LOGO, SEZNAMI IN MNOŽICE (Presek 26 (1998-1999) št. 3)

Želvja grafika je gotovo najbolj znan in najbolj priljubljen del programskega jezika logo. Uporaba grafike ni zapletena, poznati moramo le nekaj ukazov, prav pa pridejo tudi pomožne spremenljivke in seveda odločitveni stavki ter zanke. Zelo zanimive risbe lahko narišemo tudi z uporabo rekurzije. Nekoliko manj znan in tudi nekoliko zahtevnejši del loga pa so seznami. Ti sodijo v tisti del loga, ki se močno navezuje na programski jezik lisp (**lis**t **p**rocessing; list je angleška beseda za seznam).

Seznami v logu nastopajo zelo pogosto in so poleg števil in nizov znakov osnovni v logo vgrajeni podatkovni tip. Srečamo jih tudi kot parametre pri vgrajenih ukazih. Tako ima ukaz `REPEAT` dva parametra: prvi je številska vrednost; ta pove, kolikokrat se ponovijo ukazi iz seznama, ki je drugi parameter.

V nadaljevanju bomo spoznali osnovne ukaze za delo s seznami, za vajo pa bomo v narečju MSWlogo (glej `http://vlado.fmf.uni-lj.si/educa/logo`) sprogramirali nekaj ukazov za delo z množicami. Množice bomo predstavili s seznami, v katerih bodo našteti elementi množic. Na primer, množico $\{1, 2\}$ bomo predstavili s seznamom `[1 2]`, morda pa tudi s seznamom `[2 1]`, saj vrstni red, v katerem naštejemo elemente množice, ni pomemben.

Za začetek napišimo ukaz, s katerim bomo za dano število n zgradili množico, ki vsebuje števila od 1 do n. Program bomo zasnovali klasično. Uporabili bomo pomožno spremenljivko. To v logu uvedemo z ukazom `LOCAL`. Njena začetna vrednost bo prazen seznam `[]`. Nato v zanki `REPEAT`, ki jo ponovimo n-krat, *na konec* seznama, ki je shranjen v pomožni spremenljivki, z ukazom `LPUT` ("last put") po vrsti dodamo števila $1, 2, \ldots, n$.

```
TO seznam :n
; Zgradi seznam števil od 1 do n. Varianta z zanko.
  LOCAL "s
  MAKE "s []
  REPEAT :n [MAKE "s LPUT REPCOUNT :s]
  OP :s
END
```

Ukaz `REPCOUNT` vrne število, ki pove, v kateri ponovitvi zanke `REPEAT` smo. Ukaz `LPUT` svoj prvi argument, ta je v našem primeru število, ki ga vrne `REPCOUNT`, postavi na konec seznama, ki je drugi argument, in kot rezultat vrne povečani seznam. S povečanim seznamom je treba v programu nekaj storiti, sicer bo logov tolmač javil napako. Tako ga z ukazom `MAKE` zopet shranimo v spremenljivko `s`.

Kako se obnaša napisani ukaz, lahko poskusimo na primer s `SHOW seznam 7` ali pa s `SHOW seznam 2 * 5`.

Napisani ukaz `seznam` je precej omejen. Dopolnimo ga tako, da bo tudi spodnja meja seznama, ki ga želimo zgraditi, spremenljiva.

```
TO seznam :m :n
; Zgradi seznam števil od m do n. Rekurzivna izvedba.
  IF :m > :n [OP []] ;Baza rekurzije.
  OP FPUT :m (seznam :m + 1 :n)
END
```

Tokrat smo ukaz napisali rekurzivno. Rekurzija je programerski prijem, ki nam velikokrat olajša programiranje in poenostavi rešitev, hkrati pa jo naredi bolj pregledno. Pri rekurzivnem programiranju vedno potrebujemo ustavitveni pogoj (osnovo rekurzije), to je primer, ko ukaz ne pokliče samega sebe, temveč vrne neko znano vrednost ali pa opravi neko določeno opravilo. V našem programu rekurzivni klic ni potreben tedaj, ko je spodnja meja večja od zgornje (to preverimo z ukazom `IF`). Če pa spodnja meja še ni večja od zgornje, seznam števil od m do n zgradimo tako, da rekurzivno (z istim ukazom, le s spremenjenim prvim argumentom) zgradimo seznam števil od $m+1$ do n, v ta seznam pa nato na začetek z ukazom `FPUT` dodamo še število m. Pri vsakem rekurzivnem klicu se tako razlika med drugim in prvim argumentom zmanjša za ena, kar zagotavlja, da se rekurzija res izteče. Ukaz `FPUT` ("first put") deluje enako kot ukaz `LPUT`, le da element doda *na začetek* in ne na konec seznama.

