

Principi programskih jezikov

Programski jeziki :

- kako so sestavljeni
- programske koncepte
 - (objekti, zanke, lokalna spremenljivka, polimorfizem, kontinuacija, algebraični tip,)

Anatomija programskega jezika :

- sintaksa : pravila, kako pišemo veljavno kodo
- staticna semantika (pomen) :
prevajalnik preveri type : objekt + ? \rightsquigarrow napaka
- dinamična semantika : kako se program izvaja
- analiza :
 - dokazujemo pravilnost
 - optimiziramo
 - analiziramo računsko tačnost

Aritmetični izrati

cela števila, spremenljivke (read-only), + in *

$$3 * (x + 7)$$

Konkretna sintaksa

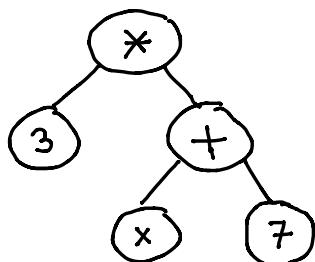
Izvorna koda (aritmetični izrat) kot niz znakov (string):

"3 * (x + 7)"

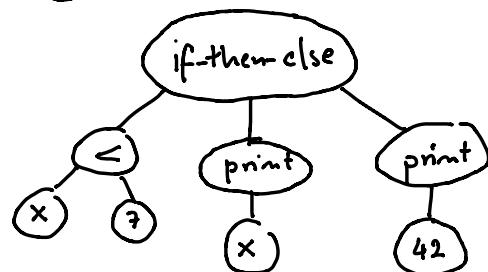
" 3 * (x + 7) "

Abstraktna sintaksa

Sintaktično drevo



```
if (x < 7) {  
    print(x);  
}  
else {  
    print(42);  
}
```





Gramatika (slownica pravila):

\rightarrow end of file (end of input)

$\langle \text{izraz} \rangle ::= \langle \text{aditivni-izraz} \rangle \text{ EOF}$ \rightarrow ali

$\langle \text{aditivni-izraz} \rangle ::= \langle \text{multiplikativni-izraz} \rangle \mid$
 $\langle \text{aditivni-izraz} \rangle + \langle \text{multiplikativni-izraz} \rangle$

$\langle \text{multiplikativni-izraz} \rangle ::= \langle \text{osnovni-izraz} \rangle \mid$
 $\langle \text{multiplikativni-izraz} \rangle * \langle \text{osnovni-izraz} \rangle$

$\langle \text{osnovni izraz} \rangle ::= \langle \text{spremenljivka} \rangle \mid \langle \text{stevinka} \rangle \mid$
 $(\langle \text{aditivni-izraz} \rangle)$

$\langle \text{spremenljivka} \rangle ::= [a-z]^+$
 \downarrow regularni izraz

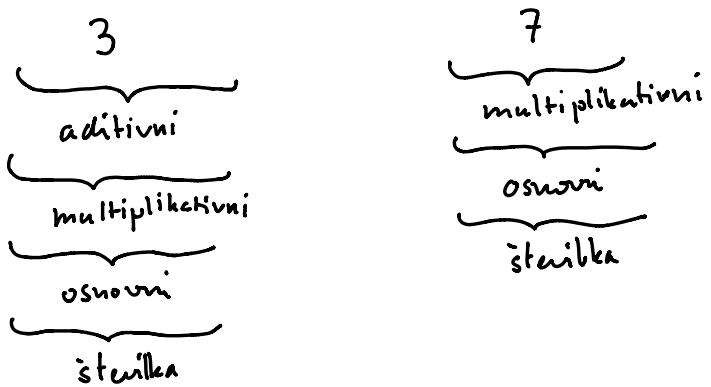
$\langle \text{stevinka} \rangle ::= [0-9]^+$

3 + 7 EOF

 |
 izraz

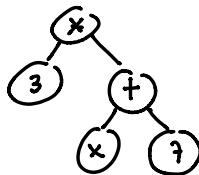
3 + 7

 aditivni-izraz



| z konkretno v abstraktno sintaksu
(razčlenjevanje ali "parsing")

" $3 * (x + 7)$ "



1. Leksična analiza:
niz razčlenimo na osnovne gradnike (tokens)

" $3 __ * (x __ + 7)$ "

ŠTEVILKA(3), KRAT, OKLEPAJ, SPREMENLJIVKA(x), PLUS,
ŠTEVILKA(7), ZAKLEPAJ, EOF

2. Razčlenjevanje:

niz gradnikov predela v drevo

(lahko javi sintaktično napako, npr. "manjka zaklepaj")

$$3 * (x + 7)$$

$x + 2 * y + \text{tempvmariboru}$

Okolje (runtime environment) :

- preslika imena spremenljivk v njihove vrednosti

- primer:

$$\eta := [x \mapsto 7, y \mapsto 12, z \mapsto 4] \quad \begin{matrix} \eta \\ \alpha \beta \gamma \delta \varepsilon \xi \theta \pi \lambda \end{matrix} \quad \text{eta}$$

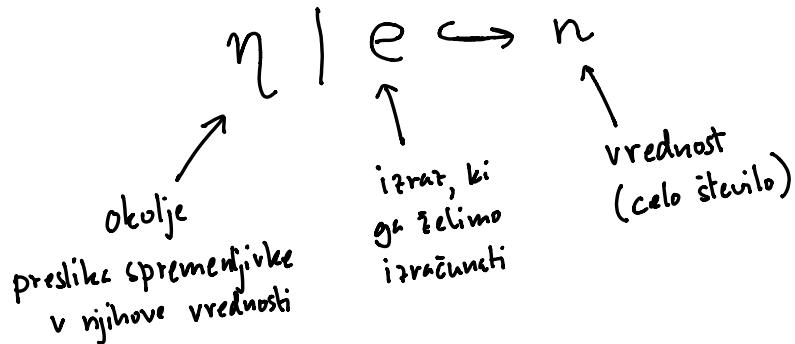
V okolju $[x \mapsto 3, y \mapsto 6]$ je vrednost izraza

$$3 * (x + 7) \quad \text{enaka } 30$$

V okolju $[x \mapsto 11]$ je vrednost $3 * (x + 7)$ enaka 54.

V okolju $[y \mapsto 1, z \mapsto 10]$ je vrednost $3 * (x + 7)$ nedefinirana.

Semantika velikih korakov



Pravilo:

$$\frac{\text{predpostava}_1, \dots, \text{predpostava}_n}{\text{sklep}}$$

$$\frac{\text{podnalogu}_1, \dots, \text{podnalogu}_n}{\text{nalogu}}$$

Primer:

• logika

$$\frac{A \quad B}{A \wedge B}$$

"A in B dohajemo falso, da
dohajemo A in dohajemo B."

$$\frac{A}{A \vee B}$$

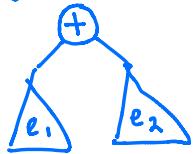
$$\frac{B}{A \vee B}$$

$$\frac{A \Rightarrow B \quad A}{B}$$

Pravila za $\eta \mid e \hookrightarrow n$:

$$\frac{\eta \mid e_1 \hookrightarrow n_1 \quad \eta \mid e_2 \hookrightarrow n_2}{\eta \mid e_1 + e_2 \hookrightarrow n_1 + n_2}$$

mišljeno kot izraz,



+ je znak

plus(e_1, e_2)

ADD

mišljeno kot matematična operacija
seštevanja

n_1 celo število
 n_2 celo število
+ sešteva

$$\frac{\eta \mid e_1 \hookrightarrow n_1 \quad \eta \mid e_2 \hookrightarrow n_2}{\eta \mid e_1 * e_2 \hookrightarrow n_1 * n_2}$$

symbol

znamenji števili

$$\frac{\eta(x) = n}{\eta \mid x \hookrightarrow n}$$

$\eta(x) = n$ v okolji η ima
 x vrednost n

$$\frac{}{\eta \mid n \hookrightarrow n}$$

↑ ↗
 številka število
 "423" \hookrightarrow 423

Primer: Okojuje $\eta = [x \mapsto 1, y \mapsto 1, z \mapsto 3]$

$$\frac{\eta(x) = 1}{\eta \mid x \hookrightarrow 1} \quad \frac{\eta(z) = 3}{\eta \mid z \hookrightarrow 3}$$

$$\frac{\eta \mid x + z \hookrightarrow 8}{\eta \mid 3 * (x + z) \hookrightarrow 24}$$

Druž način za isto zadavo: 2 rekurzivne funkcije

$$\text{eval}(\eta, e_1 + e_2) = \text{eval}(e_1) + \text{eval}(e_2)$$

$$\text{eval}(\eta, e_1 \underset{(+)}{\underset{e_2}{\textcircled{+}}} e_2) = \text{eval}(e_1) + \text{eval}(e_2)$$

$$\text{eval}(\eta, e_1 * e_2) = \text{eval}(e_1) \cdot \text{eval}(e_2)$$

$$\text{eval}(\eta, x) = \eta(x)$$

$$\text{eval}(\eta, n) = n$$

vstavi
 η !

Semantika malih korakov

$$\eta := [x \mapsto 1, y \mapsto 3]$$

$$\begin{array}{lcl} \eta \mid 3 * (x + 7) & \xrightarrow{\text{en korak}} & (3+7) \cdot (8-4) = \\ & & 10 \cdot (8-4) = \\ \eta \mid 3 * (1 + 7) & \xrightarrow{} & \underline{10} \cdot \underline{4} = \\ \eta \mid 3 * 8 & \xrightarrow{} & 40 \\ \eta \mid 24 & & \end{array}$$

Pravila: píšeme n, n_1, n_2 řádky
 e_1, e_2, \dots splošné i trate

$$\frac{\eta \mid e_1 \mapsto e'_1}{\eta \mid e_1 + e_2 \mapsto e'_1 + e_2}$$

pravila ta *
so zelo podobnou

$$\frac{\eta \mid e_2 \mapsto e'_2}{\eta \mid n_1 + e_2 \mapsto n_1 + e'_2}$$

$$\frac{}{\eta \mid n_1 + n_2 \mapsto n_1 + n_2}$$

\uparrow znak \uparrow sčítání

$$\frac{\eta(x) = n}{\eta \mid x \mapsto n}$$

$$\frac{}{\eta \mid n \mapsto n}$$

\uparrow řádka \uparrow řádlo

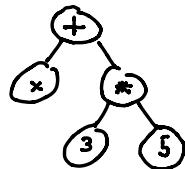
Ukazni programski jezik

Ponovimo:

aritmetični izrazi

- sintaksa (konkretna, abstraktna)

" $x + 3 * 5$ "



- operacijska/dinamična semantika

$$\frac{\text{predpostavke} \quad C_1 \quad C_2 \quad \dots \quad C_n}{C}$$

sklep

pravilo:

če imamo vse
predpostavke C_1, \dots, C_n ,
potem lahko sklepamo
 C

Podali smo pravila za

$$\eta \mid e \hookrightarrow n$$

"veliki koraki"

$\eta = [x \mapsto 5, y \mapsto 1, z \mapsto 42]$

okolje izraz vrednost izraza
(členilo)

$$\eta \mid e \mapsto e'$$

"mali koraki"

okolje izraz izraz
V okolju η se izraz e
v enem koraku
transformira v izraz e'

Primer: $\eta \mid (x + 3) \cdot 5 \hookrightarrow 40$

$$\eta \mid (x + 3) \cdot 5 \mapsto \eta \mid (5 + 3) \cdot 5 \mapsto \eta \mid 8 \cdot 5 \mapsto \eta \mid 40$$

Ukazni programske jezike:

- aritmetični izrazi
- boolovi izrazi
- ukazi: prirejanje, pogojni stavek, zanka while

Sintaksa:

Aritmetični izrazi:

```

<aritmetični-izraz> ::= <aditivni-izraz>
<aditivni-izraz> ::= <multiplikativni-izraz> | <aditivni-izraz> + <multiplikativni-izraz>
<multiplikativni-izraz> ::= <osnovni-izraz> | <multiplikativni-izraz> * <osnovni-izraz>
<osnovni-izraz> ::= <spremenljivka> | <številka> | ( <izraz> )
<spremenljivka> ::= [a-zA-Z]*
<številka> ::= -? [0-9]*

```

konkretna sintaksa
kaj pa - ?!

$$\begin{aligned} a \neq b &\Leftrightarrow !(b = a) \\ &\Leftrightarrow a < b + 1 \\ &\Leftrightarrow a = b \text{ || } a < b \end{aligned}$$

Boolovi izrazi:

```

<boolov-izraz> ::= true | false |
<aritmetični-izraz> = <aritmetični-izraz> |
<aritmetični-izraz> < <aritmetični-izraz> |
<boolov-izraz> && <boolov-izraz> |
<boolov-izraz> || <boolov-izraz> |
! <boolov-izraz>

```

abstraktna
true && false || $7 < 5$
 $(\text{true} \&\& \text{false}) \parallel (7 < 5)$
 $\text{true} \&\& (\text{false} \parallel (7 < 5))$

Ukazi:

```

<ukaz> ::= skip |
<spremenljivka> := <aritmetični-izraz> |
<ukaz> ; <ukaz> |
while <boolov-izraz> do <ukaz> done |
if <boolov-izraz> then <ukaz> else <ukaz>

```

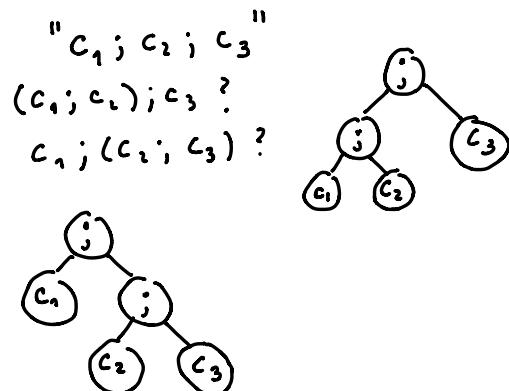
abstraktna sintaksa

Primer programa:

```

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

```



Asociativnost operatorja:

$$a \otimes b \otimes c = (a \otimes b) \otimes c \quad \text{levo asociativen}$$

$$a \oplus b \oplus c = a \oplus (b \oplus c) \quad \text{desno asociativen}$$

$$7 - 5 - 3 = (7 - 5) - 3 \quad \text{levo}$$

$$70 : 10 : 7 = (70 : 10) : 7 \quad \text{levo}$$

$$2^3 \cdot 3^2 = 2^3 \cdot (3^2 \cdot 2) \quad \text{desno} \quad 2^{3^2} = 2^{(3^2)}$$

- $p \Leftrightarrow q \Leftrightarrow r$ v resnik: $(p \Leftrightarrow q) \wedge (q \Leftrightarrow r)$
- $a \leq b \leq c$ v resnik: $a \leq b \wedge b \leq c$

Operacijska semantika

Okolje: preslikava iz spremenljivk v njihove vrednosti

$$\eta = [x \mapsto 1, y \mapsto 2, z \mapsto 42]$$

$\eta(x)$... vrednost spremenljivke x v okolju η

$$\text{Primer: } \eta(y) = 2$$

$\eta(t)$ nedeterminirano

$\eta[v \mapsto n]$ v okolju η nastavi spremenljivko v na n , vrni novo okolje

Primer:

$$\eta[x \mapsto 7] = [x \mapsto 7, y \mapsto 2, z \mapsto 42]$$

Operacijska semantika

- aritmetični izrazi: $\eta \models e \hookrightarrow n$ veliki koraki
- boolovi izrazi: $\eta \models b \hookrightarrow N$
 - boolov izraz
 - boolova vrednost: true, false

- ukaz: mali koraki

$$(\eta, C) \mapsto \eta' \quad \begin{array}{l} \text{V okolju } \eta \text{ ukaz } C \text{ konča v enem koraku} \\ \text{in novo okolje je } \eta' \end{array}$$

$$(\eta, C) \mapsto (\eta', C') \quad \begin{array}{l} \text{V okolju } \eta \text{ ukaz } C \text{ naredi en korak,} \\ \text{ičkanje se nato nadaljuje v okolju } \eta', \\ \text{izvesti moramo še ukaz } C' \end{array}$$

shift

Aritmetični izrazi:

$$\frac{}{\eta \mid n \hookrightarrow n}$$

$$\frac{\eta(x) = n}{\eta \mid x \hookrightarrow n}$$

$$\frac{\eta \mid e_1 \hookrightarrow n_1 \quad \eta \mid e_2 \hookrightarrow n_2}{\eta \mid e_1 + e_2 \hookrightarrow n_1 + n_2}$$

$$\frac{\eta \mid e_1 \hookrightarrow n_1 \quad \eta \mid e_2 \hookrightarrow n_2}{\eta \mid e_1 - e_2 \hookrightarrow n_1 - n_2}$$

$$\frac{\eta \mid e_1 \hookrightarrow n_1 \quad \eta \mid e_2 \hookrightarrow n_2}{\eta \mid e_1 * e_2 \hookrightarrow n_1 * n_2}$$

