Ukazni programski jezik `comm`

Spoznali smo aritmetične izraze s spremenljivkami. Spremenljivke smo obravnavali po mačehovsko, saj spremenljivkam ne moremo nastavljati vrednosti, prav tako pa ne moremo ustvariti novih spremenljivk.

V tej lekciji bomo spoznali ukazni programski jezik, ki ima prave spremnljivke, pogojne stavke in zanko `while`.

Po vrsti bomo obravnvali:

- sintakso jezika
- operacijsko semantiko: kako se jezik izvaja
- denotacijsko semantiko: kaj je matematični pomen jezika
- prevajalnik v poenostavljeno strojno kodo

Sintaksa

V prejšnji lekciji smo spoznali aritmetične izraze. Dodali bomo še boolove izraze in ukaze.

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⟩ | ( ⟨aritmetični-izraz⟩ )

⟨spremenljivka⟩ ::= [a-zA-z]+

⟨številka⟩ ::= -? [0-9]+
```

Boolovi izrazi:

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

Ukazi:

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

Slovnica za aritmetične izraze je enaka kot v prejšnji lekciji in je spisana tako, da vključuje asociativnosti in prioriteto operatorjev. Na primer, aritmetični izraz `x + y + z` lahko glede na pravila slovnice razčlenimo samo na en način, kot drevo

```
    +
   / \
```

```
  +   z
 / \
x   y
```

Podobno pravila določajo da je `x + y * z` enak kot `x + (y * z)` in ne `(x + y) * z`.

Pravila za Boolove izraze niso tako natančna. Na primer `b and c or d` lahko s pravili za boolove izraze računimo kot `(b and c) or d` ali kot `b and (c or d)`. Dvoumnost pravil lahko odstranimo tako, da navedemo prioriteto in asociativnost operacij. Naštejmo jih od nižje do višje prioritete:

- `;` (levo)
- `or` (levo)
- `and` (levo)
- `not`
- `+` (levo)
- `*` (levo)

Na primer, `or` je levo asociativen in ima prednost pred `;`.

Primer

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

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

Konkretna sintaksa jezika se morda zdi nekoliko nenavadna, a to je do neke mere nebistveno za teorijo programskih jezikov, saj je pomembnejša abstraktna sintaksa. Na primer, zgornji program bi lahko zapisali v Javi takole:

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

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

To *ne* pomeni, da je konkretna sintaksa nepomembna v praksi; navsezadnje so se pripravljeni programerji skregati že zaradi zamikanja kode. V zvezi s tem omenimo Wadlerjev zakon.

Operacijska semantika

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

$$\eta = [x \mapsto 4, y \mapsto 10, u \mapsto 1]$$

Ker bomo spremenljivkam tudi nastavljali vrednosti, potrebujemo ustrezno operacijo. Če je η okolje, x spremenljivka in n celo število, potem zapis

$$\eta\,[x \mapsto n]$$

pomeni okolje η, v katerem je vrednost `x` nastavljena na `n`. Primer: če je η = `[x ↦ 10, y ↦ 5]`, potem je η`[x↦20]` enako `[x ↦ 20, y ↦ 5]`.

Semantika malih korakov

Operacijska semantika aritmetičnih in boolovih izrazov

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

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

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

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

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

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

Tudi Boolovi izrazi ne predstavljajo večje težave:

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

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

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

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

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

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

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

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

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

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

Vaja: dodaj pravila za =

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

1. Izračuamo b_1 *in* b_2 in nato vrednost b_1 `and` b_2
2. Najprej izračunamo b_1. Če dobimo `false`, je vrednnost b_1 `and` b_2 enaka `false` ne glede na b_2, zato ga ne izračunamo.

Zgoraj smo uporabili drugo možnost (vaja: kako se to vidi v pravilih?)

Operacijska semantika ukazov

Semantika malih korakov je podana z dvema relacijama:

- relacija $(\eta, c) \mapsto \eta'$ pomeni: v okolju η ukaz c v enem koraku konča v okolju η'.
- relacija $(\eta, c) \mapsto (\eta', c')$: v okolju η ukaz c c enem koraku spremeni okolje v η' in se nadaljuje z ukazom c'.

Relaciji sta določeni z naslednjimi pravili:

$$\frac{}{(\eta, \texttt{skip}) \mapsto \eta}$$

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

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

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

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

$$\frac{}{(\eta, (\texttt{if b then } c_1 \texttt{ else } c_2 \texttt{ end})) \mapsto (\eta, c_1)}$$

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

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

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

Denotacijska semantika

Matematični pomen programa je preslikava, ki sprejme okolje in vrne okolje. O tem ne bomo veliko govorili, povejmo le, da se matematični pomen kode `e` ponavadi zapiše kot ⟦ `e` ⟧ .

Na primer, matematični pomen aritmetičnega izraza je celo število. Ko zapišemo