Spomnimo se, da je množica podmnožica druge množice, če je vsak element prve množice tudi element druge množice. Sestavimo ukaz v logu, ki bo ugotovil, ali je prva množica podmnožica druge.

```
TO podmnozica :a :b
; Ali je vsak element iz a tudi v b? Rekurzivna varianta.
  IF EMPTYP :a [OP "TRUE] ;Prazna množica je vedno podmnožica.
  IF NOT MEMBERP (FIRST :a) :b [OP "FALSE] ;Ni podmnožica.
  OP podmnozica (BF :a) :b
END
```

Tudi tokrat smo ukaz napisali rekurzivno. Za konec rekurzije imamo dve možnosti. Prazna množica je podmnožica vsake množice. Ali je seznam prazen, ugotovimo z ukazom `EMPTYP`. Kadar prva množica ni prazna, pogledamo, ali je njen prvi element vsebovan tudi v drugi množici. Do prvega elementa seznama pridemo z ukazom `FIRST`. Ali ta element pripada seznamu, ugotovimo z ukazom `MEMBERP`. Ukazi, ki se končajo s črko `P`, vračajo niz znakov,

ki predstavlja logično vrednost (`TRUE` – resnično ali `FALSE` – lažno). Z negacijo `NOT`, ki jo najpogosteje uporabljamo v logičnih pogojih, spremenimo vrednost `TRUE` v `FALSE` in obratno. Do rekurzivnega klica pride, kadar je prvi element prve množice tudi element druge množice. Tedaj moramo preveriti, ali je prva množica brez prvega elementa podmnožica druge množice. To storimo z rekurzivnim klicem, pri katerem za prvi argument uporabimo prvo množico brez prvega elementa (seznam brez prvega elementa vrne ukaz `BF` – "but first"). Pri vsakem rekurzivnem klicu se tako dolžina prvega argumenta zmanjša za ena, kar zopet zagotavlja, da se rekurzija res konča.

Seznama `[1 2]` in `[2 1]` nista enaka (kot seznama), predstavljata pa isto množico. Za ugotavljanje enakosti množic torej ne moremo uporabiti v logo vgrajenega operatorja `=`, ampak moramo sprogramirati svoj ukaz. To pa ni težko, saj sta množici enaki natanko tedaj, kadar je prva podmnožica druge in hkrati druga podmnožica prve.

```
TO enaki :a :b
; Ali imata seznama enake elemente?
  OP AND podmnozica :a :b podmnozica :b :a
END
```

Preizkusimo ukaze, ki smo jih sprogramirali. Če se nismo zmotili, klic

```
enaki (seznam 10 20) (REVERSE seznam 10 20)
```

vrne vrednost resnično. Pri tem je `REVERSE` ukaz, ki vrne seznam z zamenjanim vrstnim redom elementov. Na primer, klic `REVERSE [a b c]` vrne seznam `[c b a]`. Okrogli oklepaji okoli argumentov ukaza `enaki` sicer niso nujno potrebni, povečajo po čitljivost izraza.

Sedaj pa je že čas, da se lotimo (vsaj na prvi pogled) nekoliko zahtevnejše naloge. Sestavili bomo ukaz, ki izračuna presek dveh množic. Tudi tokrat je osnovna ideja rešitve preprosta. Sprehodimo se po prvi množici in na vsakem koraku za tekoči element pogledamo, ali je tudi v drugi množici ali ne. Če ni, ga le preskočimo, sicer pa ga postavimo v rezultat. Sprehod skozi prvo množico bomo zopet napisali rekurzivno.

```
TO presek :a :b
; Izračuna presek množic a in b. Rekurzivna varianta.
  IF EMPTYP :a [OP :a] ;Baza rekurzije. Konec pregleda prve množice.
  ;Prvi element iz a ni v preseku.
  IF NOT MEMBERP (FIRST :a) :b [OP presek BF :a :b]
  ;Prvi element iz a je v preseku.
  OP FPUT (FIRST :a) (presek BF :a :b)
END
```