Boolovi izrazi:

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

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

$$\frac{\eta \mid b \hookrightarrow \text{false}}{\eta \mid !b \hookrightarrow \text{true}}$$

$$\frac{\eta \mid b \hookrightarrow \text{true}}{\eta \mid !b \hookrightarrow \text{false}}$$

$$\frac{}{\eta \mid b_1 \hookrightarrow \text{false}}$$

$$\frac{}{\eta \mid b_1 \& b_2 \hookrightarrow \text{false}}$$

$$\frac{\eta \mid b_1 \hookrightarrow \text{true} \quad \eta \mid b_2 \hookrightarrow v_2}{\eta \mid b_1 \&& b_2 \hookrightarrow v_2}$$

$$\frac{}{\eta \mid b_1 \hookrightarrow \text{true}}$$

$$\frac{}{\eta \mid b_1 \mid\mid b_2 \hookrightarrow \text{true}}$$

$$\frac{\eta \mid b_1 \hookrightarrow \text{false} \quad \eta \mid b_2 \hookrightarrow v_2}{\eta \mid b_1 \mid\mid b_2 \hookrightarrow v_2}$$

$$\frac{\eta \mid e_1 \hookrightarrow n_1 \quad \eta \mid e_2 \hookrightarrow n_2 \quad n_1 = n_2}{\eta \mid e_1 = e_2 \hookrightarrow \text{true}}$$

$$\frac{}{\eta \mid e_1 = e_2 \hookrightarrow \text{true}}$$

$$\frac{\eta \mid e_1 \hookrightarrow n_1 \quad \eta \mid e_2 \hookrightarrow n_2 \quad n_1 \neq n_2}{\eta \mid e_1 = e_2 \hookrightarrow \text{false}}$$

$$\frac{}{\eta \mid e_1 = e_2 \hookrightarrow \text{false}}$$

$$\frac{\eta \mid e_1 \hookrightarrow n_1 \quad \eta \mid e_2 \hookrightarrow n_2 \quad n_1 < n_2}{\eta \mid e_1 < e_2 \hookrightarrow \text{true}}$$

$$\frac{}{\eta \mid e_1 < e_2 \hookrightarrow \text{true}}$$

$$\frac{\eta \mid e_1 \hookrightarrow n_1 \quad \eta \mid e_2 \hookrightarrow n_2 \quad n_1 \geq n_2}{\eta \mid e_1 < e_2 \hookrightarrow \text{false}}$$

$$\frac{}{\eta \mid e_1 < e_2 \hookrightarrow \text{false}}$$

Ukazi:

$$(\eta, \text{skip}) \rightarrow \eta$$

$$\eta \mid e \hookrightarrow \eta$$

$$(\eta, (x := e)) \rightarrow \eta[x \mapsto \eta]$$

$$(\eta, c_1) \rightarrow (\eta', c_1')$$

$$(\eta, (c_1 ; c_2)) \rightarrow (\eta', (c_1' ; c_2))$$

$$(\eta, c_1) \rightarrow \eta'$$

$$(\eta, (c_1 ; c_2)) \rightarrow (\eta', c_2)$$

$$\eta \mid b \hookrightarrow \text{false}$$

$$(\eta, (\text{while } b \text{ do } c \text{ done})) \rightarrow \eta$$

$$\eta \mid b \hookrightarrow \text{true}$$

$$(\eta, (\text{while } b \text{ do } c \text{ done})) \rightarrow (\eta, (c ; \text{while } b \text{ do } c \text{ done}))$$

$$\eta \mid b \hookrightarrow \text{true}$$

$$(\eta, (\text{if } b \text{ then } c_1 \text{ else } c_2)) \rightarrow (\eta, c_1)$$

$$\eta \mid b \hookrightarrow \text{false}$$

$$(\eta, (\text{if } b \text{ then } c_1 \text{ else } c_2)) \rightarrow (\eta, c_2)$$

Primer: $\eta := [x \mapsto 2, y \mapsto 3, z \mapsto 10]$

Evaluiramo:

$$(\eta, (\text{if } x < y \text{ then } z := x \text{ else } z := y)) \rightarrow \\ (\eta, z := x) \rightarrow$$

$$[x \mapsto 2, y \mapsto 3, z \mapsto 2]$$

Primer: $\xrightarrow{\text{ratumeju hot}}$

$$(\eta, (\underbrace{z := x + 7}_{C_1} ; \underbrace{y := z}_{C_2} ; \text{skip} ; x := 10)) \rightarrow \\ ([x \mapsto 2, y \mapsto 3, z \mapsto 9], (y := \underline{z} ; \text{skip} ; x := 10)) \rightarrow$$

$$\eta \mid x < y \hookrightarrow \text{true}$$

$$\eta \mid x \hookrightarrow 2$$

$$\eta \mid x + 7 \hookrightarrow 9$$

$$\eta \mid z \hookrightarrow 9$$

$$(\eta, c_1) \rightarrow \eta' ?$$

$$\eta \mid b \hookrightarrow \text{true}$$

$$(\eta, c_1) \rightarrow (\eta', c_1')$$

dva koraka!

$$(\eta, (\text{if } b \text{ then } c_1 \text{ else } c_2)) \rightarrow (\eta', c_1')$$

to pravilo je slabše (je ok, a nema grdo)

$$\begin{aligned} ([x \mapsto 2, y \mapsto 9, z \mapsto 9], (\text{skip} ; x := 10)) &\mapsto \\ ([x \mapsto 2, y \mapsto 9, z \mapsto 9], (x := 10)) &\mapsto \\ [x \mapsto 10, y \mapsto 9, z \mapsto 9] \end{aligned}$$

Primer:

$$\begin{aligned} (\eta, \underbrace{((z := x + 7 ; y := z) ; \text{skip})}_{c_1} ; \underbrace{x := 10}_{c_2}) &\mapsto \\ c_1 \downarrow c'_1 & \\ (\eta[z \mapsto 9], (((y := z) ; \text{skip}) ; x := 10)) &\mapsto \dots \\ &\text{isti rezultat, a} \\ &\text{drug vrstni red uporabe} \\ &\text{pravil.} \end{aligned}$$

Primer: $[x \mapsto 1]$

$$\begin{aligned} ([x \mapsto 1], (\text{while } x > 0 \text{ do } x := x + 1 \text{ done})) &\mapsto \begin{array}{l} [x \mapsto 1] \mid x > 0 \hookrightarrow \text{true} \\ [x \mapsto 2] \mid x + 1 \hookrightarrow 2 \end{array} \\ ([x \mapsto 1], (x := x + 1 ; \text{while } x > 0 \text{ do } x := x + 1 \text{ done})) &\mapsto \\ ([x \mapsto 2], (\text{while } x > 0 \text{ do } x := x + 1 \text{ done})) &\mapsto \\ ([x \mapsto 2], (x := x + 1 ; \text{while } x > 0 \text{ do } x := x + 1 \text{ done})) &\mapsto \\ ([x \mapsto 3], (\text{while } x > 0 \text{ do } x := x + 1 \text{ done})) &\mapsto \dots \\ \text{in taku naprej za vedno} & \text{nestekano zaporedje krovakov} \end{aligned}$$

Dilema o sintaksi:

Ali je $\text{if } b \text{ then } c_1 \text{ else } c_2 ; c_3$

tole: $(\text{if } b \text{ then } c_1 \text{ else } c_2) ; c_3$

tole: $\text{if } b \text{ then } c_1 \text{ else } (c_2 ; c_3)$?

Temu se izognemo tako, da popravimo sintaksco:

programmers of this app are idiots

Ukazi:

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

Dileme ni več:

if b then c_1 else c_2 ; c_3 maybe end

if b then c_1 else c_2 ; c_3 end

if b then c_1 else end; c_3

Kdaj sta dva programa "ekvivalentna"?

$X := X + 1 ; X := X + 1$



$X := X + 2$

Prevajalnik za comm

Oglejmo si implementacijo (različice) programskega jezika comm iz [PL Zoo](#). Tako kot vsi jeziki v PL Zoo, je comm implementiran v programskem jeziku OCaml.

Jezik comm vsebuje:

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

Komentar:

```
new x := 2 + 3 in
  x := x + 1 ;
  if x < 7 then
    print x
  else
    skip
```

v Javi:

```
{ int x = 2 + 3 ;
  x = x + 1
  if (x < 7) {
    print (x);
  } else
  { }
}
```

Dogovoriti se moramo, kaj pomeni

```
new x := e in c1 ; c2
```

Imamo dve možnosti:

1. (`new x := e in c1`) ; `c2` – x je veljaven samo v `c1`
2. `new x := e in (c1 ; c2)` – x je veljaven v `c1` in v `c2`

Dogovorimo se, da velja 2. možnost.

Ogledamo si sestavne dele implementacije:

- 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 programskeh jezikov poteka preko večih stopenj, z vmesnimi jeziki. Vsak naslednji jezik je nekoliko bolj preprost in bližje strojni kodi.

Na primerih preizkusimo, kako se prevajajo programi.

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 konkreten program, ki počne to, kar zahteva specifikacija.

Specifikacija je lahko podana bolj ali manj natančno, v človeškem jeziku ali zapisana v formalnem matematičnem jeziku. Zakaj potrebujemo specifikacije? Nekateri odgovori:

- 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 le majhen košček specifikacij, t.i. Hoarovo logiko, s katero izražamo dejstva o programih v ukaznem programskem jeziku in dokazujemo njihovo pravilnost.

Hoarova logika

V Hoarovi logiki pišemo *Hoarove trojice*

$\{ P \} c \{ Q \}$

kjer sta P in Q logični formuli in c ukaz. Formuli P pravimo *predpogoj* (angl. *precondition*), formuli Q pravimo *končni pogoj* (ang. *postcondition*). V resnici poznamo dve različici trojic:

Delna pravilnost

$\{ P \} c \{ Q \}$

pomeni

Če velja P in če bo ukaz c končal, potem bo veljal Q

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.

Primer 1

Program c zamenja vrednosti spremenljivk x in y:

{ x = m ∧ y = n } c { x = n ∧ y = m }

Tu moramo predpostaviti, da sta m in n *duhova* (angl. ghost variable), se pravi spremenljivki, ki se ne pojavljata v c.

Primer 2

Program c poskrbi, da je x manjši ali enak y:

{ true } c { x ≤ y }

Ali znamo zapisati tak program? Da, na primer

x := 0 ; y := 1

Specifikacija to dovoli. Verjetno smo hoteli v resnici tole:

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

Ko delamo s Hoarovo logiko, običajno pišemo pogoje in kodo navpično, da lahko med vrstice vrivamo pogoje.

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

Seveda potrebujemo nekakšna pravila sklepanja.

Pravila sklepanja

Za Hoarovo logiko veljajo naslednja pravila sklepanja.

Splošna pravila

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

$$\frac{P' \Rightarrow P \quad \{P\} c \{Q\} \quad Q \Rightarrow Q'}{\{P'\} c \{Q'\}}$$

$$\frac{\{P_1\} c \{Q_1\} \quad \{P_2\} c \{Q_2\}}{\{P_1 \wedge P_2\} c \{Q_1 \wedge Q_2\}}$$

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

$$FV(x \leq y \vee x > 0) = \{x, y\}$$

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

Pozor: ne sprašujemo, ali program dejansko spremeni vrednost, ampak le, ali se pojavlja v njem ukaz oblike $x := \dots$:

$$FA(\text{if } 5 < 3 \text{ then } x := y + 3 \text{ else skip end}) = \{x\}$$

$$FA(x := x) = \{x\}$$

Velja pravilo:

$$\frac{FV(P) \cap FA(c) = \emptyset}{\{P\} c \{P\}}$$

Pravilo za skip

$$\frac{}{\{P\} \text{ skip } \{P\}}$$

Pravilo za pogojni stavek

$$\frac{\{P \wedge b\} c_1 \{Q\} \quad \{P \wedge \neg b\} c_2 \{Q\}}{\{P\} \text{ if } b \text{ then } c_1 \text{ else } c_2 \text{ end } \{Q\}}$$

Pravilo za $c_1 ; c_2$

$$\frac{\{P\} c_1 \{Q\} \quad \{Q\} c_2 \{R\}}{\{P\} c_1 ; c_2 \{R\}}$$

Pravilo za zanko while

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

Formuli P pravimo *invarianta* zanke while.

Pravilo za prirejanje

$$\frac{\{ P[x \mapsto e] \} x := e \{ P \}}{\{ P \}}$$

Zapis $P[x \mapsto e]$ pomeni "v formuli P zamenjaj pojavitev x z izrazmo e.
Primer:

$$P \equiv x < n \wedge y + x = z$$

$$P[x \mapsto x+1] \equiv x+1 < n \wedge y + (x+1) = z$$

$$P[x \mapsto 0] \equiv 0 < n \wedge y + 0 = z$$

Operaciji, ki zamenja spremenljivko z nekim izrazom pravimo *substitucija*.

Popolna pravilnost

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

$$\frac{[P \wedge b] c_1 [Q] \quad [P \wedge \neg b] c_2 [Q]}{[P] \text{ if } b \text{ then } c_1 \text{ else } c_2 \text{ end } [Q]}$$

Prva izjema je pravilo

$$\frac{FV(P) \cap FA(c) = \emptyset}{\{ P \} c \{ P \}}$$

Kaj je narobe z

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

Ne velja v naslednjem primeru:

```
[ 1 = 1 ]
while true do skip done
[ 1 = 1 ]
```

Zgornja trditev ne velja (ni res, da se program konča).

Zato pravilo predelamo takole:

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

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:

$$0.1 > 0.01 > 0.001 > 0.0001 > \dots$$

Pravilo za popolno pravilnost `while` se glasi:

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

$$[P \wedge b \wedge e = z] c [P \wedge e < z]$$

$$[P] \text{ while } b \text{ do } c \text{ done } [\neg b \wedge P]$$

Stranski pogoji:

- količina e se ne more v nedogled zmanjševati
- z je duh, spremenljivka, ki se ne pojavlja nikjer drugje v P , b ali c .

Kako pa ta pravila v praksi uporabljamo? Poglejmo nekaj primerov.

Primeri

Primer 1

Zapiši s Hoarovo logiko:

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

Rešitev:

Poglejmo kar vse možne kombinacije:

- $\{ \text{true} \} c \{ \text{true} \}$ - vedno velja, ker `true` itak velja vedno
- $\{ \text{true} \} c \{ \text{false} \}$ — pravi "c se ne ustavi"
 - če velja `true` in če se bo ustavil c , bo veljalo `false`
 - če se bo ustavil c , bo veljalo `false`
 - s formulo: $c \text{ se ustavi} \Rightarrow \text{false}$
 - spomnimo se: $P \Rightarrow \text{false}$ je logično ekvivalentno $\neg P$
 - s formulo: $\neg (c \text{ se ustavi})$ (upoštevamo zgornjo ekvivalenco)

- z besedami: c se ne ustavi
- { false } c { true } - vedno velja, ker iz false sledi karkoli
- { false } c { false } - vedno velja, ker iz false sledi karkoli
- [true] c [true] - pomeni "c se ustavi"
 - če velja true, se bo c ustavil in veljalo bo true
 - c se bo ustavil in veljalo bo true
 - c se bo ustavil
- [true] c [false] - nikoli ne velja
 - če velja true, se bo c ustavil in veljalo bo false
 - s formulo: true \Rightarrow "c se ustavi" \wedge false
 - s formulo: true \Rightarrow false
 - s formulo: false
- [false] c [true] - vedno velja, ker iz false sledi karkoli
- [false] c [false] - vedno velja, ker iz false sledi karkoli

Koristni kombinaciji:

1.{ true } c { false } — pomeni "c se ne ustavi"

1. [true] c [true] - pomeni "c se ustavi"

Primer 1.5

Dokaži pravilnost programa:

```
{ true }
x := 7
{ x < 10 }
```

Zelo natančna rešitev:

```
{ true }
{ 7 < 10 }          -- P ≡ (x < 10)
{ P[x ↦ 7] }
x := 7
{ P }
{ x < 10 }
```

Praktična rešitev:

```
{ true }
x := 7
{ x = 7 } - logično sklepanje
{ x < 10 }
```

Primer 1.75

```
{ i < n }      - naslednja vrstica je ekvivalentna tej
{ (i + 1) - 1 < n }
i := i + 1
{ i - 1 < n } - uporabili smo pravilo za pritejanje
{ i < n + 1 }
```

Primer 2 - 1.125

```
{ i < n }
i := i + j
{ i < n + j }
```

Primer 2

Dokaži pravilnost programa:

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

Rešitev:

```
{ x ≤ y }
s := (x + y) / 2
{ x ≤ y ∧ s = (x + y) / 2 } - logično sklepanje
{ x ≤ s ≤ y }
```

Varianta:

```
{ x < y }
s := (x + y) / 2
{ x < y ∧ s = (x + y) / 2 } - logično sklepanje lari fari
{ x < s < y }
```

NI RES! Protiprimer: $x = 4, y = 5, s = 4$.

