi objektni jeziki pa poznajo mehanizem **razredov** (Java, C++, C#). Poglejmo še enkrat primer točke:

```
let p =
  object
    val mutable x = 10
    val mutable y = 20

    method get_x = x

    method get_y = y

    method move dx dy =
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
      x <- x + dx ;
      y <- y + dy
  end
```

Zgornja koda naredi le *eno* točko. Če želimo narediti več točk, lahko napišemo funkcijo, ki sprejme začetno pozicijo in vrne objekt. Lahko pa definiramo razred, na primer v Javi:

```
public class Point {
    private int x ;
    private int y ;

    public Point(int x0, int y0) {
        this.x = x0;
        this.y = y0;
    }

    public int get_x() { return this.x; }

    public int get_y() { return this.y; }

    public void move(int dx, int dy) {
        this.x += dx;
        this.y += dy;
    }
}
```

Sedaj tvorimo nove točke s *konstruktorjem*: `new Point(10, 20)`.

Tudi v OCamlu lahko definiramo razred:

```
class point x0 y0 =
  object
    val mutable x = x0
    val mutable y = y0

    method get_x = x

    method get_y = y

    method move dx dy =
      x <- x + dx ;
      y <- y + dy
  end
```

In tvorimo novo točko s konstruktorjem: `new point 10 20`.

Razredi niso samo bližnjica za konstruktorje, ampak omogočajo še mnoge druge mehanizme:

- enkapsulacija
- konstruktor
- dedovanje
- vmesniki
- prekrivanje metod
- preobteževanje metod

- generične metode in razredi
- abstrakne metode in razredi
- …

Vsi ti mehanizmi lahko delo z razredi naredijo precej zapleteno. Verjetno je eden od razlogov za kompleksnost ta, da programski jeziki kot so Java, C# in C++ *vse* mehanizme za organizacijo kode izražajo s pomočjo razredov. Tako Java ne pozna modulov in funktorjev, algebrajskih podatkovnih tipov, ali parametričnega polimorfizma – vse to nadomešča z razredi.

Naredimo kratek pregled osnovnih mehanizmov objektnega programiranja z razredi in jih primerjajmo s koncepti, ki smo jih spoznali do sedaj.

12.2.1 Enkapsulacija

Enkapsulacija je skrivanje stanja objekta, se pravi, da naredimo atribute (lahko pa tudi metode) nedostopne zunaj razreda. V Javi to naredimo z določilom `private`.

V OCamlu so atributi vedno privatni. Metodo naredimo privatno z določilom `private`.

Pojem *enkapsulacija* včasih zajema tudi idejo, da objekt poleg stanja (atributov) s seboj nosi tudi metode za delo z njimi.

Skrivanje definicij lahko implementiramo tudi z lokalnimi definicijami in signaturami, ki zakrijejo implementacijo.

12.2.2 Konstruktor

Konstruktor je del kode, ki nastavi začetne vrednosti atributov objekta. Običajno konstruktor sprejme argumente, se pravi, da je to funkcija.

V Javi je ime konstruktorja enako imenu razreda, prav tako v C++ in OCamlu.

12.2.3 Dedovanje

En razred lahko *deduje* od drugega:

```
public class A extends B { ... }
```

To pomeni, da ima `A` vse, kar ima `B` (torej atribute in metode).

OCaml dopušča večkratno dedovanje, ko en razred deduje hkrati od več nadrazredov.

12.2.4 Vmesniki

Vmesniki predpisujejo funkcionalnost, ki jo mora imeti objekt. V Javi vmesnik definiramo z

```
interface I { ... }
```

in od razreda zahtevamo, da zadosti vmesniku `I` (lahko tudi več)

```
class C implements I { ... }
```

O vmesnikih smo že govorili.

12.2.5 Prekrivanje

Razred lahko nekatere od podedovanih metod *prekrije* z lastnimi definicijami.

Tu se pojavi vprašanje, kako dostopati do prekritih metod, saj jih včasih potrebujemo. V Javi to naredimo s `super.imeMetode(…)`. Podobno vprašanje se pojavi pri konstruktorjih: kako iz konstruktorja pokličemo konstruktor iz nadrazreda?

12.2.6 Preobteževanje

Če imamo več metod z istim imenom `f`, pravimo, da smo `f` *preobtežili* (angl. overload). Metode se morajo med seboj razlikovati po številu argumentov ali njihovih tipih, sicer iz klice metode ne moremo razbrati, katero različico želimo.

OCaml ne pozna preobteževanja. Lahko pa uporabimo module in jih odpremo lokalno z `let open M in …` ali `M.(…)`.

12.2.7 Generične metode in razredi

Generične metode in razredi so parametrizirani z razredi ali vmesniki, takole:

```
public class A<B> {
  …
}
```

Definicija razreda `A` je *parametrizirana* z razredom `B`.

V C++ v ta namen uporabljamo predloge (angl. templates).

V OCaml poznamo tri mehanizme za generično programiranje: parametrični polimorfizem, moduli in parametrizirani razredi.

12.2.8 Abstraktne metode in razredi

Abstraktne (v C++ se imenujejo *virtualne*) so metode, ki jih ne implementiramo, ampak samo deklariramo. Se pravi, povemo njihov tip ne pa tudi implementacije.

Če razred vsebuje abstraktno metodo, mora biti tudi sam deklariran kot abstrakten. Objektov abstraktnega razreda ne moremo ustvarjati z `new`, saj so neke vrste „nedokončani“ razredi.

Abstraktni razred je neke vrste mešanica implementacije in speficikacije (specifikacija sestoji iz abstraktnih metod).

Razredi v OCamlu so lahko abstraktni, pravi se jim virtualni razredi.

POGLAVJE 13

Haskell in razredi tipov

Danes bomo spoznali programski jezik Haskell in razrede tipov (angl. type classes), ki so še en način za organizacijo kode in funkcionalnosti. Haskell je deklarativni jezik, ki se odlikuje po tem, da je tudi **čist** (angl. pure), kar pomeni, da v osnovi nima računskih učinkov (stanje, I/O, izjeme). O računskih učinkih in o tem, kako so implementirani z monadami, bomo govorili naslednjič.

Haskell lahko opredelimo kot programski jezik, ki je:

- deklarativni jezik z leno evaluacijo
- je funkcijski jezik: funkcije so vrednosti
- ima koinduktivne tipe in zapise
- ima parametrični polimorfizem in izpeljavo tipov
- čist jezik: vsi računski učinki (stanje, I/O, izjeme) so eksplicitno zavedeni v tipu izraza
- ima razrede tipov

13.1 Osnovno o Haskellu

Mnoge koncepte, ki nastopaj v programskem jeziku Haskell, že poznamo. Danes bomo spoznali

13.1.1 Konkretna sintaksa

Zamikanje

V Haskellu je treba kodo pravilno zamikati, podobno kot v Pythonu. V nekaterih primerih se lahko zamikanju izognemo z uporabo alternativne sintakse `{ ... ; ... ; ... }`. Primer bomo videli kasneje.