Naslednji problem, ki se ga bomo lotili, je izračun kartezičnega produkta para množic. Ta je definiran kot množica vseh urejenih parov, pri katerih je prva komponenta para iz prve, druga pa iz druge množice,

$$A \times B = \{(a, b) \mid a \in A, b \in B\}\,.$$

Domenimo se, da bomo urejeni par (a, b) v logu predstavili kot navadno množico z elementoma a in b, pazili bomo le, da bo a v paru naveden pred b, torej `[a b]`. Napisati torej želimo ukaz, ki bo deloval nekako takole: klic

```
kartezicni [a b c] [1 2]
```

naj vrne

```
[[a 1] [a 2] [b 1] [b 2] [c 1] [c 2]].
```

Rešitev bomo tokrat napisali klasično, z uporabo dveh vgnezdenih zank `REPEAT`. S prvo se sprehodimo skozi prvo množico, pri izbranem elementu x iz prve množice pa v drugi zanki zgradimo vse pare, ki kot prvo komponento vsebujejo element x.

```
TO kartezicni :a :b
; Izračuna kartezični produkt množic a in b. Iterativna varianta.
  LOCAL "s ;Pomožna spremenljivka za kartezični produkt.
  (LOCAL "x "y) ;Komponenti urejenega para.
  MAKE "s []
  REPEAT COUNT :a [
    MAKE "x ITEM REPCOUNT :a
    ;Naredimo vse pare, ki imajo x za prvo komponento.
    REPEAT COUNT :b [
      MAKE "y ITEM REPCOUNT :b
      MAKE "s LPUT (LIST :x :y) :s
    ]
  ]
  OP :s
END
```

V rešitvi smo uporabili nekaj ukazov, ki jih doslej še nismo srečali. Tako ukaz `COUNT` vrne število elementov seznama. Z ukazom `ITEM` dobimo izbrani element seznama; klic `ITEM :i :s` vrne `i`-ti element seznama `s`. Če želimo sestaviti seznam, ki bo kot elementa vseboval vrednosti spremenljivk `x` in `y`, ne smemo zapisati `[:x :y]`. Logov tolmač namreč izrazov znotraj seznamov ne ovrednoti, tako da se `:x` in `:y` ne bosta nadomestila z ustreznima

vrednostma. Uporabiti moramo ukaz `LIST`, da se argumenti najprej ovrednotijo, nato pa ukaz iz njih sestavi seznam. Ukaz ima lahko tudi samo enega ali več kot dva argumenta. Pri taki uporabi moramo celoto obvezno obdati z okroglimi oklepaji.

Če ste pozorno pregledali ukaz `kartezicni`, potem ste gotovo opazili, da pomožne spremenljivke `y` pravzaprav ne potrebujemo. Za vajo lahko tako poskusite napisati varianto ukaza, ki bo uporabljala le eno pomožno spremenljivko. Naj vam namignem, da je treba gornjo različico spremeniti le na treh mestih.

Naš zadnji program bo izračunal potenčno množico dane množice. Spomnimo se, da je potenčna množica množice A sestavljena iz vseh podmnožic množice A,

$$\mathcal{P}(A) = \{B \mid B \subseteq A\}\,.$$

Ukaz v logu naj bi torej deloval nekako takole: klic

```
potencna [a b c]
```

naj vrne

```
[[] [a] [b] [a b] [c] [a c] [b c] [a b c]].
```

Pri programiranju bomo spet uporabili rekurzijo. Ideja je takale. Če je vhodna množica A prazna, potem ima njena potenčna množica en element, prazno množico (to bo osnova rekurzije). Sicer pa vzamemo $a \in A$. Množice v $\mathcal{P}(A)$ so dveh vrst: tiste, ki vsebujejo a, in tiste, ki elementa a ne vsebujejo. Če poiščemo tiste, ki ne vsebujejo a (to so ravno množice iz $\mathcal{P}(A \setminus \{a\})$, te pa lahko dobimo z rekurzivnim klicem), nato pa vsako od njih "podvojimo" (naredimo še eno kopijo, ki ji dodamo a), dobimo ravno potenčno množico množice A.

```
TO potencna :a
; Določi potenčno množico množice a. Rekurzivna varianta.
  IF EMPTYP :a [OP [[]]] ;Potenčna množica prazne množice.
  OP SE (potencna BL :a) (razsiri (LAST :a) (potencna BL :a))
END

TO razsiri :x :s
; Na konec vsakega elementa iz s doda x in vrne seznam razširjenih
; seznamov. Npr. razsiri 1 [[a] [b] [a b]] --> [[a 1] [b 1] [a b 1]].
  IF EMPTYP :s [OP :s]
  OP FPUT (LPUT :x FIRST :s) (razsiri :x BF :s)
END
```