Primer 3

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 }
{ p = a ^ i ∧ i ≤ b }
while i < b do
    { i < b ∧ p = a ^ i ∧ i ≤ b }
    { i < b ∧ p · a = a ^ (i+1) ∧ i ≤ b }
    p := p * a ;
    { i < b ∧ p = a ^ (i+1) ∧ i ≤ b }
    { i+1 < b+1 ∧ p = a ^ (i+1) ∧ i+1 ≤ b+1 }
    i := i + 1
    { i < b + 1 ∧ p = a ^ i ∧ i ≤ b+1 } – sklepamo: če i < b + 1,
    potem i ≤ b
    { p = a ^ i ∧ i ≤ b }
done
{ p = a ^ i ∧ i ≤ b ∧ ¬(i < b) } – logično sklepamo: i ≤ b in b ≤ i
potem i = b
{ p = a ^ i ∧ b = i } – logično sklepamo
{ p = a ^ b }
```

Popolna pravilnost: potrebujemo količino e, ki se zmanjšuje in se ne more zmanjševati v nedogled. Predlog:

$$e \equiv b - i$$

To je celo število. Ali je nenegativno? Naša invarianta vsebuje dejstvo $i \leq b$, od koder seveda sledi, da je $b - i \geq 0$.

Zdaj bi moralo pazljivo dokazati, da se e res zmanjša. Pričakujemo, da e zmanjša za 1:

```
{ e = z }
while ...
...
done
{ e = z - 1 < z }
```

Lambda račun

Funkcijski predpis

V matematiki poznamo zapis za *funkcijski predpis*:

$$x \mapsto e$$

To preberemo "x se slika v e", pri čemer je e neki izraz, ki lahko vsebuje x.
Primer:

$$x \mapsto x^2 + 3 \cdot x + 7$$

Kadar imamo funkcijski predpis, ga lahko *uporabimo* na argumentu. Denimo, če je

$$f := (x \mapsto x^2 + 3 \cdot x + 7)$$

Kadar pišemo

$$f(x) := x^2 + 3 \cdot x + 7$$

je to v bistvu okrajšava za $f := (x \mapsto x^2 + 3 \cdot x + 7)$.

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

Pravzaprav ni nobene potrebe, da funkcijski predpis poimenujemo f, lahko bi ga kar neposredno uporabljali in tvorili aplikacijo:

$$(x \mapsto x^2 + 3 \cdot x + 7)(3)$$

To se morda zdi nenavadno, a je lahko koristno v programiraju (kot bomo videli kasneje). Nekateri programske 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, x^2 + 3*x + 7]`

Računsko pravilo (β -redukcija) za funkcijski predpis

Vsi znamo računati s funkcijskimi predpisi in aplikacijami, čeprav se tega morda ne zavedamo. Računsko pravilo, ki se iz zgodovinski razlogov imenuje " β -redukcija", pravi

$$(x \mapsto e_1)(e_2) = e_1[x \leftarrow e_2]$$

in ga preberemo:

Če uporabimo funkcijski predpis $x \mapsto e_1$ na argumentu e_2 , dobimo izraz e_1 , v katerem x zamenjamo z e_2 .

Zamenjavi spremenljivke x za neki izraz pravimo *substitucija*. Primer:

$$(x \mapsto x^2 + 3 \cdot x + 7)(3) = 3^2 + 3 \cdot 3 + 7$$

Pozoro: pravilo za funkcijski zapis *ne* trdi $(x \mapsto x^2 + 3 \cdot x + 7)(3) = 25$, ampak samo, da lahko x zamenjamo s 3 in dobimo $3^2 + 3 \cdot 3 + 7$. Torej se pogovarjamo o računskih pravilih, ki so bolj osnovna kot računanje s števili!

Vezane in proste spremenljivke

V funkcijskem predpisu

$$x \mapsto x^2 + 3 \cdot x + 7$$

se x imeuje *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:

$$a \mapsto a^2 + 3 \cdot a + 7$$

Poudarimo, da štejemo funkcijskih predpisov 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:

$$x \mapsto a \cdot x^2 + b \cdot x + c$$

Tu so a , b in c proste spremenljivke. Teh ne smemo preimenovati, ker bi se pomen izraza spremenil, če bi to storili. (Pravzaprav imamo še 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, a je prosta
- v vsoti $\sum_i i^*(i-1)$ je i vezana spremenljivka
- v limiti $\lim_{x \rightarrow a} (x - a)/(x + a)$ 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.

Proste spremenljivke se ne smejo ujeti

Kadar imamo proste in vezane spremenljivke, moramo paziti, da se prosta spremenljivka ne "ujame", kar pomeni, da bi zaradi preimenovanja vezane spremenljivke prosta spremenljivka postala vezana. Na primer:

1. $x \leftrightarrow a + x -$ "prištej a "
2. $y \leftrightarrow a + y -$ "prištej a "
3. $a \leftrightarrow a + a -$ "podvoji"

Vidimo, da se je v tretjem primeru a "ujela", ko smo x preimenovali v a . Mimogrede, treba je ločiti med tema dvema izrazoma:

- $y \leftrightarrow a + y -$ "prištej a "
- $a \leftrightarrow a + y -$ "prištej y "

Gnezdeni funkcijski predpisi

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

1. $(x \leftrightarrow (y \leftrightarrow x \cdot x + y))(42) = (y \leftrightarrow 42 \cdot 42 + y)$
2. $((x \leftrightarrow (y \leftrightarrow x \cdot x + y))(42))(1) = (y \leftrightarrow 42 \cdot 42 + y)(1) = 42 \cdot 42 + 1$
3. $(f \leftrightarrow f(f(3))) (n \leftrightarrow n \cdot n + 1) = (n \leftrightarrow n \cdot n + 1) ((n \leftrightarrow n \cdot n + 1)(3)) = (n \leftrightarrow n \cdot n + 1)(3 \cdot 3 + 1) = (3 \cdot 3 + 1) \cdot (3 \cdot 3 + 1) + 1$

Lahko se zgodi, da se zaradi vstavljanja enega funkcijskega predpisa v drugega kakšna vezana spremenljivka ujame. V takem primeru predhodno preimenujemo vezano spremenljivko. Primer:

- pravilno: $(x \leftrightarrow (y \leftrightarrow x \cdot y^2)) (z + 1) = (y \leftrightarrow (z + 1) \cdot y^2)$
- narobe: $(x \leftrightarrow (y \leftrightarrow x \cdot y^2)) (y + 1) = (y \leftrightarrow (y + 1) \cdot y^2)$
- pravilno: $(x \leftrightarrow (y \leftrightarrow x \cdot y^2)) (y + 1) = (x \leftrightarrow (a \leftrightarrow x \cdot a^2)) (y + 1) = (a \leftrightarrow (y + 1) \cdot a^2)$

λ -račun

Zapis $x \leftrightarrow e$ postane dolgovezen, ko funkcijske zapise gnezdimo. Uporabili bomo λ -zapis:

$\lambda x . e$

To je prvotni zapis funkcijskih predisov, 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

$$e_1 e_2 e_3 = (e_1 e_2) e_3$$

V abstrakciji λ vedno veže največ, kolikor lahko. Torej je $\lambda x . e_1 e_2 e_3$ je enako $\lambda x . (e_1 e_2 e_3)$ in ni enako $(\lambda x . e_1) e_2 e_3$.

Kadari imamo gnezdene abstrakcije

$$\lambda x . \lambda y . \lambda z . e$$

to pomeni $\lambda x . (\lambda y . (\lambda z . e))$. Dogovorimo se še, da lahko tako gnezedeno abstrakcijo krajše zapišemo

$$\lambda x y z . e$$

Evaluacijske strategije

Pravilo za računanje lahko uporabimo na različne načine. Primer:

$$(\lambda x . (\lambda f . f x) (\lambda y . y)) ((\lambda z . g z) u)$$

je enak

$$(\lambda x . (\lambda f . f x) (\lambda y . y)) (g u)$$

in prav tako

$$(\lambda x . (\lambda y . y) x) ((\lambda z . g z) u)$$

Vendar pa velja lastnost *confluence*, ki pravi, da vrstni red računanja ni pomemben. Natančneje, če ima e dva možna računska koraka, $e \rightarrow e_1$ in $e \rightarrow e_2$, potem lahko v e_1 in v e_2 izvedemo take računske korake, da se bosta pretvorila v isti izraz.

V zgornjem primeru:

$$\begin{aligned} & (\lambda x . (\lambda f . f x) (\lambda y . y)) (g u) = \\ & (\lambda x . (\lambda y . y) x) (g u) = \\ & (\lambda x . x) (g u) = \\ & g u \end{aligned}$$

in

```


$$\begin{aligned}
(\lambda x . (\lambda y . y) x) ((\lambda z . g z) u) &= \\
(\lambda x . x) ((\lambda z . g z) u) &= \\
(\lambda z . g z) u &= \\
g u
\end{aligned}$$


```

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

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

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

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

Programiranje v λ -računu

λ -račun je splošen programski jezik, ki je po moči ekvivalenten Turingovim strojem. Ogledamo si nekaj primerov.

Identiteta

```
id :=  $\lambda x . x$ 
```

Kompozicija

```
compose :=  $\lambda f g x . g (f x)$ 
```

Konstantna funkcija

```
const :=  $\lambda c x . c$ 
```

Boolove vrednosti in pogojni stavek

```

true :=  $\lambda x y . x$ 
false :=  $\lambda x y . y$ 
if :=  $\lambda b t e . b t e$ 

```

Urejeni pari

```

pair :=  $\lambda a b . \lambda p . p a b ;$ 
first :=  $\lambda p . p (\lambda x y . x) ;$ 
second :=  $\lambda p . p (\lambda x y . y) ;$ 

```

Ostale primere si ogledamo v PL Zoo, programski jezik lambda.

λ -racun

Funkcijski predpis

$$x \mapsto x^2 + 3$$

" x se slika u $x^2 + 3$ "

$$\left| \begin{array}{l} f : A \rightarrow B \\ f \text{ je funkcija iz } A \text{ u } B \\ f : x \mapsto \dots \\ f \text{ slika } x \text{ u } \dots \end{array} \right.$$

$$\begin{aligned} f(x) &:= x^2 + 3 && \leftarrow \begin{array}{l} \text{članjena za} \\ \text{teg}^2 \\ \text{osnova} \end{array} \\ f &:= (x \mapsto x^2 + 3) && \\ f(3) &= 3^2 + 3 = 12 && \\ (x \mapsto x^2 + 3)(3) &= 3^2 + 3 = 12 && \\ &&& \left(\begin{array}{l} (3+7) \cdot 8 \\ a := 3+7 \\ a \cdot 8 \end{array} \right) \\ &&& \hline (x \mapsto x^2 + 3)(3) && \\ f(x) &= x^2 + 3 && \\ f(3) &&& \end{aligned}$$

1. Predpis: $x \mapsto e$ "x se slika u e"

2. Uporaba (aplikacija): $(x \mapsto e_1)(e_2)$ "uporabi predpis $x \mapsto e_1$ na argumentu e_2 "

3. Racunarsko pravilo (β -redukcija):

$$(x \mapsto e_1)(e_2) = \underbrace{e_1[x \leftarrow e_2]}_{\text{n } e_1 \text{ zamenjaj } x \text{ z } e_2} \text{ SUBSTITUCIJA ali ZAMENJAVA}$$

Primer:

$$(x \mapsto 2x + 7)(3 + 8) = 2(3 + 8) + 7$$

x smo zamenjali u $2x + 7 \Rightarrow 3 + 8$.

Vetana in proste spremenljivke

VERANA V ZANKI FOR

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

PROSTA SPREMENLJIVKA

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

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

for (s = 0; s < 10; s++) { s += s; } *S smo "ujeli" z veravo v tanki!*

for (i = 0; i < 10; i++) { t += i; }

Primeri:

$$\int_a^b \frac{1 + cx}{1 + cx^3} dx$$

prosta
vetana

$$\int_a^b \frac{1 + ct}{1 + ct^3} dt$$

$$\int_a^d \frac{1 + cx}{1 + cx^3} dx$$

$$\int_a^b \frac{1 + ex}{1 + ex^3} dx$$

$$\sum_{i=0}^n a \cdot r^i = a \cdot \frac{1 - r^{n+1}}{1 - r} = \sum_{sin=0}^n a \cdot r^{sin}$$

Vetana

$$\int_0^{\pi/2} \cos(\sin) d \sin$$

for (while = 0; while < 10; while++) { s += while; }
slaba ideja

$$x \mapsto ax^2 + 3$$

vezana prosta konstanta

Gnezdimo predpise:

$$x \mapsto (y \mapsto ax^2 + by - 1)$$

"x se slike v funkcijo, ki sprejme y in vrne $ax^2 + by - 1$

$$u \mapsto ((x \mapsto x^2 + 3u)(17))$$

$u \mapsto 17^2 + 3u$

$u \mapsto 289 + 3u$

RAZLIČNI PREDPISI,
 KI DOLOČAJO
 ISTO FUNKCIJO

$$\left. \begin{array}{l} 3 \cdot (7+8) \\ 3 \cdot 15 \end{array} \right\}$$

RAZLIČNI ARITMETIČNI IZRAZI,
 KI DOLOČAJU ISTO ŠTEVILO

45

Primer med odmorom:

$$ax^2 + by - 1$$

VEZANE:
 PROSTE: x, y, a, b

$$y \mapsto ax^2 + by - 1$$

VEZANE: y
 PROSTE: x, a, b

$$x \mapsto (y \mapsto ax^2 + by - 1)$$

VEZANE: x, y
 PROSTE: a, b