Imena

- Imena spremenljivk se pišejo z malo začetnico: `x`, `y`, …
- Imena tipov se pišejo z veliko začetnico: `Int`, `Bool`, `Char`, …
- Parametri v polimorfnih tipih se pišejo z malo začetnico: `a`, `b`, `c`, …
- Imena konstruktorjev algebraičnih tipov se pišejo z veliko

Definicije

Vrednost `t` tipa `A` zapišemo

```
t :: A
t = definicija-t
```

Pozor: zapis `t :: A` pomeni »`t` ima tipa `A`« zapis `x : ℓ` pa pomeni seznam z glavo `x` in repom ℓ (ravno obratno kot v OCamlu).

Definicija ima lahko tudi več vrstic, na primer:

```
f :: Int -> Int
f 0 = 1
f 1 = 1
f n = n * f (n - 1)
```

Definicije so lahko rekurzivne.

Tip v definiciji ni treba podati in smemo zapisati samo

```
t = defincija-t
```

V tem primeru bo Haskell izpeljal tip `t`. Vendar pa je v Haskellu navada, da se zapiše tip vrednosti, ki jo definiramo.

Lokalne definicije

Lokalno definicijo zapišemo

```
let x = e₁
in
  e₂
```

ali

```
e₂ where x = e₁
```

Določilu `where` lahko sledi več definicij:

```
e₁ where
  x = e₂
  y = e₃
  z = e₄
```

Seznami

Prazen seznam zapišemo z `[]`, stik glave `x` z repom ℓ zapišemo `x : ℓ`, elemente lahko tudi naštejemo z `[x₁, x₂, ..., x]`.

Seznam števil od `1` do `n` zapišemo `[1..n]` in podobno za interval `[a..b]`.

Haskell ima izpeljane sezname (angl. list comprehension), podobno kot Python:

- matematika: $\{f(x) \mid x \in A\}$ je množica elementov $f(x)$, kjer je $x \in A$
- Python `[f(x) for x in ℓ]` je seznam elementov `f(x)`, kjer `x` preteče seznam ℓ
- Haskell `[f x | x <- ℓ]` je seznam elementov `f x`, kjer `x` preteče seznam ℓ

Podrobnosti o izpeljanih seznamih si preberite na zgornji povezavi.

Funkcije

Funkcijo $x \mapsto e$ zapišemo kot `\ x -> e`, kar nas spominja na $\lambda\ x\ .\ e$.

Naloga: Zakaj se ta programski jezik imenuje Haskell?

Izraz `case`

Izraz

```
case e of
  p₁ -> e₁
  p₂ -> e₂
  ...
  p -> e
```

primerja `e` z vzorci `p₁, ..., p`. Vrednost izraza je enaka `e`, kjer je `p` prvi vzorec, ki se ujema. Torej je `case` podoben izrazu `match` iz OCaml.

> **Opozorilo**
>
> OCaml opozori na manjkajoče primere v `match`, Haskell tega ne počne. Če uporabimo opcijo `-Wall` (vsa opozorila), nas opozori na manjkajoče primere.

Primeri morajo biti pravilno zamaknjeni, lahko pa uporabimo tudi sintakso

```
case e of { p₁ -> e_1 ; ... ; p -> e }
```

ki ne zahteva zamikanja. V Haskellu velja navada, da raje zamikamo kodo, kot da bi uporabljali `{ ... }`.

13.1.2 Tipi

Imena tipov pišemo z velikimi črkami

Osnovni konstruktorji tipov

- Osnovni tipi so `Int`, `Char`, `Bool`, ...
- `a -> b` je tip funkcij iz `a` v `b`
- `(a, b)` je produktni tip, ki ga v Ocamlu pišemo `a * b`
- `[a]` je tip seznamov elementov tipa `a`, v OCamlu `a list`

Definicije tipov

Haskell pozna definicije tipov

```
type T = ...
```

in definicije (koinduktivnih) algebraičnih tipov

```
data T = ...
```

Definicija `type` uvaja okrajšavo, `data` pa nov podatkovni tip.

(Tipe lahko definiramo tudi z `newtype`, ki ga ta trenutek ne bomo obravnavali.)

Na primer, sezname lahko definiramo takole:

```
data List a =
    Nil
  | Cons (a, List a)
```

Tip `Maybe` je definiran z

```
data Maybe a =
    Nothing
  | Just a
```

To je pravzaprav tip `a option` iz OCaml.

13.1.3 Primer: AVL drevesa

Več podrobnosti bomo spoznali tako, da bomo iz OCaml prepisali implementacijo AVL dreves v Haskell:

```
-- AVL drevesa v Haskellu

module Avl where

  -- višino drevesa merimo s celim številom
  type Height = Integer

  -- koinduktivni podatkovni tip AVL dreves
  data AVLTree a =
      Empty
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
  | Node a Height (AVLTree a) (AVLTree a)
  deriving Show

-- primer drevesa
t :: AVLTree Integer
t = Node 5 3
    (Node 3 2
     (Node 1 1 Empty Empty)
     (Node 4 1 Empty Empty))
    (Node 8 1 Empty Empty)

height :: AVLTree a -> Height
height Empty = 0
height (Node _ h _ _) = h

leaf :: a -> AVLTree a
leaf v = Node v 1 Empty Empty

-- pametni konstruktor, ki poskrbi za višino
node :: a -> AVLTree a -> AVLTree a -> AVLTree a
node v l r = Node v (1 + max (height l) (height r)) l r

-- drevo t zapisano s pamentim konstruktorjem
t' :: AVLTree Integer
t' = node 5
      (node 3
       (node 1 Empty Empty)
       (node 4 Empty Empty))
      (node 8 Empty Empty)

-- drevo t zapisano še bolje
t'' :: AVLTree Integer
t'' = node 5
      (node 3
       (leaf 1)
       (leaf 4))
      (leaf 8)

-- seznam elementov v drevesu
toList :: AVLTree a -> [a]
toList Empty = []
toList (Node x _ l r) = toList l ++ (x : toList r)

search :: Ord a => a -> AVLTree a -> Bool
search x Empty = False
search x (Node y _ l r) =
    case compare x y of
      EQ -> True
      LT -> search x l
      GT -> search x r

test1 = search 1 t

test2 = search 42 t

rotateLeft :: AVLTree a -> AVLTree a
rotateLeft (Node x _ a (Node y _ b c)) = node y (node x a b) c
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
  rotateLeft t = t

  rotateRight :: AVLTree a -> AVLTree a
  rotateRight (Node y _ (Node x _ a b) c) = node x a (node y b c)
  rotateRight t = t

  imbalance :: AVLTree a -> Integer
  imbalance Empty = 0
  imbalance (Node _ _ l r) = height l - height r

  balance :: AVLTree a -> AVLTree a
  balance Empty = Empty
  balance (t@(Node x _ l r)) =
      case imbalance t of
        2 ->  case imbalance l of
                -1 -> rotateRight (node x (rotateLeft l) r)
                _ ->  rotateRight t
        -2 -> case imbalance r of
                1 -> rotateLeft (node x l (rotateRight r))
                _ -> rotateLeft t
        _ -> t

  add :: Ord a => a -> AVLTree a -> AVLTree a
  add x Empty = leaf x
  add x (t@(Node y _ l r)) =
      case compare x y of
        EQ -> t
        LT -> balance (node y (add x l) r)
        GT -> balance (node y l (add x r))

  remove :: Ord a => a -> AVLTree a -> AVLTree a
  remove x Empty = Empty
  remove x (Node y _ l r) =
      let removeSuccessor Empty = error "impossible"
          removeSuccessor (Node x _ Empty r) = (r, x)
          removeSuccessor (Node x _ l r) = (balance (node x l' r), y) where (l', y) =␣
↪removeSuccessor l
      in
        case compare x y of
          LT -> balance (node y (remove x l) r)
          GT -> balance (node y l (remove x r))
          EQ -> case (l, r) of
                  (_, Empty) -> l
                  (Empty, _) -> r
                  _ -> balance (node y' l r') where (r', y') = removeSuccessor r
```