V programu je uporabljenih tudi nekaj novih ukazov. Ukaza `LAST` in `BL` ("but last") delujeta podobno kot ukaza `FIRST` in `BF`, le da učinkujeta na koncu in ne na začetku seznama. Delovanje ukaza `SE` (to je okrajšava za ukaz `SENTENCE`) je nekoliko bolj zapleteno. Če so argumenti seznami, potem ukaz vrne seznam, ki ga dobi z združitvijo argumentov. Argumente, ki niso seznami, ukaz obravnava kot enoelementne sezname. Na primer, ukazi

```
(SE [a] [b] [c]),    SE 1 2    in    (SE [1 2] 3 [] [[4]])
```

vrnejo

```
[a b c],    [1 2]    in    [1 2 3 [4]].
```

Ponovimo, katere ukaze za delo s seznami smo spoznali. Prvi element seznama vrne ukaz `FIRST`, zadnji element seznama pa dobimo z ukazom `LAST`. Seznam brez prvega elementa vrne ukaz `BF`, seznam brez zadnjega elementa pa ukaz `BL`. Če katerega od ukazov `FIRST`, `LAST`, `BF`, `BL` uporabimo na praznem seznamu, pride do napake med izvajanjem. Ali je seznam prazen, ugotovimo z ukazom `EMPTYP`. Sezname povečujemo z ukazoma `FPUT` in `LPUT`. Prvi doda element na začetek, drugi pa na konec seznama. Koliko elementov ima seznam, nam pove ukaz `COUNT`. Z ukazom `MEMBERP` ugotovimo, ali je izbrani objekt element seznama, z ukazom `REVERSE` pa obrnemo vrstni red elementov v seznamu. Do elementa na izbranem mestu pridemo z ukazom `ITEM`. Sezname gradimo z ukazoma `LIST` in `SENTENCE`. Našteti ukazi (razen ukaza `REVERSE`) so standardni in jih poznajo praktično vsa narečja loga. Omenimo še, da večina zgoraj naštetih ukazov smiselno deluje tudi na številih in nizih znakov.

Za konec dodajmo še nekaj nalog za nadaljnje delo:

1. Če prvi ukaz `seznam` pokličemo z negativnim številom, npr. `seznam -3`, pride do napake med izvajanjem. Ukaz `REPEAT` namreč kot prvi parameter zahteva nenegativno število. Spremenite ukaz tako, da bo takrat, ko bo argument negativen, vrnil seznam števil od -1 do vrednosti argumenta. Primer: klic `seznam -3` naj vrne `[-1 -2 -3]`.

2. Omenili smo že, da isto množico lahko predstavimo z različnimi seznami. Zamenjamo lahko vrstni red elementov, pa tudi nekateri elementi se v seznamu lahko ponovijo. Na primer, vsi seznami `[1 2]`, `[2 1]` in `[1 2 2 1]` predstavljajo isto množico (tudi ukaz `enaki`, ki smo ga napisali, jih med seboj ne loči). Sprogramirajte ukaz `poenostavi`, ki vzame seznam in iz njega izloči večkratne pojavitve elementov; npr. `poenostavi [1 2 3 1 2]` naj vrne `[1 2 3]`.

3. S pomočjo ukaza `poenostavi` iz prejšnje naloge in ukaza `COUNT` sestavite ukaz `moc`, ki vrne moč množice, ki jo predstavlja seznam. Na primer, klic `moc [1 2 3 1 2]` mora vrniti `3`.

4. Uporabite ukaz `poenostavi` iz druge naloge ter ukaz `SENTENCE` in sestavite ukaz `unija`, ki poišče unijo dveh množic. Pri tem naj bo unija predstavljena s seznamom, v katerem ne bo večkratnih pojavitev elementov.

5. V teoriji množic urejeni par (x, y) običajno definiramo kot množico $\{\{x\}, \{x, y\}\}$. Spremenite ukaz `kartezicni` tako, da bo elemente kartezičnega produkta vrnil v taki obliki. Tako naj klic z argumentoma `[a b]` in `[1 2]` vrne seznam

```
[[[a] [a 1]] [[a] [a 2]] [[b] [b 1]] [[b] [b 2]]].
```

6. Ukaz `potencna` razvrsti elemente seznama, ki predstavlja potenčno množico, tako, da so v prvi polovici tisti, ki ne vsebujejo zadnjega elementa, v drugi polovici pa tisti, ki ga vsebujejo. Spremenite ukaz tako, da bodo elementi potenčne množice razvrščeni po moči, pri enaki moči pa na enak način kot doslej. Na primer, klic z argumentom `[a b c]` naj vrne

```
[[] [a] [b] [c] [a b] [a c] [b c] [a b c]].
```

7. Celotno skupino ukazov za delo z množicami bi lahko zasnovali tudi drugače. Vsako množico bi predstavili v "kanonski" obliki z urejenim seznamom brez ponovitev elementov. Pri taki predstavitvi se nekatere operacije nad množicami poenostavijo oziroma jih lahko sprogramiramo bolj učinkovito. Na primer, enakost množic lahko preverimo kar z vgrajenim operatorjem `=`. Poskusite sestaviti ukaze, ki bodo delali s tako predstavljenimi množicami. Seveda morate paziti, da bodo rezultati, ki jih vračajo ukazi, spet v "kanonski" obliki.

Martin Juvan