```

for (int i = 0; i < 10; i++) {
    s += i;
    for (int i = 0; i < 20; i++) {
        t += i * i;
    }
}

```

i prekriva (shadow) *i* v notranji zanki

$$x \mapsto (3x + (x \mapsto 2x+1)(x+3))$$

$$x \mapsto (3x + (l \mapsto 2l+1)(x+3))$$

λ -racun

Namesto

$$x \mapsto e$$

x se sliká v e

uporabimo

$$\lambda x. e$$

x se sliká v e

Alonzo Church 1930

Programski jezik :

~~sterila~~

~~true, false~~

~~tabele~~

~~objekti~~

~~stringi~~

funkcije

~~zanke while, for~~

~~if - then - else~~

~~rekurzija~~

~~tipi~~

Sintaksa λ -računa:

- funkcijski predpis, abstrakcija:

$$\lambda x. e$$

" x izraz je smuo abstrahiramo"

- uporaba ali aplikacija:

$$e_1(e_2) \quad f(a)$$
$$e_1, e_2 \quad f a$$

" e_1 uporabimo na e_2 "

$$Ax$$

Aplikacija je levo asociativna

$$e_1 e_2 e_3 = (e_1 e_2) e_3$$

λ veže do konca:

$$\lambda x. e_1 e_2 e_3 = \cancel{(\lambda x. e_1) e_2} e_3$$
$$\cancel{(\lambda x. (e_1 e_2)) e_3}$$
$$\underline{\lambda x. (e_1 e_2 e_3)} ?$$

$$\lambda x. f \times y (\lambda z. z z) = \lambda x. (f \times y (\lambda z. (z z)))$$

$$x \mapsto ((f(x))(y))(z \mapsto z(z))$$

$$x \mapsto (f \times y (z \mapsto z z))$$

$\hat{x} \wedge_x \lambda x$

Računsko pravilo (β -redukcija):

$$(\lambda x. e_1) e_2 \rightsquigarrow e_1[x \leftarrow e_2]$$

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

ohrašamo

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

Funkcijski predpis sprejme en argument:

$$\lambda x. e$$

Kako naredimo funkcijo, ki sprejme dva (ali več) argumentov?

1. Namesto "funkcija sprejme dva argumenta"

"funkcija sprejme en argument, ki je urejeni par"

$$f: \mathbb{R}^2 \rightarrow \mathbb{R} \quad f: \underbrace{\mathbb{R} \times \mathbb{R}} \rightarrow \mathbb{R}$$

2. "f sprejme x in y" predelamo v

"f sprejme x in urne funkcijo, ki sprejme še y"

Primer:

$$f(x,y) := x^2 + y^3 - 7 \quad \begin{matrix} \text{dva argumenta } x, y \\ \text{druga komponenta} \end{matrix}$$

$$f(p) := (\pi_1 p)^2 + (\pi_2 p)^3 - 7$$

$$f(x) := (y \mapsto x^2 + y^3 - 7)$$

$$f := (x \mapsto (y \mapsto x^2 + y^3 - 7))$$

$$f(x) := (y \mapsto x^2 + y^3 - 7)$$

$$f := (x \mapsto (y \mapsto x^2 + y^3 - 7))$$

$$f := \lambda x. \lambda y. x^2 + y^3 - 7$$

$$\lambda x. y. x^2 + y^3 - 7$$

Namesto $e_1 + e_2$ pišemo plus $e_1 e_2 \dots$

Programiramo v λ -računu

Identiteta : $\lambda x. x$

Boolove vrednosti in pogojni stavek:

iščemo izrate

true , false , if

Namensko
if (p) { A } else { B }

pišemo if $p A B$

da velja: if true $A B = A$

if false $A B = B$

true := $\lambda a b. a$

false := $\lambda a b. b$

if := $\lambda p a b. p a b$

if true $A B =$

$(\lambda p a b. p a b) \text{ true } A B =$

$(\lambda a b. \text{ true } a b) A B =$

true $A B =$

$(\lambda a b. a) A B = (\lambda b. A) B = A$

Vaja: Premi if false $A \ B = B$

Urejeni pari:

| Števno

pair	first	second
	fst	snd

dc netja:

$$\text{fst}(\text{pair } u v) = u$$

$$\text{snd}(\text{pair } u v) = v$$

Matematika:

$$(,) \quad (u, v)$$

pair $u v$

$\pi_1 p$ prva komponenta fst p

$\pi_2 p$ druge komponente snd p

$$\pi_1(u, v) = u$$

$$\pi_2(u, v) = v$$

~~$$\text{pair} := \lambda u v. \lambda s. s u v$$~~

~~$$\text{fst} := \lambda x y. x$$~~

~~$$\text{snd} := \lambda x y. y$$~~

$$\text{fst} := \lambda p. p(\lambda x y. x)$$

$$\text{snd} := \lambda p. p(\lambda x y. y)$$

$$\text{pair} := \lambda u v. s. s u v$$

$$\begin{aligned}
 (\lambda p. p) \text{ true } A \ B &= \text{true } A \ B \\
 &= ((\lambda a b. a) A) B \\
 &= (\lambda b. A) B \\
 &= A
 \end{aligned}$$

Štuila:

$$0 = \lambda f x . x$$

$$1 = \lambda f x . f x$$

$$2 = \lambda f x . f(f x)$$

$$n = \lambda f x . \underbrace{f(f(\dots f)}_n x \dots)$$

Seštevanje: iščemo plus, da velja:

$$\text{plus } n m = \lambda f x . \underbrace{f(f(\dots f)}_{n+m} x \dots \underbrace{f(f(\dots f x))}_m$$

$$\text{plus} = \lambda n m . \lambda f x . \underbrace{n f}_{\underbrace{f(\dots f}_{n}(m f x) \dots)} \underbrace{(f(\dots f}_{m} (f \dots f x))}$$

Rekurziona definicija:

$$0 = D(0)$$

$$x = 2x + 3 \quad (x = -3)$$

$$x = x + 7 \quad ??$$

Rekursivna definicija:

$$x = f x$$

\hookrightarrow funkcia

$$x = f x \quad f = \lambda z. 2z + 3$$

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

$$\text{fact} = F \text{ fact} \quad \text{kej je}$$
$$F = \lambda f. \lambda n. \text{if } n=0 \text{ then } 1 \text{ else } n \cdot f(n-1)$$

Siemo program fix, da velja

$$\text{fix } F = F(\text{fix } F)$$

$$\text{fix} = \lambda f. (\lambda x. f(x x))(\lambda x. f(x x))$$

$$\begin{aligned}\text{fix } F &= (\lambda x. F(x x))(\lambda x. F(x x)) \\ &= \underbrace{F((\lambda x. F(x x))(\lambda x. F(x x)))}_{\text{fix } F} \\ &= F(\text{fix } F) = F(F(\text{fix } F)) \\ &= F(F(F(\text{fix } F)))\end{aligned}$$

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 omenjam grozne sintakse, zaradi katerih so programi neučinkoviti.)

Obdržimo, kar ima λ -račun koristnega, a ga nato nadgradimo z manjkajočimi koncepti. Pomembna spoznanja so:

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 potem takem 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 želimi imeti, ne pa nujno kako se izračuna. Postopek, s katerim pridemo do rezultata je nato v večji ali manjši meri prepuščen programskemu jeziku.

Podatki

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

Programski jezik naj programerju omogoči neposredno izražanje idej.

Če mora programer neko idejo v programu izraziti tako, da jo simulira, je večja možnost napake. Poleg tega prevajalnik ne bo imel informacije o tem, kaj programer počne, in ne bo razpoznal manj napak in imel manj možnosti za optimizacijo.

Ponazorimo to načelo z idejo. Denimo, da želimo računati s seznamimi števil. Potem od programskega jezika pričakujemo *neposredno* podporo za sezname: sezname lahko preprosto naredimo, jih analiziramo, podajamo kot argumente. 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 v jezik. To je prikladna rešitev, vendar pa ne moremo pričakovati, da bomo lahko z vgrajenimi podatkovnimi strukturami zadovoljili vse potrebe. V vsakem primeru moramo progamerju omogočiti, da definira *nove* strukture in *nove* načine organiziranja idej, ki jih načrtovalec jezika ni vnaprej predvidel. Ražlični 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. Spoznajmo *neporedne* konstrukcije podatkovnih tipov, ki *niso* simulacije. Navdih bomo vzeli iz matematike, kjer poznamo operacije, s katerimi gradimo množice.

Konstrukcije množic

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

Zmnožek ali kartezični produkt

Zmnožek ali kartezični produkt množic A in B je množica, katere elementi se imenujejo *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:

$$\begin{aligned}\pi_1(x, y) &= x \\ \pi_2(x, y) &= y\end{aligned}$$

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 četvertice (x, y, z, t) in štiri projekcije, π_1, π_2, π_3 in π_4 .

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 $y \in B$ lahko tvorimo element $\iota_2(y) \in A + B$

Predstavljamо 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 označujemo z 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 imamo element $u \in A + B$, potem lahko *obravnavamo dva primera**, saj je u bodisi oblike $\iota_1(x)$ za neki $x \in A$ bodisi oblike $\iota_2(x)$ za neki $y \in B$.

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 programske 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 N množica naravnih števil, lahko izdelek predstavimo kot element množice

$$(B \times N) + N + N$$

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

Eksponent ali množica funkcij

Eksponent B^A , ki ga pišemo tudi $A \rightarrow B$, je množica vseh funkcij iz A v B . Če je $f \in B^A$, pravimo, da je A **domena** in B **kodomena** funkcije f . O funkcijah tu ne bomo povedali veliko več, jih pa bomo s pridom uporabljali.

Podatkovni tipi

V programskej jeziku ne govorimo o množicah, ampak o **tipih**, ki so podobni množicam, a so bolj splošni in imajo širšo uporabno vrednost. Tipi so zelo splošen in uporaben koncept, ki presega meje programiranja in celo računalništva. Tipi se uporabljajo 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 , bomo to zapisali z

$$e : T$$

Ta zapis nas spominja na zapis $e \in T$ iz teorije množic. Vendar pa je bolje, da na tip T ne gledamo kot na "zbirko elementov", ampak kot na informacijo o tem, kakšen podatek je e in kaj lahko z njim počnemo.

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

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

Zmnožek tipov

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

```
Standard ML of New Jersey v110.81 [built: Mon Oct  2 10:01:15 2017]
- (3, "banana") ;
val it = (3,"banana") : int * string
```

Zapisali smo urejeni par $(3, "banana")$. SML je ugotovil, da je tip tega urejenega para $\text{int} * \text{string}$ in to izpisal. Z izpisom `val it = ...` je povedal še, da lahko zadnjo vrednost, ki smo jo izračunali, dobimo z `it`:

```
- 3 + 6 ;
val it = 9 : int
- it ;
val it = 9 : int
- 3 * 14 ;
val it = 42 : int
- it * 100 ;
val it = 4200 : int
```

Če želimo vpeljati definicijo, to naredimo z `val x = ...`:

```
- val i = 10 + 3 ;
val i = 13 : int
- val j = 100 + i * i ;
val j = 269 : int
```

Tvorimo lahko tudi urejene n-terice, za poljuben n:

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

Projekcije π_1, π_2, \dots v SML pišemo kot `#1, #2, ...`:

```
- #2 (1, "banana", false, 2) ;
val it = "banana" : string
- #3 (1, "banana", false, 2) ;
val it = false : bool
- val p = (1, "banana", false, 2) ;
val p = (1,"banana",false,2) : int * string * bool * int
```

```
- #3 p ;
val it = false : bool
```

Enotski tip

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

```
- ( ) ;
val it = () : 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 = {*}:`

$$A \cong 1 \times A$$

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 predpisani element, potem vemo, kaj bo vrnila, in tega ni treba razlagati.

Zapomnimo si torej, da funkcija, ki "ne vrne ničesar" v resnici vrne `()`. V SML 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!

Zapis

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

`(leto, mesec, dan, ura, minuta, sekunda, milisekunda)`

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

Težavo razrešimo tako, da komponent ne štejemo po vrsti, ampak jih pojmenujemo. Dobimo tako imenovani tip *zapis* (angl. *record*):

```
- { ime = "Mojca", priimek = "Novak" } ;
val it = {ime="Mojca",priimek="Novak"} : {ime:string, priimek:string}
```

Torej je $\{x_1=e_1, \dots, x_i=e_i\}$ kot urejena terica (e_1, \dots, e_i), le da smo poimenovali njene komponente x_1, \dots, x_i . Tip zapisa pišemo $\{x_1:a_1, \dots, x_i:a_i\}$. Če bi bil to kartezični produkt, bi ga zapisali $a_1 * \dots * a_i$.

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

```
- { priimek = "Novak", ime = "Mojca" } ;
val it = {ime="Mojca",priimek="Novak"} : {ime:string, priimek:string}
```

V resnici so urejene večterice poseben primer zapisov, namreč takih, ki imajo polja po vrsti poimenovana 1, 2, 3, ...

```
- { 1 = 2, 2 = "banana", 3 = false, 4 = 2 } ;
val it = (2,"banana",false,2) : int * string * bool * int
```

Z zapisom lahko zapišemu tudi urejeno "enerico":

```
- {1 = "foo"};
val it = {1="foo"} : {1:string}
```

V Pythonu se to zapiše ("foo",).

Do komponente z imenom foo dostopamo s #foo:

```
val mati = {ime="Neza",priimek="Cankar"} : {ime:string, priimek:string}
- #ime mati ;
val it = "Neza" : string
```

Definicije tipov v SML

V SML lahko s type a = ... definiramo okrajšave za tipe, da jih ni treba vedno znova pisati. Na primer:

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

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

type color = { red : real, green : real, blue : real }

type krneki = int * bool * string
```

Vsota tipov

Elemente vsote množic A + B smo označevali z τ_1 in τ_2 . Izbor oznak je z matematičnega stališča nepomemben, namesto τ_1 in τ_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 SML najprej definirati z datatype. Zgornji primer izdelkov v spletni trgovini, bi zapisali takole:

```
datatype izdelek
  = Cevelj of {blue:real, green:real, red:real} * int
  | Palica of int
  | Posoda of int
```

Ta definicija pravi, da je izdelek vsota treh tipov: prvi tip je zmnožek tipov {blue:real, green:real, red:real} in int. Drugi in tretji tip sta oba int. Za oznake smo izbrali Cevelj, Palica in Posoda. Tem oznakam v SML 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
```

Razločevanje primerov

Kot smo omenili, potrebujemo zapis za razločevanje primerov. Nadalujmo 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 SML zapišemo s case:

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

Splošna oblika stavka case je

```
case e of
  p1 => e1
  | p2 => e2
  | p3 => e3
  :
  | pi => ei
```

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

```
- case z of
  Cevelj (b, v) => if v < 35 then 15 else 25
  | Posoda y => 7 ;
= = stdIn:42.2-44.19 Warning: match nonexhaustive
  Cevelj (b,v) => ...
  Posoda y => ...

uncaught exception Match [nonexhaustive match failure]
  raised at: stdIn:44.19
```

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

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

V C je to tip `enum`. Imena konstruktorjev se lahko pišejo z malo začetnico. V SML bi lahko `bool` definirali sami, če ga še ne bi bilo:

```
datatype bool = false | true
```

V resnici SML zgornjo definicijo spremo in z njo *prekrije* že obstoječo definicijo tipa `bool`, kar lahko pripelje do velike zmede!

Vzorci v stavku `case` 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 sene 1000 evrov
- palica dolžine $x \neq 42$ stane $1 + 2 * x$ evrov
- posoda stane 7 evrov ne glede na prostornino

Pripadajoči stavek `case` se glasi:

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

Vzorce lahko uporabljammo tudi v definicijah vrednosti `val`:

```

- val (Posoda p) = Posoda 10 ;
val p = 10 : int

- val Covelj (x, y) = Covelj ({red=1.0,green=0.5,blue=0.0}, 43) ;
val x = {blue=0.0,green=0.5,red=1.0} : {blue:real, green:real, red:real}
val y = 43 : int

- val Covelj ({red=r,green=g,blue=b},v) = Covelj ({red=1.0,green=0.5,
  blue=0.0}, 43) ;
val b = 0.0 : real
val g = 0.5 : real
val r = 1.0 : real
val v = 43 : int

```

Vzorcem se bomo bolj podrobno še posvetili.