13.2 Razredi tipov

Razred tipov (angl. *type class*) je način, kako v Haskellu opišemo skupino tipov, ki si delijo neko skupno funkcionalnost. Razred določa imena in tipe operacij (nekakšen vmesnik), tipi pa postanejo njegovi člani tako, da podajo **instanco**, v kateri te operacije implementirajo. Polimorfne funkcije lahko nato zahtevajo, da je njihov tipni parameter `a` v določenem razredu, in s tem dobijo na voljo pripadajoče operacije. V tem smislu so razredi tipov sorodni vmesnikom (angl. *interface*) v objektno usmerjenih jezikih, le da pripadnost razredu ni del same definicije tipa, temveč jo lahko dodamo naknadno z ločeno deklaracijo `instance`. Razredi tipov so tudi mehanizem, s katerim Haskell rešuje **ad-hoc polimorfizem**: ista operacija (npr. `==` ali `show`) se za različne tipe obnaša različno, prevajalnik pa glede na tip argumenta sam izbere ustrezno instanco.

Razrede tipov bomo razložili v živo na predavanjih s primerom `size.hs`, ki ilustrira osnovno idejo:

- Z deklaracijo `class Sized a where size :: a -> Int` uvedemo razred `Sized`, ki opisuje tiste tipe `a`, katerih vrednosti imajo neko »velikost«, izraženo s celim številom. Razred sam po sebi ne podaja implementacije; določa zgolj, da mora vsak član ponuditi funkcijo `size`.
- Funkcija `f :: Sized a => a -> Int` je polimorfna, a omejena: deluje za poljuben tip `a`, ki je v razredu `Sized`. Zapis `Sized a =>` pred tipom imenujemo **omejitev** (angl. *constraint*) in pove, da se sme znotraj `f` na vrednosti tipa `a` uporabiti operacija `size`.
- Z deklaracijami `instance` posameznim tipom dodelimo članstvo v razredu: za `Bool`, `Char` in `Int` enostavno povemo, koliko bitov zavzamejo. Pri tem ima vsaka instanca popoln dostop do strukture svojega tipa, zato lahko npr. instanca za `Bool` ujema vzorca `True` in `False`.
- Zapis `instance Sized a => Sized [a]` pove: če je `a` v razredu `Sized`, je tudi `[a]` v razredu `Sized`, velikost seznama pa je vsota velikosti njegovih elementov. Podobno za pare. Tako lahko iz nekaj osnovnih instanc samodejno izpeljemo velikost zelo bogatih tipov.

```
-- Ideja: podatki imajo neko velikost
-- Funkcionalnost: tipi, katerih vrednosti imajo "velikost"

-- Razred, ki opisuje tiste tipe a, ki so opremljeni s funkcijo "size", ki slika v Int
class Sized a where
    size :: a -> Int

-- Funkcija, ki vrne velikost svojega argumenta, povečana za 3
f :: Sized a => a -> Int
f x = size x * 3

-- Booli so veliki 1 bit
instance Sized Bool where
    size True = 1
    size False = 1

-- Chari imajo velikost 8 bitov
instance Sized Char where
    size _ = 8

-- Inti so veliki 64 bitov
instance Sized Int where
    size x = 64

-- Velikost seznamov
instance Sized a => Sized [a] where
    size [] = 0
    size (x : xs) = size x + size xs
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
-- Velikost parov
instance (Sized a, Sized b) => Sized (a,b) where
    size (x, y) = size x + size y

-- Primeri

-- V Haskellu so nizi seznami znakov, se pravi [Char], zato Haskell iz zgornjih
-- instanc izpelje velikost niza kot 8 * dolžina niza.
demo1 = size "This string contains forty-two characters."

-- Če napišemo samo 14, dobimo tip Num a => a, če pa napišemo 14 :: Int,
-- Haskell prislimo, da ima 14 tip Int. (Če bi napisali 14 :: Float, bi
-- ga prisilil, da bi imel tip Float).
demo2 = size [(14 :: Int), 15]

demo3 = size ("foo", (False, 42 :: Int))
```

13.2.1 Razredi tipov v standardni knjižnici

Ogledali si bomo nekatere najbolj uporabne razrede v standardni knjižnici.

`Eq`

Razred tipov, katerih vrednosti lahko primerjamo na enakost. Ponuja operatorja == in `/=` (neenakost). V razred sodijo vsi tipi, pri katerih ima primerjanje smisel: osnovni tipi, seznami in pari elementov iz `Eq`, uporabniško definirani podatkovni tipi (`data`), … Funkcije in akcije z računskimi učinki vanj ne sodijo.

`Ord`

Razred tipov z **linearno urejenostjo**. Razširja `Eq` in doda primerjalne operatorje <, <=, >, >= ter funkciji `min` in `max`. Tipično je dovolj implementirati eno od metod (`compare` ali <=), ostale Haskell izpelje samodejno. Za sestavljene tipe (npr. pare in sezname) je standardna instanca **leksikografska**.

`Show`

Razred tipov, katerih vrednosti znamo pretvoriti v niz znakov z metodo `show :: a -> String`. Uporablja se za izpis vrednosti v REPL-u in pri razhroščevanju. Sorodni razred `Read` opisuje obratno operacijo: razčlenjevanje niza v vrednost.