```
⟦ 42 ⟧ = 42
```

s tem povemo, da je pomen *niz znakov* `42` število, ki mu po slovensko rečemo “dvainštirideset”. Podobno podamo pomen znaka + z enačbo:

```
⟦ e1 + e2 ⟧ = ⟦ e1 ⟧ + ⟦ e2 ⟧
```

To pomeni, da je pomen *izraza* oblike `e1 + e2` (plus kot znak UTF8) enak *vsoti* (plus kot matematična operacija) pomenov podizrazov `e1` in `e2`.

Pomen programa `e` v ukaznem jeziku funkcija, ki slika okolja v okolja:

```
⟦ c ⟧ : Env → Env
```

Tu je `Env` množica vseh okolij. Pomen definiramo takole, kjer smo predpostavili, da imamo že na voljo funkcijo `eval`, ki izračuna matematični pomen aritmetičnih in boolovih izrazov:

```
⟦ skip ⟧ (η) := η
⟦ x := e ⟧ (η) := η ⟦x ↦ eval(η,e)⟧
⟦ c1 ; c2 ⟧ (η) := ⟦c2⟧ ( ⟦c1⟧ (η))
⟦ if b then c1 else c2⟧ (η) := ⟦c1⟧ (η) če eval(η,b) = true
⟦ if b then c1 else c2⟧ (η) := ⟦c2⟧ (η) če eval(η,b) = false
⟦ while b do c ⟧ (η) := ?
```

Pomen zanke `while` je še pod vprašajem in se ga na tem mestu ne bomo lotili.

Ekvivalenca programov

Programa sta ekvivalenta, če se v vseh kontekstih obnašata enako. To pomeni, da lahko vedno enega zamenjamo z drugim.

Natančneje, **evaluacijski kontekst** `C[ ]` je del programske kode `C`, ki ima "luknjo" `[ ]`. Programska koda `A` je **ekvivalentna** programski kodi `B`, če za *vse* kontekste `C[ ]` velja, da imata `C[A]` in `C[B]` enak rezultata in enako spreminjata okolje.

Primer

Programa

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

in

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

sta ekvivalentna.

Primer

Programa

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

in

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

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

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

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

Naloga

Ali je program, ki sešteje prvih 100 števil

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

ekvivalentne programu

```
s := 5050
```

Odgovor: nista ekvivalentna, ker drugi program ne nastavi vrednosti `i`. Ekvivalentno bi bilo:

```
i := 101 ;
s := 5050
```

Prevajalnik

Implementiramo prevajalnik v strojno kodo. Ogledali si bomo implementacijo v OCamlu, glej programski jezik comm v Programming Languages Zoo.

Implementirana je razširjena različica jezika, ki ima še:

- ukaz `print`, s katerim lahko izpišemo celo število
- ukaz `new x := e in c`, ki uvede novo lokalno spremenljivko `x` v ukazu `c`.

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` c_1 `else` c_2 `done`
- zanka `while b do c done`
- ukaz `skip`
- sestavljani ukaz c_1 `;` c_2
- ukaz `print e`

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 programskih 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.