Funkcijski tip

Funkcijski tip $a \rightarrow b$ je tip funkcij, ki sprejmejo argument tipa a in vrnejo rezultat tipa b . V SML λ -abstrakcijo $\lambda x . e$ zapišemo kot $fn\ x \Rightarrow e$:

```

- fn x => 2 * (x + 3) + 3 ;
val it = fn : int -> int

```

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

```

- fn x => (fn y => 2 * x - y + 3) ;
val it = fn : int -> int -> int

```

SML je izračunal tip funkcije $int \rightarrow int \rightarrow int$. Operator \rightarrow je *desno asociativen*, torej je

$$a \rightarrow b \rightarrow c \equiv a \rightarrow (b \rightarrow c)$$

Tip $int \rightarrow int \rightarrow int$ torej opisuje funkcije, ki sprejmejo int in vrnejo $int \rightarrow int$. Funkcije lahko tudi uporabljamо:

```

- (fn x => (fn y => 2 * x - y + 3)) 10 ;
val it = fn : int -> int

```

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

```

- (fn x => (fn y => 2 * x - y + 3)) 10 3 ;
val it = 20 : int

```

Funkcijo lahko poimenujemo:

```

- val f = fn x => x * x + 1 ;
val f = fn : int -> int
- f 10 ;
val it = 101 : int

```

Namesto `val f = fn x => ...` lahko pišemo tudi `fun f => ...`

```
- fun g x = x * x + 1 ;
val g = fn : int -> int
- g 10 ;
val it = 101 : int
```

Definicija s `fun` je lahko rekurzivna:

```
- fun fact n = (if n = 0 then 1 else n * fact (n - 1)) ;
val fact = fn : int -> int
- fact 10 ;
val it = 3628800 : int
```

Kot vidimo, SML 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:

```
- fn (x, y) => (y, x) ;
val it = fn : 'a * 'b -> 'b * 'a
```

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

```
- fn (x, y, z) => (x, y + z, x) ;
val it = fn : 'a * int * int -> 'a * int * 'a
```

Ko zapišemo funkcijo, lahko podamo tip njenih argumentov:

```
- fn (x : string) => x ;
val it = fn : string -> string
- fn x => x ; val it = fn : 'a -> 'a
```

Tipi

Množice:

- osnovne množice $\mathbb{N}, \mathbb{R}, \mathbb{Z}, \dots$
- konstrukuirje:
 - kartesiani produkt $A \times B$
 - vsota $A + B$
 - podmnožica $\{x \in A \mid P(x)\}$
 - eksponent B^A množica funkcij iz A v B
 - ⋮

Kartesiani produkt ali zmnožek

$$A \times B$$

elementi: (x, y) lejer $x \in A$ in $y \in B$
 ↑ urejeni par

$$p \in A \times B$$

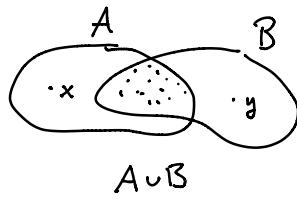
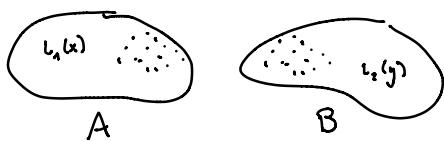
projekciji: $\pi_1(p) \in A$ prva projekcija
 $\pi_2(p) \in B$ druga projekcija

Velja: $\pi_1(17, 8) = 17$ $\pi_1(x, y) = x$
 $\pi_2(x, y) = y$

$A \times B \times C \times D$, urejene četverice (x, y, z, t)
 projekcije $\pi_1, \pi_2, \pi_3, \pi_4$

Vsota ali disjunktna unija

$A + B$



$A \cup B$

$$\alpha \beta \gamma \delta \eta \zeta \xi \eta \\ n^2$$

Elementi: $l_1(x) \quad \text{če } x \in A$
 $l_2(y) \quad \text{če } y \in B$

$x \in A \Rightarrow$ vi res $x \in A + B$
 je res $l_1(x) \in A + B$

Primer: $\mathbb{N} \times \mathbb{N} + \mathbb{Z}$
 $l_1(3, 7), l_2(-6), l_2(0), \cancel{l_3(1)}$

Primer: $\mathbb{N} + \mathbb{N}$
 $\uparrow \quad \downarrow$
 $l_1(7) \quad l_2(7)$

$\mathbb{N} \cup \mathbb{N} = \mathbb{N}$
 $\uparrow \quad \nwarrow$

$A + B + C + D$
 $\underbrace{l_1, l_2, l_3, l_n}_{\text{injekuje}}$

$u \in A + B$ lahko obravnavamo primere:

u je bokisi oblike $l_1(x)$ za neki $x \in A$
 bokisi oblike $l_2(y)$ za neki $y \in B$

$$\begin{cases} \dots x \dots & \text{če je } u = l_1(x) \\ \dots y \dots & \text{če je } u = l_2(y) \end{cases}$$

Rekurzija in rekurzivni tipi

Splošna oblika rekurzije

Obravnavajmo rekurzivno funkcijo f , ki računa faktorielo:

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

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

```
def telo(g, n):
    if n == 0:
        return 1
    else:
        return n * g(n - 1)

def f(n):
    return telo(f, n)
```

Samo drugi del je rekurziven. Zapišimo to še v SML. Prvotna definicija faktoriele:

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

Ko ločimo definicijo od rekurzivnega sklica:

```
fun telo g n = (if n = 0 then 1 else n * g (n - 1))
```

```
fun f n = telo f n
```

Še malo drugače:

```
fun telo g = (fn n => if n = 0 then 1 else n * g (n - 1))
```

```
fun f n = telo f n
```

Vsako rekurzivno funkcijo lahko razstavimo na ta način in drugi del je vedno enak. Definirajmo si funkcijo *rek* (v angleščini običajno rec ali *fix*), ki sprejme telo t rekurzivne definicije in vrne pripadajočo rekurzivno funkcijo:

```
fun rek t = (fn x => t (rek t) x)
```

Vsa rekurzija je shranjena v *rek*, od tu naprej je ne potrebujemo več:

```
fun f = rek (fn g => fn n => if n = 0 then 1 else n * g(n - 1))
```

Poglejmo si tip funkcije rek. SML je izpeljal njen tip:

(('a -> 'b) -> 'a -> 'b) -> 'a -> 'b

Tipa 'a in 'b sta *parametra*, ki označujeta poljubna tipa. Zapišimo z α in β :

(($\alpha \rightarrow \beta$) → ($\alpha \rightarrow \beta$)) → ($\alpha \rightarrow \beta$)

Preberemo: "rek je funkcija, ki sprejme funkcijo t tipa $(\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)$ in vrne funkcijo tipa $\alpha \rightarrow \beta$."

Poglejmo si postopek še enkrat:

1. $f\ n = (\text{if } n = 0 \text{ then } 1 \text{ else } n * f(n - 1))$
2. $f = (\lambda n . \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n - 1))$
3. $f = t\ f$ kjer je $t = (\lambda g . \text{if } n = 0 \text{ then } 1 \text{ else } n * g(n - 1))$
4. $f = \text{rek } t$

Kadar imamo preslikavo h in točko x , ki zadošča enačbi $x = h(x)$, pravimo, da je x **negibna točka** preslikave h . V numeričnih metodah je eden od osnovih postopkov reševanja enačb ta, da enačbo zapišemo v obliki $x = h(x)$ in nato iščemo njeno rešitev kot zaporedje približkov

$x_0, h(x_0), h(h(x_0)), h(h(h(x_0))), \dots$

Negibne točke so pomembne tudi na drugih področjih matematike in o njih matematiki veliko vedo.

Opomba: če imam enačbo $l(x) = d(x)$, jo lahko prepišemo v $x = d(x) - l(x) + x$ in definiramo $h(x) = d(x) - l(x) + x$, da dobimo $x = h(x)$.

Ugotovili smo, da je tudi rekurzivna definicija funkcije f pravzaprav enačba, ki ima oblike negibne točke:

$f = t\ f$

Pomnimo:

Rekurzivno definirana funkcija je negibna točka.

Rekurzivna funkcija več argumentov

Ali to deluje tudi za rekurzivne funkcije dveh argumentov? Seveda!

1. $s(n, k) = (\text{if } k = 0 \text{ then } n \text{ else } s(n + k, k - 1))$
2. $s = (\text{fn } (n, k) \Rightarrow \text{if } k = 0 \text{ then } n \text{ else } s(n + k, k - 1))$
3. $s = t\ s$, kjer je $t = (\text{fn } g \Rightarrow \text{fn } (n, k) \Rightarrow \text{if } k = 0 \text{ then } n \text{ else } g(n + k, k - 1))$

Hkratna rekurzivna definicija

Kaj pa definicija rekurzivnih funkcij f in g , ki kličeta druga drugo? Primer: funkcija f kliče f in g , funkcija g pa kliče f :

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

Če obravnavamo f in g skupaj kot urejeni par (f, g) dobimo

$$(f, g) = ((\lambda x . \text{if } x = 0 \text{ then } 1 + f(x - 1) \text{ else } 2 + g(x - 1)), (\lambda y . \text{if } y = 0 \text{ then } 1 \text{ else } 3 * f(y - 1)))$$

To je *rekurzivna definicija urejenega para (funkcij)*.

kar prepišemo v

$$(f, g) = t (f, g)$$

kjer je

$$t = \lambda (f', g') . ((\lambda x . \text{if } x = 0 \text{ then } 1 + f'(x - 1) \text{ else } 2 + g'(x - 1)),
(\lambda y . \text{if } y = 0 \text{ then } 1 \text{ else } 3 * f'(y - 1)))$$

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

Iteracija je poseben primer rekurzije

V proceduralnem programiraju poznamo zanke, na primer zanko while. Ali je tudi ta negibna točka? Če upoštevamo ekvivalenco

while b do c done

in

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

vidimo, da je zanka while b do c done negibna točka. Če pišemo W za našo zanko:

$$W \equiv (\text{if } b \text{ then } (c ; W) \text{ else skip})$$

Torej je W in s tem zanka while b do c done negibna točka funkcije:

$$t = (\lambda W . \text{if } b \text{ then } (c ; W) \text{ else skip})$$

Tudi **iteracija** je negibna točka!

Opomba: zanko while lahko na zgornji način "odvijamo v nedolged":

Faza 0:

while b do c done

Faza 1:

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

```

else skip

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

```

Faza 2:

```

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!

Rekurzivni tipi

Do sedaj smo spoznali podatkovne tipe:

- produkt $a * b$
- vsota $a + b$
- eksponent $a \rightarrow b$

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

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

Zapis [1, 2, 3] je okrajšava za $1 :: (2 :: (3 :: []))$.

Sezname 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 SML se tako definicijo zapiše takole:

```
datatype seznam = nil | cons of int * seznam
```

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

In spet vidimo, da je rekuzija 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`.

Induktivni tipi

Izhajamo iz definicije seznama:

```
datatype 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`?

Posebej pomembni so **induktivni podatkovni tipi**. 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 sezname
3. končna drevesa
4. abstraktna sintaksa jezika:
 - programski jeziki
 - jeziki za označevanje podatkov
5. hierarhija elementov v uporabniškem vmesniku

Primer: naravna števila

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:

```
datatype stevilo = Nic | Naslednik of stevilo
```

Ta definicija ni učinkovita. Poskusimo 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

Ideja: z $shl0\ n$ predstavimo število $2 \cdot n + 0$ in z $shl1\ n$ predstavimo število $2 \cdot n + 1$. Primer: število

```
shl0 (shl1 (shl0 (shl1 0)))
```

je število 10. Kot podatkovni tip:

```
datatype stevilo = zero | shl0 of stevilo | shl1 of stevilo
```

Ni optimalno:

```
0 = shl0 0 = shl0 (shl0 0)
```

Vaja: poišči predstavitev dvojiških števil z induktivnimi tipi (lahko jih je več), da bo imelo vsako število natanko enega predstavnika.

Primer: JSON

Poglejmo si [definicijo standarda JSON](#) in iz nje izluščimo podatkovni tip:

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

Primer rekurzivnega tipa, ki ni induktiven:

Rekurzivni tipi so lahko že do nenavadni:

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

Poskusite si predstavljati, kaj so vrednosti tega tipa...

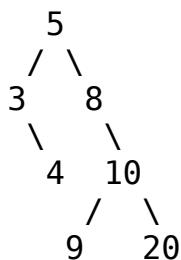
Struktturna rekurzija

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 **sestavljen**o 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:



Podatkovni tip v SML se glasi:

```
datatype 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.

Vaja: Sestavi funkcije za iskanje, vstavljanje in brisanje elementov v iskalnem drevesu.

Koinduktivni tipi in leno računanje

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

Tipičen primer je **komunikacijski tok podatkov**:

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

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* neksnončna, a je *potencialno* neskončna, kar pomeni, da lahko dva procesa komunicirata v nedogled in brez vnaprej postavljenih omejitve.

Koinduktivni tipi so rekurzivni tipi, ki dovoljujejo tudi neskončne vrednosti. Vendar pozor, kadar imamo opravka z neskončno velikimi seznamami, 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.

Koinduktivni tipi

Ponovimo osnovno o koinduktivnih tipih

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

Tipičen primer je **komunikacijski tok podatkov**:

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

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* nekončna, a je *potencialno* neskončna, kar pomeni, da lahko dva procesa komunicirata v nedogled in brez vnaprej postavljenih omejitv.

Koinduktivni tipi so rekurzivni tipi, ki dovoljujejo tudi neskončne vrednosti. Vendar pozor, kadar imamo opravka z neskončno velikimi seznamami, 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.

Tokovi

Poglejmo si različico tokov, ki so 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 SML

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

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

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

```
Cons (1, Cons (1, Cons (1, Cons (1, ...))))
```

Tokovi v Haskellu

Ista definicija v Haskellu deluje, ker ima Haskell koinduktivne tipe. Poglejmo si to na primeru.

Tokovi v SML

V SML lahko *simuliramo* tokove z uporabo tehnike *zavlačevanja* (angl. "thunk"). Imamo težavo, da hoče SML takoj izračunati preostanek toka. V splošnem lahko "zavlačujemo" z računanjem izraza e tako, da ga zapakiramo v funkcijo `fn () => e` in dobimo "thunk". Kasneje ga lahko "aktiviramo" tako, da ga uporabimo na `()`.

Izpeljava tipov

Kako programski jeziki uporabljajo tipe

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

Kako striktni so tipi

Tipi so lahko bolj ali manj **striktni**. Če so popolnoma striktni, ima vsak izraz v veljavnem programu tip (SML, OCaml, 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 SML so tipi striktni. Tip urejene trojice je bolj informativen:

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

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#, SML, 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.

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 pravino 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: SML, OCaml, Haskell.

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**.

Izpeljava tipov

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

Kakšen tip ima funkcija $\lambda x . x$, oziroma v SML `fn x => x`? Možnih je veliko odgovorov:

- $\text{int} \rightarrow \text{int}$
- $\text{bool} \rightarrow \text{bool}$
- $\text{int} * \text{int} \rightarrow \text{int} * \text{int}$
- $\alpha \text{ list} \rightarrow \alpha \text{ list}$ za poljuben α
- $\beta \rightarrow \beta$ 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 $\beta \rightarrow \beta$ *glavni* tip funkcije `fn 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 parameter).

SML je načrtovan tako, da ima vsak veljaven izraz glavni tip, ki ga SML sam izpelje.

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 zadostovati
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 $\tau_1 = \text{int}$, $\tau_2 = \text{int}$
Podobno obravnavamo ostale aritmetične izraze $e_1 * e_2$, $e_1 - e_2$, ...

- boolov izraz $e_1 \text{ and } e_2$: 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 $\tau_1 = \text{int}$, $\tau_2 = \text{int}$

- 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 , $\tau_1 = \text{bool}$, $\tau_2 = \tau_3$

- 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 $\tau_1 \times \tau_2$, 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_1 , $\tau = \alpha \times \beta$.

- 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_1 , $\tau = \alpha \times \beta$.

- funkcija $fn\ x \Rightarrow e$: 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 $fn\ x \Rightarrow e$ je $\alpha \rightarrow \tau$ 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 , $\tau_1 = \tau_2 \rightarrow \alpha$

- 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

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

...
 $l_i = d_i$

v neznankah $\alpha, \beta, \gamma, \delta, \dots$ 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 zvržemo 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 $\alpha \leftrightarrow d$. Povsed v r in E zamenjamo α z d in v r dodamo rešitev $\alpha \leftrightarrow d$
 - če je enačba oblike $l = \alpha$, kjer je α neznanka, imamo primer, ki je simetričen prejšnjemu
 - če je enačba oblike $(l_1 \rightarrow l_2) = (d_1 \rightarrow d_2)$, 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 $(l_1 \rightarrow l_2) = (d_1 \times d_2)$, postopek prekinemo, ker *ni rešitve*.

Kako to deluje, si poglejmo na primerih.

Primer 1

Izpelji glavni tip funkcije

`fn x => x + 3`

Odgovor:

Primer 2

Izpelji glavni tip funkcije

`fn f x => f (x, x)`

Odgovor:

Primer 3

Izpelji glavni tip izraza

```
if 3 < 5 then (fn x => x) else (fn y => y + 3)
```

Odgovor:

Primer 4

Izpeli glavni tip izraza

```
if 3 < 5 then (fn x => x) else (fn y => (y, y))
```

Odgovor:

Primer 5

Izpeli glavni tip rekurzivne funkcije

```
fun f x = (if x = 0 then 1 else x * f (x - 1))
```

Odgovor:

Churchovi numerali

Kakšen je tip števila 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 SML:

```
val zero  = (fn f => fn x => x) ;
val one   = (fn f => fn x => f x) ;
val two   = (fn f => fn x => f (f x)) ;
val three = (fn f => fn x => f (f (f x))) ;
```

Churchovi-Scottovi numerali

Kakšen je tip števila 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 SML:

```
val zero  = (fn f => fn x => x) ;
val one   = (fn f => fn x => f zero x) ;
val two   = (fn f => fn x => f one (f zero x)) ;
val three = (fn f => fn x => f two (f one (f zero x))) ;
```

Koinduktivni Tipi

• Induktivni tipi:

rekurzivni tip, vrednosti su konine

primer: konini seznam, konina drevesa, ...

Koinduktivni tipi

rekurzivni tipi, vrednosti konine in neskoncne

primer: tok podatkov (neskoncni seznam)

Neučakano računanje (eager, call by value)

- v aplikaciji f e najprej izračunamo e
in nato rezultat vstavimo v f

(Java, C, C++, Python, SML, Ocaml, Javascript)

Leno računanje (lazy, call by need)

- v aplikaciji f e neizračunan e vstavimo v f,
e se bo izračunal, če ga bo f zares uporabil

(Haskell)

if p then e₁ else e₂

↑ ↑ ↑
neučakano leno leno

Polimorfizem & Izpeljave tipov

Primer: $\text{fn } x \Rightarrow \underbrace{x + 3}_{\text{int}}$
 $\alpha \rightarrow \text{int}$

$x : \alpha$

Ravnano $x + 3$

- tip x je α
- tip 3 je int

tip $x + 3$ je int, enačbe $\alpha = \text{int}$, $\text{int} = \text{int}$

Kandidat: $\alpha \rightarrow \text{int}$

enačbe: $\alpha = \text{int}$, $\text{int} = \text{int}$ register $\alpha \rightarrow \text{int}$

Vstavimo: $\text{int} \rightarrow \text{int}$ ODGOVOR

Primer:

$\text{if } 3 < 5 \text{ then } (\text{fn } x \Rightarrow x) \text{ else } (\text{fn } y \Rightarrow y + 3)$

$\underbrace{\quad}_{\text{bool}}$ $\underbrace{\quad}_{\alpha \rightarrow \alpha}$ $\underbrace{\quad}_{\beta \rightarrow \text{int}}$ $\beta = \text{int}$, $\text{int} = \text{int}$

1. $\text{fn } x \Rightarrow x$

$\underbrace{\quad}_{\alpha \rightarrow \alpha}$ $x : \alpha$

Ravnano tip: x ima tip α

2. $\text{fn } y \Rightarrow y + 3$
 $\beta \rightarrow \text{int}$

$y : \beta$

Ravnano: $y + 3$ ima tip int in
 β int imamo enačbi:
 $\beta = \text{int}$, $\text{int} = \text{int}$

Kandidat za if then else:

$$\alpha \rightarrow \alpha \quad \text{enakbe: } \beta = \text{int}, \text{int} = \text{int}$$
$$(\alpha \rightarrow \alpha) = (\beta \rightarrow \text{int})$$

Resujemo enakbe:

$$\text{int} = \text{int}$$

$$\forall \text{zanesmo } (\alpha \rightarrow \alpha) = (\beta \rightarrow \text{int}), \text{ rabijsimo}$$

$$\alpha = \beta, \quad \alpha = \text{int}$$

$$\text{Imamo enakbe: } \beta = \text{int}, \alpha = \beta, \alpha = \text{int}$$

$$\forall \text{zanesmo } \beta = \text{int}. \quad \text{Risitv:} \quad \beta \mapsto \text{int}$$

$$\text{Imamo enakbe: } \alpha = \text{int}, \alpha = \text{int}$$

$$\forall \text{zanesmo } \alpha = \text{int}. \quad \text{Risitv:} \quad \alpha \mapsto \text{int}$$

$$\text{Imamo enakbe: } \text{int} = \text{int}$$

$$\text{Odgovor: } \text{int} \rightarrow \text{int}$$

Specifikacija, implementacija, abstrakcija

Specifikacija & implementacija

Specifikacija (angl. specification) S je *zaheva*, 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 strankom 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 programskej jezikih poskrbimo za zapis specifikacij in kako programski jezik preveri, ali dana koda (implementacija) zadošča dani specifikaciji.

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 zakonam morajo zadoščati operacije

Primer: grupa

- signatura:
 - množica G
 - operacija $\cdot : G \times G \rightarrow G$
 - operacija $^{-1} : G \rightarrow G$
 - konstanta $e : G$

- aksiomi:

$$\begin{aligned}x \cdot (y \cdot z) &= (x \cdot y) \cdot z \\x \cdot e &= x \\e \cdot x &= x \\x \cdot x^{-1} &= e \\x^{-1} \cdot x &= e\end{aligned}$$

Primer: usmerjen graf

- signatura:
 - množica V (vozlišča)
 - množica E (povezave)
 - operacija $\text{src} : E \rightarrow V$ (začetno vozlišče povezave)
 - operacija $\text{trg} : E \rightarrow V$ (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.

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 dve uporabi specifikacij:

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

Specifikacije v Javi

V Javi je specifikacija S podana z vmesnikom

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

v katerem lahko naštejemo metode in tipe. To v grobem ustreza signaturi, vendar smo omejeni na en sam tip (razred). Na primer, usmerjeni grafi so naravno podani z dvema tipoma (za vozlišča in za povezave), ki ju moramo v Javi razdeliti na dva razreda. To seveda ne pomeni, da so vmesniki v Javi neuporabni! Pravzaprav je vsaka možnost podajanja specifikacij vedno dobrodošla.

Čeprav je to nekoliko nenavadno, lahko zapišemo signaturo za grupo in za graf v Javi:

```
public interface Group {  
    public Group e();  
    public Group mul(Group x, Group y);  
    public Group inv(Group x);  
}
```

Enotski element bi morda želeli deklarirati kot statično polje

```
public static Group e;  
a se Java pritožuje.
```

Kako narediti specifikacijo usmerjenih grafov z vmesnikom, ni povsem jasno. Morda bi poskusili takole:

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

A to ni *povsem* isto kot zgornja specifikacija, ker vzame V in E kot *vhodni podatek*, naša specifikacija pa vsebuje V in E kot del signature.

Specifikacije v SML

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

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

In takole za usmerjeni graf:

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

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 večim vmesnikom. (Opomba: podrazredi so mehanizem, ki se *ne* uporablja za specifikacijo in implementacijo.)

Implementacija v SML

Implementacija v SML se imenuje **struktura (angl. structure)**, ker se v algebri matematični objekti v splošnem imenujejo "(algebraične) strukture". Struktura je skupk definicij tipov in vrednosti, lahko pa vsebuje tudi še nadaljnje podstrukture.

Primer implementacij grup in grafov si ogledamo v datoteki `algebra.sml`.

V praksi seveda implementiramo podatkone strukture in ostale računalniške zadeve, a analogija z algebrskimi strukturami je pomembna s teoretičnega vidika.

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 **abstraktна**.

Programski jeziki omogočajo abstrakcijo v večji ali manjši meri:

- Java nadzoruje dostopnost do komponent z določili `private`, `public` in `protected`
- Python nima nikakršne abstrakcije
- SML omogoča abstrakcijo z določilom `:>`, ki ga preizkusimo na datoteki `algebra.sml`.

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.

Generično programiranje v Javi

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

```
public class Knjižnica3D<Driver extends GraphicsDriver> {  
    ...  
}
```

Generično programiranje v SML

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

Funktor je preslikava iz struktur v strukturo in je bolj splošen kot generični razredi v Javi (ker lahko struktura vsebuje podstrukture in definicije večih tipov, razred pa je v grobem definicija enega samega tipa in pripadajočih funkcij).

Funktor F , ki sprejme strukturo S in vrne strukturo T zapišemo takole:

```
functor F(S) =  
struct  
    (definicija strukture T)  
end
```

V `algebra.sml` je primer preprostega funkторja. Bolj uporaben primer sledi.

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 podatkovnih vrst
- `empty` je prazna podatkovna 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 $(\text{SOME } x, q')$ 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.

Implementacija v SML

Oglejmo si implementacijo v SML (datoteka `priority_queue.sml`).

Implementacija v Javi

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

Signatura:

- podatkovni tip `element`
- operacija `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`

Zahete 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

Primer: množice

Na vajah boste na dva načina implementirali končne množice. Oglejmo si SML signaturo, ki jih opisuje (datoteka `set.sml`).

Specifikacija, implementacija, abstrakcija

Specifikacija : opis, zahteva, ki pove kaj želimo

Implementacija : primerch tega, kar specifikaciję opisuje

✓ algebri : grupa, vektorski prostor, kolobar, grafi, ...

Grupa :

- množica G
- element $e \in G$
- operacija $\cdot : G \times G \rightarrow G$
- operacija $^{-1} : G \rightarrow G$

} Signatura

} Specifikacija

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 :

$$G = \mathbb{Z}_3 = \{0, 1, 2\}$$

$$e = 0$$

tu bi moral
biti + vendar
v specifikaciji plšie
da mora biti .

\cdot	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

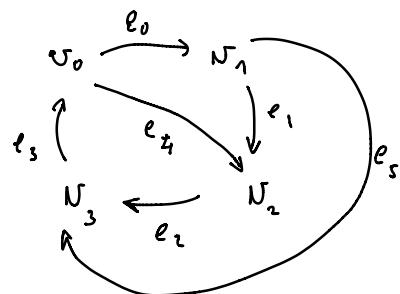
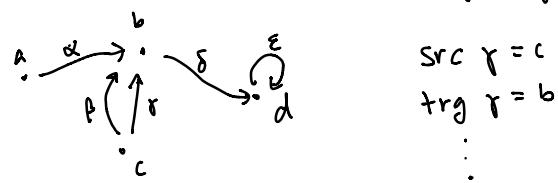
-1	0	1	2
0	2	1	0

} implementacija grupe

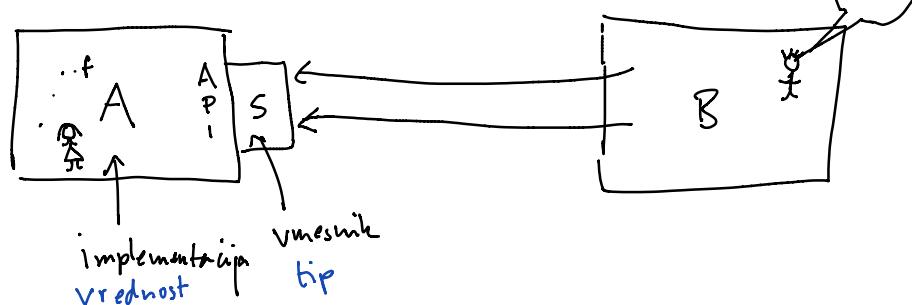
$$\text{let } \cdot = +$$

$+$	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

Usmerjen graf



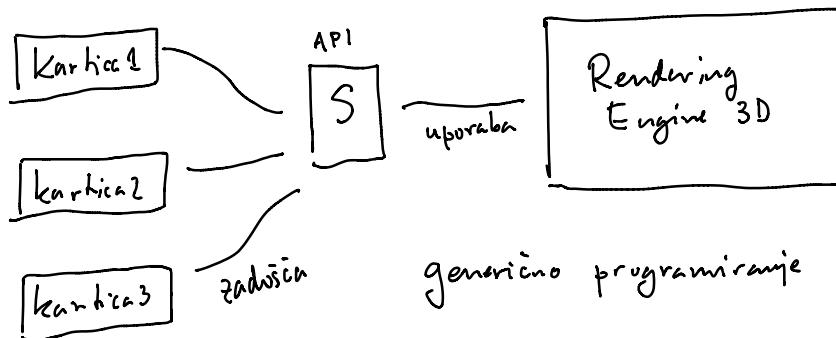
1. elipsa



2. elipsa

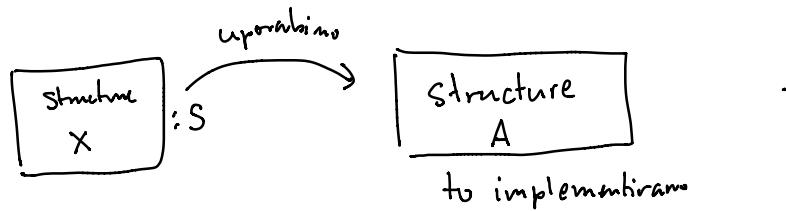
A.f!

42 : int
 ↑
 implementacija vrednost ↑
 specifikacija tip



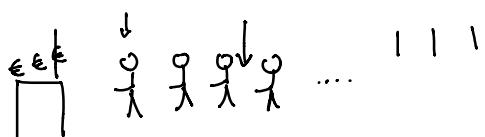
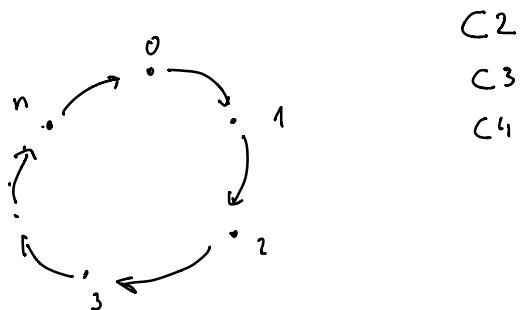
$$\text{fn } (x, y) \Rightarrow (y, x) : \alpha \times \beta \rightarrow \beta \times \alpha$$

↑ ↑
generating



funktor: praheredit struktur u strukture

Kohabur $R \longrightarrow$ Polna mnica idealov $\times R$



Še nekaj o signaturah

Zadnjič smo se spraševali, kako narediti signaturo za grupo v Javi. V SML se glasi takole:

```
signature GROUP =
sig
  type t
  val e : t
  val mul : t -> t -> t
  val inv : t -> t
end
```

Aljaž Eržen mi je poslal idejo, ki dobro deluje. V Javi moramo signaturo *parametrizirati* s tipom, namesto da ga vstavimo v singaturo:

```
public interface Group<T> {
    T e();
    T mul(T a, T b);
    T inv(T a);
}
```

Nato lahko definiramo grupo Z_3 takole:

```
public class Z3 implements Group<Z3> {
    :
}
```

To malce spominja na operacijo curry v funkciskem programiranju:

$T \times \dots : \text{Structure}$

$\text{Type} \rightarrow \dots : \text{Structure}$

Curry:

$A \times B \rightarrow C$

$A \rightarrow (B \rightarrow C)$

And now for something completely different.

Logično programiranje

Logična pravila

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 med izvajanjem predelamo v končno *vrednost*.

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

Hornove formule

V logiki prvega reda lahko zapišemo formule sestavljene iz konstant \perp , T , veznikov \wedge , \vee , \Rightarrow , \neg in kvantifikatorjev \forall , \exists . Take formule so lahko precej zapletene in niso primerne za logično programiranje, kjer se omejimo na tako imenovane **Hornove formule**, ki so oblike

$$\forall x_1, \dots, x_i . (\Phi_1 \wedge \Phi_2 \wedge \dots \wedge \Phi_j \Rightarrow \Psi)$$

Tu so Φ_1, \dots, Φ_j in Ψ **osnovne formule**, se pravi vsaka od njih je oblike

$$p(t_1, \dots, t_i)$$

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

Poseben primer Hornove formule je **dejstvo**:

$$\forall x_1, \dots, x_i . \Psi$$

Drugi primer je formula brez kvantifikatorjev (v kateri ni spremenljivk, samo konstante):

$$\Phi_1 \wedge \Phi_2 \wedge \dots \wedge \Phi_j \Rightarrow \Psi$$

To je res Hornova formula, če si predstavljam, da je oblike

$$\forall x_1, \dots, x_i . (\Phi_1 \wedge \Phi_2 \wedge \dots \wedge \Phi_j \Rightarrow \Psi)$$

kjer je $j = 0$.

Poglejmo si nekaj primerov.

Primer

Hornova formula

$$\forall a . (pes(a) \Rightarrow zival(a))$$

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

Primer

Hornova formula

$$\forall x y z . (otrok(x, y) \wedge otrok(y, z) \wedge zenska(z) \Rightarrow `babica(x, z))$$

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 ".

Vzgojen primer

Formula

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

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

Primer

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

$$\begin{aligned} \forall n . n + 0 &= n \\ \forall k \ m . k + \text{succ}(m) &= \text{succ}(k + m) \end{aligned}$$

Pravzaprav v Prologu ni funkcij, težko definiramo vsoto kot funkcijo! Definirati moramo *relacijo*, ki predstavlja funkcijo:

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

Za funkcijo seštevanja: namesto operacije $+$ bomo definirali relacijo *vsota*, da bo veljalo:

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

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

$$\begin{aligned} \forall n . \text{vsota}(n, \text{zero}, n) \\ \forall k \ m \ n . \text{vsota}(k, m, n) \Rightarrow \text{vsota}(k, \text{succ}(m), \text{succ}(n)) \end{aligned}$$

Prva formula očitno ustreza prvemu aksiomu, druga pa je ekvivalentna

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

kar je ekvivalentno drugemu aksiomu (premisli zakaj!).

Naloga

S Hornovimi formulami zapiši Peanova aksioma za množenje:

$$\begin{aligned} \forall n . n \cdot 0 &= 0 \\ \forall k \ m . k \cdot \text{succ}(m) &= k + k \cdot m \end{aligned}$$

Uporabi relacijo *vsota* iz prejšnjega primera, ter relacijo *zmnozek*(x, y, z), ki ga beremo "zmnožek x in y je z ".

Formule, ki niso Hornove

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

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

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

Primer

Ali iz

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

3. sodo(zero)

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

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

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

Logično programiranje

V logičnem programiraju je program podan s

- seznamom **pravil** H_1, \dots, H_i in
- **poizvedbo** G

Pravila so Hornove formule. Poizvedba je formula oblike

$$\exists y_1, \dots, y_j . p(y_1, \dots, y_j)$$

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

1. V seznamu H_1, \dots, H_i poišči prvo formulo, ki je oblike

$$\forall x_1, \dots, x_i . \Phi_1(x_1, \dots, x_i) \wedge \dots \wedge \Phi_r(x_1, \dots, x_i) \Rightarrow \Psi(x_1, \dots, x_i),$$

katere sklep Ψ je **zdržljiv** z G . To pomeni da lahko za y_1, \dots, y_j vstavimo neke vrednosti u_1, \dots, u_j in za x_1, \dots, x_i neke vrednosti v_1, \dots, v_i , da sta formuli

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

in

$$\Psi(v_1, \dots, v_i)$$

enaki.

Opomba: možno je, da izbira vrednosti u_1, \dots, u_j in v_1, \dots, v_i ni enolična. V tem primeru izberemo *najbolj splošne vrednosti*. To naredimo s postopkom združevanja (angl. unification), ki smo ga spoznali pri parametričnem polimorfizmu.

2. Poizvedbo smo predelali na poizvedbe $\Phi_1(v_1, \dots, v_i), \dots, \Phi_r(v_1, \dots, v_i)$, 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 for

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

in poizvedbo

$$\exists z . \text{liho}(z)$$

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

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

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

prevedemo na enačbo

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

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

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

Ali lahko to združimo s prvo formulo? Poskusimo rešiti

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

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

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

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

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

Naloga

Če v prejšnjem primeru zamenjamo vrstni red pravil,

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

potem poizvedba

$$\exists z . \text{liho}(z)$$

privede do neskončne zanke (ker iščemo v globino in vedno uporabimo prvo pravilo, ki deluje). Namreč, z uporabo prvega pravila dobimo poizvedbo

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

nato z uporabo drugega pravila

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

nato z uporabo prvega pravila

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

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

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 $\forall x \dots$ in $\exists x \dots$ pišemo **spremenljivke 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 datoeko spravimo pravila (pri čemer pravila za `sodo` zložimo skupaj, da se ne pritožuje):

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

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 oddgovor, 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.

Seznami

Predstavitev seznamov v Prologu

Kako bi v Prologu naredili sezname? V SML smo spoznali, da jih lahko definiramo sami:

```
datatype '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`.