Numerični razredi

Številski tipi v Haskellu so organizirani v hierarhijo razredov, ki vsak doda nove operacije:

- `Num` — osnovne aritmetične operacije +, −, *, `negate`, `abs`, `fromInteger`.
- `Integral` — celoštevilske operacije, kot sta `div` in `mod` (npr. `Int`, `Integer`).
- `Fractional` — deljenje `/` in `recip` (npr. `Float`, `Double`, `Rational`).
- `Floating` — transcendentne funkcije, kot so `sqrt`, `exp`, `log`, `sin`, `cos`.

Zaradi te hierarhije ima številska konstanta `14` polimorfni tip `Num a => a` in se specializira glede na kontekst.

Monoid

Razred tipov, ki tvorijo **monoid**: imajo asociativno binarno operacijo `mappend` (pogosto pisano kot `<>` iz razreda `Semigroup`) in nevtralni element `mempty`. Klasični primeri so seznami (stikanje, `[]`), nizi, števila pod seštevanjem ali množenjem itn. Razred omogoča enotno pisanje funkcij, ki zlagajo vrednosti (npr. `mconcat`).

Functor

Razred **enoparametričnih konstruktorjev tipov** (oblike `f a`), ki znajo preslikati funkcijo nad svojo vsebino. Edina metoda je `fmap :: (a -> b) -> f a -> f b`, ki posploši `map` s seznamov na poljuben funktor. V razred sodijo `[]`, `Maybe`, `Either e`, `IO`, drevesa, … Instanca mora spoštovati **zakona funktorja**: ohranjanje identitete in kompozicije.

Applicative

Razred, ki razširja `Functor` in omogoča delo s funkcijami več argumentov v kontekstu `f`. Doda `pure :: a -> f a` (vstavi vrednost v kontekst) in `(<*>) :: f (a -> b) -> f a -> f b` (uporabi funkcijo v kontekstu na argumentu v kontekstu). `Applicative` je ključen vmesni korak med funktorji in **monadami**, ki jih bomo spoznali naslednjič.

POGLAVJE 14

Monade in računski učinki

Monade so matematični pojem, ki posplošuje algebrajske strukture, kot so grupa, kolobar in vektorski prostor. Hkrati pa se monade uporablja tudi v teoretičnem računalništvu in deklarativnem programiranju za opisovanje *računskih učinkov*. V tej lekciji bomo spoznali osnovno uporabo monad v programiranju.

Haskell je čist programski jezik in vse računske učinke predstavi z monadami. Programer lahko v Haskellu definira nove monade in z njimi simulira razne računske učinke. Tudi drugi programski jeziki imajo monade, le da niso neposredno dostopne programerju. Jeziki kot so OCaml, Java, C++ uporabljajo le eno monado, ki predstavlja vse računske učinke, podprte v jeziku (I/O, izjeme, stanje).

14.1 Računski učinki

Računski učinki so širok pojem, ki ga je težko povsem opredeliti. Lahko rečemo, da je računski učinek vsak pojav v programu, ki ni le preprosto procesiranje informacije. Med računske učinke štejemo:

- **manjkajoča vrednost**: program lahko ugotovi, da rezultat ne more izračunati, na primer, ker ena od vmesnih operacij ni definirana
- **izjeme**: program lahko nenadoma prekine izvajanje in javi napako ali vrže izjemo (`throw`, `catch`)
- **vhod/izhod**: program lahko komunicira z zunanjim svetom preko vhodnega in izhodnega kanala (`input`, `print`)
- **stanje**: program ima trenutno stanje, ki se spreminja, ko program teče (`set`, `get`)
- **nedeterminizem**: izvajanje porgrama lahko pripelje do več rezultatov
- **verjetnostno računanje**: program lahko izbira med več možnostmi z verjetnostno porazdelitvijo
- **timeout**: izvajanje se lahko prekine, če traja predolgo

Poznamo pa še veliko drugih računskih učinkov.

14.2 Monade

Zgoraj našteti računski učinki so zelo raznoliki, a kljub temu imajo skupno strukturo, ki jo zajema pojem monade. Razložimo ga iz stališča programerja, ki želi *simulirati* računske učinke v »na roko«, se pravi v čistem jeziku.

Najprej ločimo med:

- **(čista) vrednost** (pure value): že izračunan podatek, ki je »inerten«, ga ne izvajamo.
- **izračunom** (computation): koda, ki jo moramo izvesti, da dobimo rezultat, med izvajanjem pa se lahko pojavljajo računski učinki.

Primer

V programskem jeziku OCaml, je

```
print_endling "hello" ; let x = 14 in (3 * x, ["cow"; "rabbit"])
```

izračun. Ko ga poženemo, sproži učinek (izpis na zaslon) in vrne vrednost `(42, ["cow"; "rabbit"])`.

Ker bomo izračune *simulirali*, jih bomo predstavili z ustreznimi podatkovnimi tipi, ki izražajo računske učinke. Vsak računski učinek ali kombinacija učinkov ima svoj podatkovni tip.

Primer

Izračun, v katerem se lahko zgodi učinek »manjkajoča vrednost«, predstavimo s tipom `Maybe a`:

- če izračun uspešno izračuna končno vrednost `v`, to predstavimo z `Just v`
- če naleti na manjkajočo vrednost, to predstavimo z `Nothing`

Primer

Izračun, ki lahko izpisuje na standardni izhod, predstavimo s tipom `(String, a)`:

- izračun, ki na zaslon izpiše `Kdor to bere, je osel` in vrne vrednost `42`, predstavimo z urejenim parom `("Kdor to bere, je osel", 42)`.

Primer

Izračun, ki prebere niz s standardnega vhoda in vrne rezultat, predstavimo s tipom `String -> a`, ker je rezultat lahko odvisen od vhodnega podatka, ki ga je dobil izračun.

Primer

Izračun, ki ima stanje tipa `s` in rezultat tipa `a`, prestavimo s tipom `s -> (s, a)`, kar preberemo takole: izračun sprejme stanje in vrne spremenjeno stanje in rezultat.

Ni nujno, da vsak izračun povzroči računske učinke. Taki, ki samo vrnejo vrednost, se imenujejo **čisti izračuni**. Poleg tega lahko izračune **komponiramo**, podobno kot funkcije, da se zgodijo eden za drugim. Struktura, ki opisuje čiste izračune in komponiranje, je **monada**.

14.3 Monada

V funkcijskem programiranju monada predstavlja enega ali kombinacijo večih učinkov. Njene sestavine so:

1. Preslikava `m : Type → Type`, ki tip `a` preslika v tip `m a`. Lahko si mislimo, da so elementi `m a` izračuni, ki izračunajo vrednost tipa `a` in sprožajo učinke, ki jih zajema `m`.
2. Preslikava `return : a → m a`, ki vrednost tipa `a` predstavi kot čisti izračun.
3. Operacija `>>= : m a → (a → m b) → m b`, s katero kombiniramo izračune. Če je `c` izračun, ki izračuna vrednost tipa `a` in je `f : a → m b` funkcija, ki vrednosti tipa `a` preslika v izračun tipa `m b`, potem je `c >>= f` izračun, ki ju združi v izračun tipa `m b`.

Primer

Računski učinek **manjkajoča vrednost** zajame monada `Maybe`:

1. `m a = Maybe a`. Izračun je bodisi `Just v` za neko vrednost `v` tipa `a`, bodisi `Nothing`, kadar vrednost manjka.
2. `return v = Just v`, saj čisti izračun uspešno vrne vrednost `v` brez učinka manjkanja.
3. `c >>= f` definiramo po primerih:
   - če je `c = Nothing`, je `c >>= f = Nothing`,
   - če je `c = Just v`, je `c >>= f = f v`.

Tretja točka zajame bistvo: če vmesni izračun `c` vrne `Nothing`, prekinemo računanje in vrnemo `Nothing`; če pa vrne vrednost `v`, jo predamo nadaljnjemu izračunu `f`. Tako se manjkajoča vrednost samodejno razširi skozi celotno zaporedje izračunov, ne da bi jo bilo treba ročno preverjati na vsakem koraku.

Primer

Računski učinek **nedeterminizem** dobimo, kadar lahko izračun vrne več rezultatov, oziroma med računanjem izbira med več možnostmi. Ustrezna monada je:

1. `m a = [a]`. Izračun vrednosti tipa `a` predstavimo s seznamom vseh možnih rezultatov.
2. `return v = [v]`, saj je čisti izračun tak, ki nedvoumno vrne natanko eno vrednost `v`.
3. `c >>= f = concat (map f c)`. Izračun `c` ima več možnih rezultatov; za vsakega izmed njih z `f` poženemo nadaljnji izračun, ki ima spet svoj seznam možnosti, in vse skupaj zlepimo v en seznam.

Pojasnimo tretjo točko na primeru. Naj bo `c = [1, 2, 3]` in `f n = [n, -n]`. Potem je `map f c = [[1, -1], [2, -2], [3, -3]]`, po `concat` pa dobimo

```
c >>= f  =  [1, -1, 2, -2, 3, -3]
```

Vsako izmed treh možnosti v `c` smo razvejali na dve, skupaj torej šest končnih rezultatov.

14.3.1 Enačbe za monado

Da definicija monade res zajame računski učinek, mora zadoščati naslednjim trem enačbam:

1. **Leva enota:** `return x >>= f = f x`
2. **Desna enota:** `c >>= return = c`
3. **Asociativnost:** `(c >>= f) >>= g = c >>= (\x -> f x >>= g)`

Pomen enačb je naslednji:

- **Leva enota** pove, da čisti izračun `return x` ne sproži nobenega učinka: njegovo zaporedje s funkcijo `f` je enakovredno temu, da preprosto poženemo `f x`.
- **Desna enota** pove, da je dodajanje `return` na konec izračuna brez učinka, saj `return` le zapakira že izračunano vrednost nazaj v izračun.
- **Asociativnost** pove, da pri zaporedju treh izračunov vrstni red oklepanja ne vpliva na rezultat; učinki se v obeh primerih zgodijo v istem vrstnem redu.

Skupaj povedo, da se monada obnaša kot zaporedje izračunov, v katerem so vrinjeni `return` nepomembni, oklepanje pa ne vpliva na končni rezultat.

Primer (Enačbe za monado `Maybe`)

Preverimo enačbe neposredno iz definicije `>>=` za `Maybe`.

- **Leva enota:** `return x >>= f = Just x >>= f = f x` po drugi alineji definicije `>>=`.
- **Desna enota:** ločimo dva primera glede na `c`:
  - če je `c = Nothing`, je `Nothing >>= return = Nothing = c`;
  - če je `c = Just v`, je `Just v >>= return = return v = Just v = c`.
- **Asociativnost:** spet ločimo primera:
  - če je `c = Nothing`, sta obe strani enaki `Nothing`;
  - če je `c = Just v`, je leva stran `(Just v >>= f) >>= g = f v >>= g`, desna pa `Just v >>= (\x -> f x >>= g) = (\x -> f x >>= g) v = f v >>= g`. Strani sta enaki.

Primer (Enačbe za monado seznamov)

Preverimo enačbe še za seznamsko monado, kjer je `return v = [v]` in `c >>= f = concat (map f c)`.

- **Leva enota:** `return x >>= f = [x] >>= f = concat (map f [x]) = concat [f x] = f x`.
- **Desna enota:** če je `c = [v1, ..., vn]`, je

```
c >>= return  =  concat (map return c)
              =  concat [[v1], ..., [vn]]
              =  [v1, ..., vn]  =  c
```

- **Asociativnost:** na obeh straneh dobimo seznam, ki ga sestavimo tako, da za vsako možnost `v` iz `c` najprej poženemo `f v`, dobimo seznam `[w1, ..., wk]`, in nato za vsak tak `w` poženemo `g w`. Vrstni red prehajanja je enak ne glede na oklepanje, zato sta tudi končna seznama enaka.

Več o monadah lahko preberete na strani Monad v Haskell wiki.

14.4 Monade in računski učinki v Haskellu

Računski učinki so v Haskellu predstavjeni s tremi razredi tipov, ki so urejeni hierarhično: vsak naslednji razred razširi prejšnjega z dodatnimi operacijami.

14.4.1 `Functor`

Razred `Functor` opisuje konstruktorje tipov `t`, na katerih znamo preslikati funkcijo čez »vsebovane« vrednosti, ne da bi spremenili obliko strukture. Glavna metoda je

```
fmap :: Functor t => (a -> b) -> t a -> t b
```

ki funkcijo `f : a -> b` predela v funkcijo `fmap f : t a -> t b`. Če si mislimo, da `t` predstavlja računski učinek, potem je `c :: t a` izračun podatka tipa `a`. Funkcija `fmap f c`, izračun `c` predela tako, da rezultat (ali rezultate) `c` preslika z `f`.

Namesto `fmap f v` lahko pišemo `f <$> v`.

Primer

Naj bo `readMaybe :: String -> Maybe Int` funkcija, ki vrne `Just n`, kadar niz predstavlja število, sicer pa `Nothing`. Če želimo prebrano število podvojiti, uporabimo `(*2)` s pomočjo `fmap`:

```
fmap (*2) (readMaybe "21")    -- Just 42
fmap (*2) (readMaybe "abc")   -- Nothing
```

Operacija `fmap` torej ohrani strukturo: vrednost `Nothing` ostane `Nothing`, saj je manjkajoča vrednost računski učinek, ki ga le ponovimo. Na seznamu, ki je prav tako funktor, `fmap` deluje hkrati na vseh elementih:

```
fmap (*2) [1, 2, 3]   -- [2, 4, 6]
```

14.4.2 `Applicative`

Razred `Applicative` razširi razred `Functor`, se pravi vsak `Applicative` je hkrati `Functor`. Doda dve metodi:

```
pure  :: Applicative t => a -> t a
(<*>) :: Applicative t => t (a -> b) -> t a -> t b
```

- `pure` čisto vrednost predstavi kot izračun brez učinkov in igra vlogo operacije `return` iz definicije monade.
- `<*>` uporabi funkcijo, ki je *sama rezultat izračuna z učinki*, na argument, ki je prav tako rezultat izračuna z učinki, ter pri tem združi učinke obeh izračunov.