Relacija elem

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

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).
```

Poiskusimo:

```
?- 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 sezname 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)))))) .

```

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 SML:

```

fun join nil y = y
| join (cons(a, x)) y =
  let val z = join x y
  in cons (a, z)
end

```

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:

- $[e_1, e_2, \dots, e_i]$ je seznam elementov e_1, e_2, \dots, e_i .
- $[e | \ell]$ je seznam z glavo e in repom ℓ
- $[e_1, e_2, \dots, e_i | \ell]$ je seznam, ki se začne z elementi e_1, e_2, \dots, e_i in ima rep ℓ .

Za delo s seznami je na voljo [knjižnica `<code>lists</code>`](#), ki jo naložimo z ukazom

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

Ta že vsebuje relacije `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 podeznama:

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

Enakost in neenakost

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

Zanimiv primer

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

Programiranje z omejitvami

To se bomo učili naslednjič.

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

Dokazit C : ✓

v posledu pridete dve premissi: 1. in 2.

• poslussimo 1. premisso $X \wedge Y \Rightarrow C$: ✗

• dokazi X ✗

• dokazi Y

• poslussimo 2. premisso $A \wedge B \Rightarrow C$: ✓

• dokazi A : velja po 5 ✓

• dokazi B :

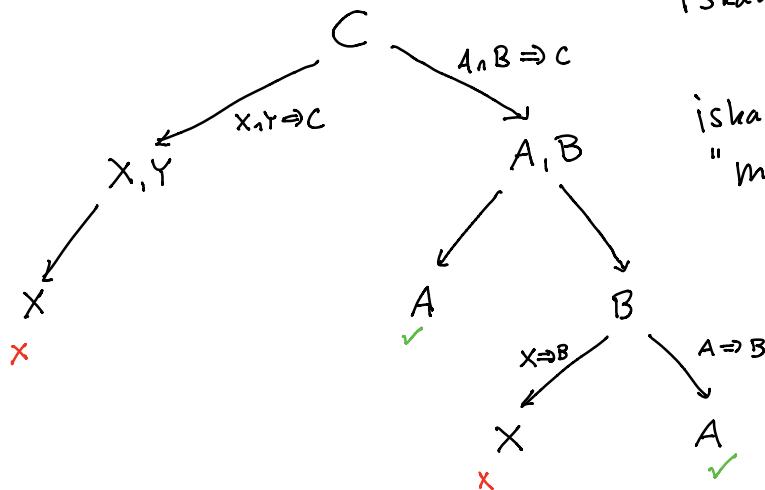
poslussimo 3. $X \Rightarrow B$:

• dokazi X ✗

poslussimo 4. $A \Rightarrow B$:

• dokazi A : velja po 5 ✓

Z dravsom

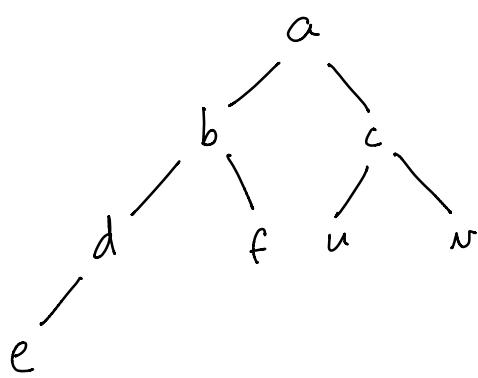


Iškanje dokaza

je

iskanje v dravsu
"možnosti"

Iškanje v globino in Širino



V globino iz levega desno:

a, b, d, e, f, c, u, n

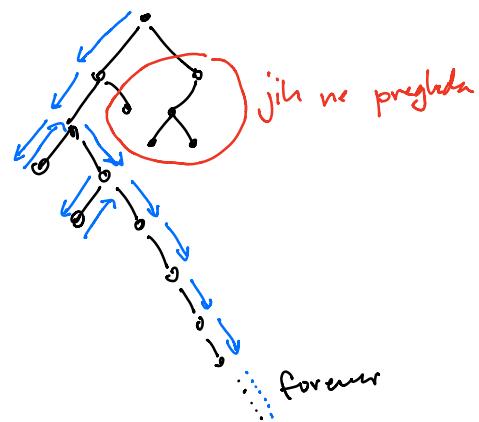
V Širino:

a, b, c, d, f, u, v, e

Urata

a
b, c
c, d, f
d, f, u, v
f, u, v, e
u, v, e
v, e
e

Če je drevo neshomino, se lahko iskanje v globino ne ustavi:



Prolog išče v globino!

Spesifikasi se : ukuran jarak
skip, if then else, while, $x := e$

Operasi ke semantika

$$\eta \text{ okuje } \eta = [(x, 2), (y, 3), (z, 1)]$$

1. $(\eta, c) \mapsto \eta'$ relasi
2. $(\eta, c) \mapsto (\eta', c')$

Logično programiranje z omejitvami

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 <code>is</code>](#) 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,  
    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/2p25px951n_ht9607h1  
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 vredostjo. 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.

Logično programiranje z omejitvami

V logičnem programiraju 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.

V SWI prologu nam to omogoča knjižnica `clpfd` (constraint logic programming on finite domains):

```
?- use_module(library(clpf)).  
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.

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.

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 $L \text{ ins } A..B$ pomeni, da mora veljati $A \leq X \leq B$ za vse elemente $X \in L$ seznama L.

Kombinatorne in globalne omejitve

Omejitev all_distinct([X_1, \dots, X_i]) zagotovi, da imajo spremenljivke X_1, \dots, X_i različne vrednosti. Primer uporabe bomo videli kasneje.

Ostale kombinatorne in globalne omejitve so opisane [v priročniku za SWI prolog](#).

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 $X, Y \in \{1, 2, 3, 4\}$ 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:

- $X \in \{1, 2, 3\}$
- $X \leq Y - 1$
- $Y \in \{2, 3, 4\}$

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 .
```

Pogljemo si primere, s katerimi bomo nabolje spoznali, kako deluje programiranje z omejitvami.

Primer: faktoriela

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: pitagorejske trojice

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,  
    !.
```

Primer: permutacije

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 sesname P, ki zadoščajo navedenim trditvam.

Poskusimo:

```
?- permutacie(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",

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

in preverimo, ali dana seznam je permutacija:

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

Primer: Sudoku

V datoteki [sudoku.pl](#) je program, ki z omejitvami reši Sudoku. Skupaj ga poskusimo razumeti.

Načrt:

1. Imamo seznam vrstic Rows.
2. Imamo matriko 9×9 :
 - imamo 9 vrstic: `length(Rows, 9)`.
 - vsaka vrstica ima dolžino 9: vsak element Rows je seznam dolžine 9.
3. Vsi elementi matrike so med 1 in 9: vsak element V iz Rows, za vsak element X iz V, velja `X in 1..9`.

4. Vsaka vrstica je permutacija: vsak element V seznama Rows zadošča pogoju `all_distinct(V)`.
5. Vsak stolpec je permutacija.
6. Vsak podkvadrat 3×3 je permutacija.

Vsakega od zgornjih omejitev zapišemo v prologu.

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 elemtov 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(Row) :- Row ins 1..9.

maplist(vsi_med_1_9, Rows).
```

Z uporabo `append`:

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

Vsaka vrstica je permutacija

```
map_list(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).
```

Računski učinki

Kaj so računski učinki

Pojem *računskega učinka* je zelo splošen in ga je težko natančno opredeliti. Računski učinki so vsi pojavi v programu, ki spremenjajo ali kako drugače delujejo z zunanjim okoljem programa. Pod pojmom "zunanje okolje" imamo v mislih operacijski sistem in strojno opremo, na kateri teče program.

Primeri računskih učinkov so I/O (pisanje in branje na datoteke), spremjanje in branje pomnilnika, prekinitve izvajanja z izjemo, naključna in verjetnostna izbira, ter še mnogi drugi.

S stališča principov programskih jezikov se je smiselno vprašati, ali imajo vsi ti pojavi kaj skupnega. Jih lahko sistematicno obravnavamo? Ali lahko v programske jezik vgradimo koncepte, ki nam omogočajo nadzorovanje in fleksibilno delo z računskimi učinki?

Da bomo lahko odgovorili na ta vprašanja, najprej spoznajmo nekaj vrst računskih učinkov. Poskusimo ugotoviti, ali imajo kakšno skupno strukturo.

Primeri računskih učinkov

Izjeme

Izjeme pozna večina popularnih programskih jezikov. Izjema je računski učinek, ki prekine izvajanje programa in namesto rezultata vrne posebno vrednost, ki se imenuje *izjema*. Izjem je lahko več ("deljenje z nič", "neveljaven indeks", "datoteka ne obstaja" ipd.), pravzaprav jih je lahko za cel podatkovni tip `exception`.

Program, ki uporablja izjeme lahko

1. vrne rezultat, *ali*
2. sproži izjemo

Če del programa sproži izjemo, se izvajanje prekine takoj in nadaljni izračuni se *ne* izvajajo.

Vhod & izhod

Vhod/izhod (angl. input/output ali kar I/O) je računski učinek, kjer program komunicira z zunanjim okoljem tako, da bere podatke iz vhoda (tipkovnica, datoteka, komunikacijski kanal) in jih izpisuje na izhod (zaslon, datoteka, komunikacijski kanal).

Program, ki uporablja vhod in izhod

1. vrne rezultat, *in*

2. med izvajanjem *izpisuje* in *bere* znake
3. vrne rezultat

Stanje

Stanje je podatek, ki ga lahko program bere in spreminja. To je običajno pomnilniška lokacija, spremenljivka, ali element tabele. Program ima lahko kombinirano stanje iz velikega števila posameznih stanj (na primer, velika tabela ali pomnilnik), a lahko na tako stanje še vedno gledamo kot na enovito stanje, ki pa ima ogromno število možnih vrednosti.

Program, ki uporablja stanje:

1. se začne izvajati v *začetnem stanju*;
2. med izvajanjem *bere* in *nastavlja* stanje;
3. vrne rezultat *in*
4. se konča v *končnem stanju*.

Monade

Na prvi pogled učinki nimajo prav dosti skupnega, podrobnejša analiza pa pokaže, da imajo skupno strukturo, ki jo lahko izrazimo z matematičnim pojmom **monada**.

Najprej moramo ločiti programe, ki *ne* uporabljajo učinkov od tistih, ki jih uporabljajo. Pravimo, da je program, ki ne uporablja učinkov **čist** ali da je **vrednost** (angl. value). Program, ki uporablja učinke, imenujemo **učinkoven** (angl. effectful) ali **izračun** (angl. computation).

Na primer,

```
let x = 2 + 3 in
let y = x + 2 in
y * y - x
```

je čist, saj samo izračuna vrednost. Program

```
let x = 2 + 3 in
print "hello" ;
let y = x + 2 in
y * y - x
```

je izračun, ki uporablja učinek izhod (angl. output).

Veljata naslednji zelo splošni lastnosti:

1. Vsaka vrednost je tudi izračun.
2. Izračune lahko *setavljamo*.

Prva lastnost pravi, da lahko an vrednost gledamo kot na izračun, ki pač ni uporabil nobenega učinka.

Druga lastnost nam omogoča, da lahko izračun sestavimo iz bolj prepostih delov. Operacijo, ki sestavi skupaj dva izračuna, imenujemo bind, ker je neke vrste "vez" med izračuni. Če je p izračun, ki uporablja učinke in vrne rezultat, q pa je izračun, ki pričakuje rezultat od pje sestavljeni izračun

```
bind p q
```

V Haskellu namesto bind uporabimo operator $>>=$:

```
p >>= q
```

Mislimo si: "rezultat, ki ga izračna p pošljemo v izračun q , njune učinke pa sestavimo skupaj."

Primer: izjeme

Denimo, da lahko izračuni uporabljo izjemo Abort in nobenega drugega učinka. Torej je rezultat računa bodisi izjema Abort bodisi rezultat Result v , kjer je v neka vrednost. Konstruktor Result je pretvori vrednost v v izračun.

Tedaj bi lahko bind definirali takole:

```
bind Abort q = Abort
bind (Result v) q = q v
```

Prva vrstica pove, da je treba izvajanje prekiniti, če prvi izračun sproži izjemo. Druga vrstica pove, da nadaljujemo z drugim izračunom, če prvi vrne rezultat. Pri tem rezultat podamo drugemu izračunu.

V Haskellu bi zapisali:

```
Abort >>= q = Abort
(Result v) >>= q = q v
```

Primer: stanje

Na izračun p , ki vrne rezultat tipa τ in hkrati uporablja stanje tipa σ lahko gledamo kot na funkcijo tipa

$$\sigma \rightarrow \tau \times \sigma$$

Res, p sprejme začetno stanje in vrne rezultat ter končno stanje. Sam po sebi p ne naredi ničesar, ker je funkcija. Šele ko ga uporabimo na začetnem stanju, dobimo rezultat in končno stanje.

Če je v vrednost tipa τ , ki ne uporablja stanja, jo lahko pretvorimo v funkcijo, ki stanje sprejme in ga nespremenjenega vrne:

$$\lambda s . (v, s)$$

Kako pa sestavimo dva izračuna, ki uporabljata stanje?

$$p >>= q = \lambda s_1 . \text{let } (v, s_2) = p s_1 \text{ in } (q v) s_2$$

Premislimo: sestavljeni izračun $p \gg= q$ je funkcija:

1. sprijemimo začetno stanje s_1
2. izvede izračun p v začetnem stanju s_1 in dobi rezultat v ter stanje s_2
3. z vrednostjo v nadaljuje izračun q v začetnem stanju s_2 .

Definicija monade

Monada opisuje eno zvrst računskega učinka in sestoji iz naslednjih delov:

1. *Operacija M iz tipov v tipe.* Za vsak tip τ si mislimo, da je $M \tau$ tip izračunov, ki (morda) uporablja računski učinek in vrnejo rezultat tipa τ .
2. *Funkcija return : $\tau \rightarrow M \tau$,* ki pretvori vrednost tipa τ v izračun (ki ne uporablja nobenih učinkov).
3. *Funkcija bind : $M \rho \rightarrow (\rho \rightarrow M \tau) \rightarrow M \tau$,* ki sestavi izračun tipa $M \rho$ in izračun tipa $M \tau$, pri čemer le ta pričakuje kot vhodni podatek vrednost tipa ρ .