Razlika v izrazni moči je naslednja: razred `Functor` omogoča uporabo *čiste* funkcije enega argumenta, razred `Applicative` pa kombiniranje *več neodvisnih* izračunov s funkcijo več argumentov.

Primer

Z operatorjem `<*>` razširimo seštevanje na monado `Maybe`:

```
(+) <$> Just 3 <*> Just 4     -- Just 7
(+) <$> Just 3 <*> Nothing    -- Nothing
```

Izraz najprej z `<$>` (oziroma `fmap`) uporabi operacijo `(+)` na `Just 3` in dobi `Just (3+)`, torej funkcijo znotraj učinka. Operator `<*>` to funkcijo nato uporabi še na `Just 4`. Kadar manjka katerikoli izmed argumentov, je rezultat `Nothing`.

Na seznamih razred `Applicative` opisuje nedeterminizem in vrne vse možnosti:

```
(+) <$> [1, 2] <*> [10, 20]   -- [11, 21, 12, 22]
```

14.4.3 Monad

Razred `Monad` razširi razred `Applicative`, tako da je vsak `Monad` hkrati `Applicative` in s tem `Functor`. Bistveni metodi sta:

```
return :: Monad m => a -> m a
(>>=)  :: Monad m => m a -> (a -> m b) -> m b
```

- `return` je pravzaprav `pure`.
- Operator `>>=` omogoča, da naslednji izračun *izberemo glede na rezultat prejšnjega*. Pri operatorju `<*>` so vsi izračuni in njihovi učinki znani vnaprej, pri `>>=` pa je naslednji izračun funkcija prejšnjega rezultata, zato so lahko učinki odvisni od izračunanih vrednosti.

14.4.4 Hierarhija razredov

Ti trije razredi tvorijo verigo `Functor ⇒ Applicative ⇒ Monad`, pri čemer vsaka stopnja doda novo izrazno moč:

- `Functor` omogoča preslikavo čiste funkcije *enega* argumenta skozi učinkovit izračun.
- `Applicative` doda kombiniranje *več neodvisnih* učinkovitih izračunov v enoten izračun.
- `Monad` doda *medsebojno odvisnost*: naslednji izračun je odvisen od rezultata prejšnjega.

Hierarhija ima tudi praktične posledice: če za neki tip definiramo instanco razreda `Monad`, mora Haskell zanj poznati tudi instanci razredov `Functor` in `Applicative`. V vsakdanji rabi operatorja `<$>` (iz razreda `Functor`) in `<*>` (iz razreda `Applicative`) pogosto nastopata znotraj definicije monade, saj naredita zapis krajši in preglednejši, kadar učinki niso odvisni eden od drugega.

14.5 Zapis do

Pri daljših zaporedjih izračunov z operatorjem `>>=` koda hitro postane težko berljiva, saj se gnezdijo lambda izrazi. Haskell zato ponuja **zapis do**, ki je okrajšava za zaporedje klicev operatorja `>>=` in deluje v poljubni monadi.

Splošno pravilo prevajanja je: zapis

```
do x <- c1
   y <- c2
   c3
```

prevedemo v

```
c1 >>= \x ->
c2 >>= \y ->
c3
```

Vrstice oblike `x <- c` pomenijo »poženi izračun `c` in njegov rezultat poimenuj `x`«; zadnja vrstica je izračun, katerega rezultat je rezultat celotnega `do` bloka.

Primer

Zapišimo izračun, ki nedeterministično izbere število `x` iz seznama `[1, 2, 3]` in število `y` iz seznama `[10, 20, 30]` ter vrne njuno vsoto. Z zapisom `do` je tak izračun zelo berljiv:

```
vsote :: [Int]
vsote = do
    x <- [1, 2, 3]
    y <- [10, 20, 30]
    return (x + y)
```

Isti izračun z operatorjem >>= zapišemo takole:

```
vsote :: [Int]
vsote = [1, 2, 3] >>= \x ->
        [10, 20, 30] >>= \y ->
        return (x + y)
```

V obeh primerih dobimo seznam vseh devetih vsot:

```
[11, 21, 31, 12, 22, 32, 13, 23, 33]
```

Zapis `do` torej ohrani vrstico za vsak izračun in spominja na zaporedje stavkov v imperativnem jeziku, čeprav se v ozadju ves čas izvaja monadno komponiranje z operatorjem >>=.

14.6 Verjetnostno računanje

Doslej obravnavane monade so bile že vgrajene v Haskellu (`Maybe` in seznami). Pokažimo še, kako sami definiramo monado za **verjetnostno računanje**.

14.6.1 Matematična definicija

Končna verjetnostna porazdelitev nad množico X je funkcija

$$\mu : X \to [0, 1]$$

s končnim nosilcem (le na končno mnogo $x \in X$ je $\mu(x) > 0$), za katero velja $\sum_{x \in X} \mu(x) \leq 1$. Strogo gledano bi morala biti vsota enaka 1, a dovolimo tudi **pod-porazdelitve**, kjer manjkajoča verjetnost ustreza zavrženim izidom. Naj $D(X)$ označuje množico vseh takih porazdelitev nad X.

Pripadajoča **monada porazdelitev** D ima sestavine:

- Vsakemu tipu X priredimo tip porazdelitev $D(X)$.
- Čisti izračun je *gotov dogodek*: return$(x) = \delta_x$, kjer je $\delta_x(x) = 1$ in $\delta_x(x') = 0$ za $x' \neq x$.
- Komponiranje. Za porazdelitev $\mu \in D(X)$ in funkcijo $f : X \to D(Y)$ je μ >>= f porazdelitev v $D(Y)$, ki je dana s formulo

$$(\mu \text{ >>= } f)(y) \;=\; \sum_{x \in X} \mu(x) \cdot f(x)(y),$$

  ki pravi: verjetnost izida y v sestavljenem izračunu dobimo tako, da po *vseh* vmesnih izidih x seštejemo prispevke $\mu(x) \cdot f(x)(y)$, torej verjetnost, da prvi izračun da x, pomnoženo z verjetnostjo, da iz x pridemo do y.

Vsota $\sum_{x \in X}$ v formuli **združi** vse x, ki vodijo do istega y, v skupno verjetnost izida y.