Veljati morajo naslednje enačbe:

1. $\text{return } v \equiv v$
2. $\text{return } p \equiv p$
3. $\text{bind } p \text{ (} \lambda v . \text{ bind } (q v) r \text{)} \equiv \text{bind } (\text{bind } p q) r$

V Haskellovi notaciji:

1. $(\text{return } v) \gg= f \equiv f v$
2. $p \gg= \text{return } \equiv p$
3. $p \gg= (\lambda x . q x \gg= q) \equiv (p \gg= q) \gg= r$

Premisli, ali te enačbe veljajo za izjeme in učinke, kot smo jih definirali zgoraj.

Izjeme kot monada

Poglejmo v datoteko [./exception.hs](#).

Blitz Haskell

Vsi programske jeziki nam omogočajo sestavljanje izračunov. V Pythonu in Javi enostavno pišemo ukaze enega za drugim in le ti se sestavijo. V SML uporabimo let, saj

```
let x = p in q
```

najprej požene izračun p (in s tem vse njegove učinke), rezultat shrani v vrednost x , nato pa požene še q , ki ima dostop do x .

V SML, Pythonu in Javi programer ne more nadzorovati monade za sestavljanje izračunov, ker je ta vgrajena v programski jezik. Če torej želimo spoznati programiranje z monadami, moramo uporabiti jezik, ki programerju omogoča, da nadzoruje in sam definira svoje monade - Haskell.

Na vajah boste spoznali osnove Haskella. Ker že poznamo SML bo prehod precej enostaven, seveda pa se bo treba spet navaditi na novo sintakso. SML in Haskell se razlikujeta v dveh delih:

1. SML je *neučakan* in Haskell *len* programski jezik. To pomeni, da SML vedno izračuna vsako vrednost in izračun takoj, Haskell pa šele, ko jo potrebuje.
2. SML ima eno vgrajeno monado za vse učinke, ki jih podpira (reference, izjeme, I/O, kontinuacije). Haskell je *čist*, programer pa samo definira svojo monado.

Kdor je neučakan, lahko sam [bere učbenik o Haskellu](#) in Haskell preizkusim kar [v brskalniku](#).

Računski učinki kot monade v Haskellu

Monado v Haskellu definiramo točno tako, kot smo jo definirali zgoraj: podamo operacijo m , ki slika tipe v tipe ter funkciji `return` in `>>=`.

Haskell ima posebno notacijo, ki omogoča programerju bolj prijazen zapis. Namesto

```
p1 >>= (λ x . p2 >>= (λ y . p3 >>= ...))
```

pišemo

```
do x <- p1
   y <- p2
   ...
```

Namesto

```
do x <- p1
   _ <- p2
   ...
```

pišemo

```
do x <- p1
   p2
   ...
```

V Haskellu moramo pravilno zamikati kodo, podobno kot v Pythonu.

Poglejmo si, kako bi standardne učinke implementirali v Haskellu.

Še nekateri drugi učinki

Naslednjič bomo spoznali dosti bolj zanimive učinke, ki jih lahko uporabimo za učinkovito programiranje.

Računski učinki

Kaj so računski učinki?

~~Računalnik:~~

Program

1. Računa

2. Vpliva na zunajne okolje } operacijski sistem
in okolje vpliva na nj } strojna oprema

Čisto računanje : samo računa brez interakcije z okoljem

Računski učinki : računa in morda še interakcija z okoljem

I/O

Vhod : bere tipkovnico, datoteko, ...
izhod : izpisuje
piše v datoteko, ...

Izjema

poseben dogodek, ki
prekine običajno
izvajanje

Stanje

branje in pisanje
v pomnilnik

Verjetnostno računanje

Vhod / izhod

Program :

1. izpisuje (print)
2. bere (read)
3. na koncu vrne rezultat
in izpis glede na prebrane podatke

Stanje

Program

(spremenljivke,
atributi v objektih,
tabele)

0. zaine se v začetnem stanju
1. bere stanje ($x+3$)
2. spreminja stanje ($x := 3$)
3. vrne rezultat in
konča v končnem stanju

Izjeme

Program

1. Izračuna rezultat ali
2. sproži izjemo

Pravinmo, da je:

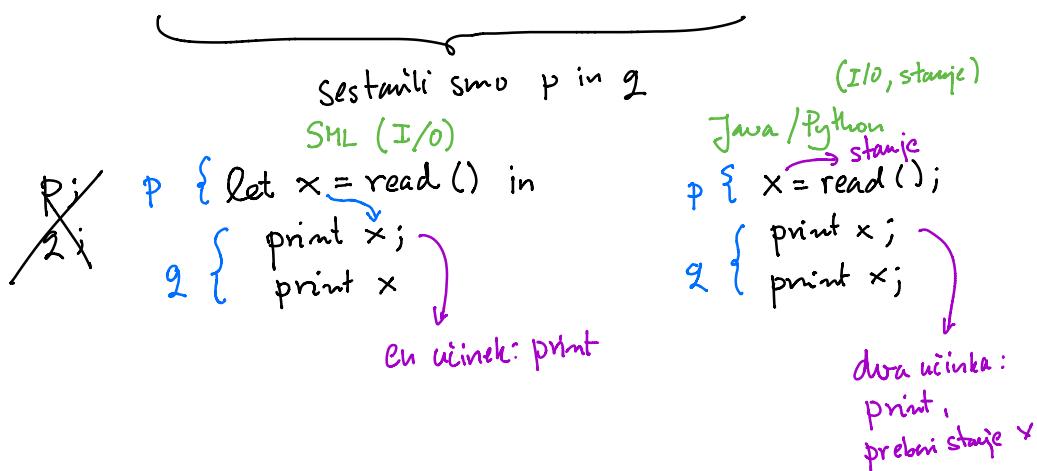
- program vrednost (value), če je čist (pure), t.e. ne uporablja računskih učinkov
- program izračun (computation), če je učinkoven (effectful), t.j. lahko uporablja računske učinke

Izračune lahko sestavljamo:

```

x = 3
print(x+2)
y = x - 7
return(x+y)
  
```

V splošnem: izračun p, ki ima učinke in rezultat,
ta rezultat uporabi izračun q

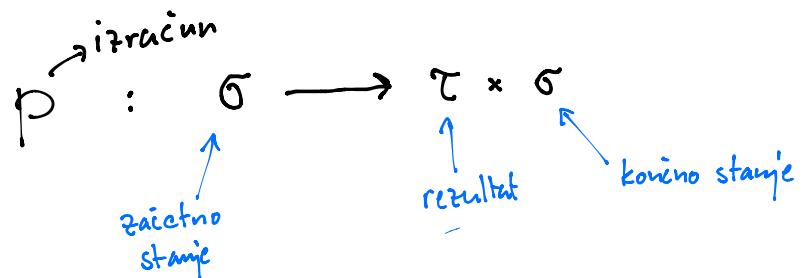


Monada

1. Kako vrednost predelamo v izračun (return)
2. Kako sestavljamo izračune (bind, $\gg=$)

Stanje

Izračun, ki vrne rezultat tipa τ in uporablja stanje tipa σ .



1. Če je n vrednost, jo predelamo v izračun kot:

$$s \mapsto (n, s) \quad \leftarrow \begin{array}{l} \text{izračun, ki ga dobimo ko} \\ \text{čisto vrednost pretvorimo} \\ \text{v izračun} \end{array}$$

$n \leftarrow \text{čista vrednost}$

2. Kako sestavimo

$$P : \sigma \xrightarrow{\text{začetno stanje}} \tau * \sigma \xrightarrow{\text{vmesni rezultat}} \text{vmesno stanje}$$
$$Q : \tau \xrightarrow{\text{začetno stanje}} (\sigma \rightarrow \sigma * \sigma) \xrightarrow{\text{končno stanje}} \text{končni rezultat}$$

Diagram illustrating the construction of a monad from a comonad Q and a function P :

- $P : \sigma \rightarrow \tau * \sigma$ is the computation function.
- $Q : \tau \rightarrow (\sigma \rightarrow \sigma * \sigma)$ is the comonad function.
- Annotations show the flow of data:
 - From σ to $\tau * \sigma$ via P (labeled "začetno stanje" and "končno stanje").
 - From τ to $(\sigma \rightarrow \sigma * \sigma)$ via Q (labeled "začetno stanje" and "končno stanje").
 - From $\tau * \sigma$ to $(\sigma \rightarrow \sigma * \sigma)$ via the multiplication operation (labeled "vmesni rezultat" and "končni rezultat").

Programiranje z monadami

V prejšnji lekciji smo ugotovili, da lahko z monadami predstavimo računske učinke. Tokrat si bomo ogledali, kako lahko monade uporabimo kot programerji. Razne programerske tehnike lahko namreč izrazimo z monadami.

Še prej pa ponovimo, kaj je monada.

Definicija monade

Sestavni deli **monade** so:

1. *Operacija M iz tipov v tipe.* Za vsak tip τ si mislimo, da je $M \tau$ tip *izačunov*, ki (morda) uporabljo računski učinek in vrnejo rezultat tipa τ .
2. *Funkcija return : $\tau \rightarrow M \tau$,* ki pretvori vrednost tipa τ v izračun (ki ne uporablja nobenih učinkov).
3. *Funkcija bind : $M \rho \rightarrow (\rho \rightarrow M \tau) \rightarrow M \tau$,* ki sestavi izračun tipa $M \rho$ in izračun tipa $M \tau$, pri čemer le ta pričakuje kot vhodni podatek vrednost tipa ρ .

Veljati morajo naslednje enačbe:

1. $\text{return } v \equiv v$
2. $\text{return } p \equiv p$
3. $\text{bind } p \text{ (} \lambda v . \text{ bind } (q v) r \text{)} \equiv \text{bind } (p q) r$

V Haskellovi notaciji:

1. $(\text{return } v) >= f \equiv f v$
2. $p >= \text{return } \equiv p$
3. $p >= (\lambda x . q x >= q) \equiv (p >= q) >= r$

Monade v Haskellu

Programski jezik Haskell je *čist*, s čimer želimo povedati, da je treba vse računske učinke narediti z monadami. (Tudi v ostalih programskeh jezikih se v ozadju skriva monada računskih učinkov, a je ne opazimo, ker ni izražena s pomočjo tipov.)

Monade so v Haskellu **razred tipov**, zato najprej povejmo nekaj o tem konceptu.

Razredi tipov

Razredi tipov (angl. type classes) so mehanizem, s katerim lahko organiziramo večje programske enote. Namenjeni so podobnemu namenu kot razredi v objektnem programiranju in strukture v SML, a se od obojih razlikujejo.

Razred tipov C definiramo takole (poenostavljena različica):

```
class C a where
    v1 :: τ1(a)
    ...
    vi :: τi(a)
```

Simbol a označuje poljuben tip. Zgornja definicija je neke vrste vmesnik ali signatura, ki predpisuje, da je razred tipov C podan s tipom σ in vrednostmi v₁, ..., v_i tipov τ₁(σ), ..., τ_i(σ).

Primerek ali **instanca** (angl. instance) razreda tipov C je definiran z

```
instance C σ where
    v1 = ...
    ...
    vi = ...
```

Najbolje bo, da si ogledamo primer.

Razred tipov Num

Vsek programskega jezika ima **numerične tipe**. To so tisti tipi, katerih vrednosti lahko seštevamo, odštevamo, množicmo itd. V Haskellu je v ta namen definiran razred tipov [Num](#).

Če v Haskellu definiramo funkcijo, ki uporablja aritmetične operacije, kakšen je njen tip?

```
λ> let f x y = 2 * x * x + 3 * y + 7
λ> :t f
f :: Num a => a -> a -> a
```

Haskell odgovori, da ima f tip a -> a -> a, kjer je a poljuben tip, ki mora biti iz razreda Num. To je oblika *ad-hoc polimorfizma*: f ima več tipov (polimorfizem), vendar se ne obnaša enakomerno za vse tipe, ampak je njen obnašanje pri tipu a določeno s tem, kako so definirane aritmetične operacije na a (ad-hoc obnašanje).

Programer lahko dodaja svoje numerične tipe. Dodajmo kompleksna števila:

```
data Complex = C { re :: Float, im :: Float }
               deriving (Show, Eq)
```

Zgornja definicija pove, da kompleksno število $x + y \cdot i$ zapišemo $C x y$. Če je z kompleksno število, dobimo njegovi komponenti z `re z` in `im z`. Določilo `deriving` je navodilu Haskellu, naj za `Complex` sam generira še instance za razreda tipov `Show` in `Eq`.

Podatkovni tip `Complex` opremimo s strukturo numeričnega tipa:

```
instance Num Complex where
    z + w = C (re z + re w) (im z + im w)
    z * w = C (re z * re w - im w * im w) (re z * im w + im z * re w)
    abs z = C (sqrt (re z * re z + im z * im z)) 0
    signum z = let r = sqrt (re z * re z + im z * im z) in C (re z / r)
               (im z / r)
    fromInteger k = C (fromInteger k) 0
    negate z = C (negate (re z)) (negate (im z))
```

Sedaj lahko računamo s kompleksnimi števili in jih uporabljamo povsod, kjer Haskell pričakuje numerični tip:

```
λ> C 1 2 + C 3 4
C {re = 4.0, im = 6.0}
λ> C 1 2 * C 3 4
C {re = -13.0, im = 10.0}
λ> C 0 1 * C 0 1
C {re = -1.0, im = 0.0}
λ> abs (C 1 1)
C {re = 1.4142135, im = 0.0}
λ> f (C 4 6) (C (-2) 3)
C {re = -156.0, im = -111.0}
```

Razreda tipov Functor, Applicative

Haskellova standard knjižnica zahteva, da mora vsak tip, ki je monada (primerek razreda tipov `Monad`) biti hkrati še `Applicative`. Vsak primerek `Applicative` pa mora biti `Functor`. Te zahteve izhajajo iz matematične definicije monade.

Oglejmo si, kaj sta [Functor](#) in [Applicative](#).

Razred `Functor` zahteva, da opremimo tip s funkcijo `fmap`, ki se obnaša tako kot `map` na seznamih. Torej je tip primerek razreda `Functor`, če je neke vrste "zbirka" elementov, ki jo lahko preslikamo v drugo "zbirko". Dvojiška drevesa so `Functor`:

```
data Tree a = Empty | Node a (Tree a) (Tree a)
             deriving Show

instance Functor Tree where
    fmap f Empty = Empty
    fmap f (Node x l r) = Node (f x) (fmap f l) (fmap f r)
```

Razred `Applicative` opisuje zbirke, pri kateri lahko zbirko funkcij kombiniramo z zbirko argumentov in dobimo zbirko rezultatov.

Monada `Maybe a`

Sedaj se lahko posvetimo nekaterim monadom. Poglejmo si monado za tip `Maybe`. To je monada, ki pove, kako računamo z *opcijskimi vrednostmi*.

Ali vas na kaj spominja?

Monada `[a]`

Seznami tvorijo monado, ker si lahko seznam vrednosti tipa `a` predstavljamo kot **nedeterministični izračun**, ki vrne poljubno število rezultatov. To pride prav, kadar računamo rešitve problema, ki ima lahko nič ali več rešitev, saj jih preprosto zložimo v seznam.

Primer: problem šahovskih dam

Na vajah boste rešili problem šahovskih dam na klasičen način in "na roke" izračunali boste seznam vseh rešitev. Na predavanjih poglejmo, kako lahko isti problem rešimo z monado.

Najprej poglejmo primer uporabe. Radi izračunali vse možne vsote, ko vržemo dve kocki:

1. Brez monade: `[x + y | x <- [1..6], y <- [1..6]]`
2. Z monado:

```
do x <- [1..6]
   y <- [1..6]
   return (x + y)
```

Še en primer: funkcija, reši kvadratno enačbo.

Iskanje v globino

Nazadnje si oglejmo monado `Search` za splošno iskanje rešitve problema.

Predstavljammo si, da poskušamo izračunati rezultat tipa `a`, pri čemer moramo v postopku računanja izbirati med možnostmi tipa `r`. Nekatere možnosti uspešno vodijo do rezultatov, druge pa ne. Radi bi uspešno izračunali rezultat in zaporedje možnosti, ki je pripeljalo do njega. (Še bolje, radi bi seznam vseh rezultatov in zaporedij množnosti, ki so pripeljali do njega.)

Na primer, v labirintu iščemo lonec zlata (tip `a` je "zlato"), na vsakem razpotju pa se moramo odločiti, kam bomo zavili (tip `r` je "smer").

To klasično naloge bi lahko rešili s *sestopanjem*: vsakič, ko imamo na voljo več možnosti, po vrsti preizkusimo vsako (sestopimo). Če možnost ne deluje, preizkusimo naslednjo.