14.6.2 Implementacija v Haskellu

V Haskellu porazdelitev predstavimo s seznamom parov $[(x_1, p_1), \ldots, (x_n, p_n)]$, kar je v datoteki `verjetnost.hs` tip `Verjetnost a`. Definicija `return` ne povzroča težav, pri definiciji >>= pa naletimo na težavo: združevanje `x`-ov v vsoti zahteva, da lahko elemente `X` primerjamo (Haskellov razred `Eq`) – razreda `Monad` in `Applicative` v Haskellu pa ne dovolita dodatnih omejitev (delovati mora za poljuben tip, tudi tak, ki nima instance `Eq`). Združevanja zato ne moremo vključiti v sam >>=, temveč ga izvedemo naknadno s pomožno funkcijo `zdruzi :: Eq a => Verjetnost a -> Verjetnost a`, ki združi verjetnosti enakih izidov.

Datoteka prikaže to monado na dveh primerih:

- **vsota metov dveh kock**: porazdelitev vsote dveh poštenih šestostranih kock;
- **vsota metov dveh kock s pogojem**: enako, le da met zavržemo, kadar je vsota soda. Za zavračanje uporabimo pomožno funkcijo `pogoj :: Bool -> Verjetnost ()`, ki ob neresničnem pogoju vrne prazno porazdelitev in s tem izloči to vejo izračuna. Dobimo pod-porazdelitev, katere vsota verjetnosti je $\frac{1}{2}$.

```
-- Monada za verjetnostno računanje.
--
-- Izračun predstavlja končno verjetnostno porazdelitev nad poljubnim tipom.
-- Porazdelitev elementov tipa a predstavimo s seznamom parov (izid, verjetnost):
-- seznam [(x_1, p_1), ..., (x_n, p_n)] pomeni, da izid x_i nastopi z
-- verjetnostjo p_i. Vsota verjetnosti je največ 1; če je strogo manjša,
-- manjkajoča verjetnost ustreza neuspelim ali zavrženim izračunom.

module Verjetnost where

-- Iz standardne knjižnice uvozimo nekaj funkcij za delo s seznami,
-- ki jih bomo potrebovali v funkciji `zdruzi`.
import Data.List (sortBy, groupBy)
import Data.Ord (comparing)
import Data.Function (on)

-- Definicija tipa za verjetnostno porazdelitev.
--
-- Ključna beseda `newtype` v Haskellu definira nov tip, ki ima natanko
-- en konstruktor in eno polje, in pri tem ne uvede dodatne ovojnice v
-- pomnilniku. Tip `Verjetnost a` je v bistvu `[(a, Float)]`,
-- vendar ga Haskell obravnava kot ločen tip. Tako ga lahko opremimo z
-- lastnimi instancami razredov tipov (Functor, Applicative, Monad).
--
-- Zapis `{ izidi :: [(a, Float)] }` poleg konstruktorja `Verjetnost`
-- definira še konstruktor
--     izidi :: Verjetnost a -> [(a, Float)]
-- s katerim iz vrednosti tipa `Verjetnost a` dobimo nazaj seznam parov.
newtype Verjetnost a = Verjetnost { izidi :: [(a, Float)] }

-- Instanca razreda `Show` določa, kako naj se vrednost tipa
-- `Verjetnost a` izpiše v GHCi. Vsak izid postavimo v svojo vrstico,
-- ločeno s tabulatorjem od njegove verjetnosti.
instance Show a => Show (Verjetnost a) where
    show (Verjetnost xs) =
        concat [show x ++ ":\t" ++ show p ++ "\n" | (x, p) <- xs]

-- Da bi tip `Verjetnost` postal Haskellova monada, moramo podati
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
-- tri instance: `Functor`, `Applicative` in `Monad`. Razred `Monad`
-- namreč deduje `Applicative`, ta pa `Functor`.

-- Functor: na vsakem izidu uporabimo čisto funkcijo `f`, verjetnost pa
-- ostane nespremenjena. Tako npr. `fmap (+10) kocka` pretvori izide
-- 1..6 v 11..16, vsak še vedno z verjetnostjo 1/6.
instance Functor Verjetnost where
    fmap f (Verjetnost xs) = Verjetnost [(f x, p) | (x, p) <- xs]

-- Applicative:
--   * `pure x` je porazdelitev, ki z verjetnostjo 1 vrne izid `x`.
--     To je čisti izračun, ki vedno vrne isto vrednost.
--   * `<*>` izvede dva neodvisna izračuna in zmnoži njuni verjetnosti,
--     saj je verjetnost neodvisnih dogodkov produkt njihovih verjetnosti.
--     Pozor: isti dogodek se bo pojavil večkrat, pripadajoče verjetnosti
--     bi morali združiti. To počne funkcija zdruzi, definirana spodaj.
instance Applicative Verjetnost where
    pure x = Verjetnost [(x, 1)]
    (Verjetnost fs) <*> (Verjetnost xs) =
        Verjetnost [(f x, p * q) | (f, p) <- fs, (x, q) <- xs]

-- Monad: ključna operacija `>>=` (beri "veži") sestavi dva izračuna
-- v zaporedje. Za vsak izid `x` z verjetnostjo `p` v prvem izračunu
-- poženemo nadaljnji izračun `f x`, ki vrne svojo porazdelitev. Vsako
-- verjetnost v tej drugi porazdelitvi pomnožimo s `p`. Tako dobimo
-- skupno verjetnost zaporedja dveh dogodkov. Le-ti spet niso združeni.
instance Monad Verjetnost where
    return = pure
    (Verjetnost xs) >>= f =
        Verjetnost [(y, p * q) | (x, p) <- xs, (y, q) <- izidi (f x)]

-- Funkcija `zdruzi`: sešteje verjetnosti enakih izidov in
-- vrne urejen seznam različnih izidov.
--
-- Matematično gledano bi to združevanje sodilo neposredno v operacije
-- `>>=` in `<*>`, saj je seštevanje verjetnosti enakih izidov del definicije.
-- V Haskellu tega ne moremo narediti neposredi, ker razreda `Monad` in `Applicative`
-- ne dovolita dodatnih omejitev: `>>=` mora delovati za poljuben tip izidov `a`,
-- ne le za tiste z `Eq a` ali `Ord a`. Brez primerjanja pa ne moremo prepoznati
↪enakih izidov.
-- Združevanje zato naknadno opravimo z `zdruzi`, ki potrebuje omejitev `Ord a`.
zdruzi :: Ord a => Verjetnost a -> Verjetnost a
zdruzi (Verjetnost xs) =
    Verjetnost [ (k, sum (map snd gs))
               | gs@((k, _) : _) <- groupBy ((==) `on` fst) (sortBy (comparing fst)
↪xs) ]

-- Pomožna funkcija `preveri`: zavrže veje izračuna, ki ne ustrezajo pogoju.
--   * Če pogoj velja, vrnemo gotovi izid `()`;
--     `>>=` nato z verjetnostjo 1 nadaljuje s preostankom izračuna.
--   * Če pogoj ne velja, vrnemo prazno porazdelitev in s tem ta veja izračuna
↪prekine.
-- Uporabljamo jo v `do` blokih za filtriranje izidov.
pogoj :: Bool -> Verjetnost ()
pogoj True  = Verjetnost [((), 1)]
pogoj False = Verjetnost []
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
-- Met poštene šestostrane kocke: vsak izmed izidov
-- 1, 2, ..., 6 nastopi z verjetnostjo 1/6.
kocka :: Verjetnost Int
kocka = Verjetnost [(n, 1/6) | n <- [1..6]]

-- Primer 1: porazdelitev vsote pri metu dveh kock.
--
-- Vsote se ponovijo: 7 dobimo kot 1+6, 2+5, 3+4, 4+3, 5+2, 6+1,
-- torej z verjetnostjo 6/36 = 1/6. Brez `zdruzi` bi izpis imel
-- 36 vrstic, s klicem `zdruzi` pa 11 vrstic za različne vsote 2..12.
vsotaDveh :: Verjetnost Int
vsotaDveh = zdruzi $ do
    x <- kocka
    y <- kocka
    return (x + y)

-- Primer 2: porazdelitev vsote dveh metov, pri čemer met zavržemo,
-- če je vsota soda.
--
-- Polovica vseh 36 parov ima sodo vsoto, zato vsota verjetnosti
-- preostalih izidov ni 1, temveč 1/2. Dobimo torej "pod-porazdelitev",
-- v kateri manjkajoča verjetnost ustreza zavrženim metom.
vsotaDvehLihih :: Verjetnost Int
vsotaDvehLihih = zdruzi $ do
    x <- kocka
    y <- kocka
    preveri (odd (x + y))
    return v
```

14.7 Dodatni primeri monad

14.7.1 Izjeme

Izjeme naredimo še enkrat od začetka, brez uporabe `Maybe`.

```
-- Primer računskega učinka "napaka"
-- V Haskellu je to že definirano kot Maybe

-- izračun bodisi javi napako bodisi vrne rezultat
data Computation a = Error | Result a
                   deriving Show

instance Functor Computation where
    fmap f Error = Error
    fmap f (Result v) = Result (f v)

instance Applicative Computation where
    pure = Result

    Error <*> _ = Error
    _ <*> Error = Error
    (Result f) <*> (Result v) = Result (f v)
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
instance Monad Computation where
    -- return = pure

    Error >>= f         =  Error
    (Result v) >>= f    =  f v

-- catch C H
-- C = izračun
-- H = če C javi napako, jo ujamemo in nadomestimo s H
--
-- try:
--   C
-- except Error:
--   H
catch :: Computation a -> Computation a -> Computation a
catch (Result v) _ = Result v
catch Error h      = h

-- trije načini, da se naredi isto

demo1 :: Computation Integer
demo1 = (Result 5) >>= (\v -> Result (v + 10))

demo2 :: Computation Integer
demo2 = return 5 >>= (\v -> return (v + 10))

demo3 :: Computation Integer
demo3 = do v <- return 5
           return (v + 10)

-- uporabimo error
demo4 :: Computation Integer
demo4 = do x <- return 5
           y <- return (x + 1)
           z <- if y < 7 then return 7 else Error
           w <- return (z + 3)
           return (x + y + z + w)
```

14.7.2 Stanje

Izračun, ki uporablja stanje tipa `s`, predstavimo s funkcijo `s -> (s, a)`, ki sprejme začetno stanje in vrne (novo stanje, rezultat). Z operacijama `get` in `set` lahko stanje beremo in posodabljamo, kakor da bi pisali v ukaznem jeziku.

```
-- State s a je tip izračunov, ki imajo stanje tipa s in rezultat tipa a
-- naivna definicija:
--   type State s a = (s -> (s, a))

newtype State s a = State { run ::  s -> (s, a) }
-- State c .... izračun c, ki uporablja state,
--              c je funkcija, ki sprejme začetno stanje
--              in vrne (končno stanje, rezultat)
--
-- run c s0 ... poženi izračun c v začetnem stanju s0

instance Functor (State s) where
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
    fmap f c = State (\s -> let (s', x) = run c s in (s', f x))

instance Applicative (State a) where
    pure x = State (\s -> (s, x))

    cf <*> cx = State (\s0 -> let (s1, f)  = run cf s0
                                  (s2, x) = run cx s1
                              in (s2, f x))

instance Monad (State a) where
    -- return x = State (\s -> (s, x))

    c >>= f = State (\s0 -> let (s1, x) = run c s0
                            in run (f x) s1)

-- operacije na stanju

-- vrni trenutno stanje
get :: State s s
get = State (\s0 -> (s0, s0))

-- nastavi trenutno stanje
set :: s -> State s ()
set s = State (\_ -> (s, ()))

demo :: State Int String
demo = do x <- get
          set (x + 30)
          y <- get
          set (x + y)
          z <- get
          return (if z < 100 then "foo" else "bar")

-- run demo 15
```

14.7.3 Problem n dam

Datoteka `queens.hs` reši klasičen **problem n dam** — postavi n šahovskih dam na šahovnico $n \times n$, tako da se ne napadajo — s pomočjo seznamske monade za nedeterminizem. Funkcija `postavi` z `do` zapisom za vsako vrstico izbere stolpec, s `preveri` pa zavrže izbire, ki vodijo do napada. Rezultat je seznam vseh veljavnih postavitev.

```
-- Problem šahovskih dan, rešen z monado za nedeterminizem

-- Polja na šahovnici predstavim s koordinatami
type Polje = (Int, Int)

-- Dama je predstavljena kar s poljem, na katerem stoji
type Dama = Polje

-- ali se dve polji napadata?
napad :: Polje -> Polje -> Bool
napad (x1,y1) (x2,y2) =
    (x1 == x2) || (y1 == y2) || abs (x1 - x2) == abs (y1 - y2)

-- napadeno p ds -- ali dame ds napadajo polje p?
```

(continues on next page)

(nadaljevanje iz prejšnje strani)

```
napadeno :: Polje -> [Dama] -> Bool
napadeno p = any (napad p)

-- preveri b prekine izvajanje (vrne nič možnosti) ali nadaljuje z eno samo možnostjo,
-- glede na to, ali pogoj b velja
preveri :: Bool -> [()]
preveri True = [()]
preveri False = []

-- Postavi n dam na šahovnico n x n, tako da se ne napadajo
dame :: Int -> [[Dama]]
dame n = postavi [] 1
  where postavi :: [Dama] -> Int -> [[Dama]]
        postavi dame k | k > n     = return dame
                       | otherwise = do j <- [1..n]
                                        preveri $ not (napadeno (k,j) dame)
                                        postavi ((k,j) : dame) (k+1)
```