Univerzitet u Novom Sadu
Prirodno-matematički fakultet
Departman za matematiku i informatiku

Dragan Mašulović

Uvod u programiranje II
(za gimnazijalce)

Novi Sad, 2016.

Glava 1

Sortiranje i pretraživanje

Ovu glavu ćemo posvetiti naprednim programerskim tehnikama: sortiranju niza brojeva i traženju elementa niza. Pre toga, međutim, moramo da naučimo kako da prenesemo niz kao argument procedure ili funkcije.

1.1 Sortiranje

Postupak kojim se dati niz brojeva uređuje u neopadajući ili nerastući poredak, recimo ovako:

$$7, 2, 9, 2, 9, 1, 3, -1 \longrightarrow -1, 1, 2, 2, 3, 7, 9, 9$$

se zove *sortiranje*. Postupci za sortiranje se po brzini dele u dve velike grupe

- *inferiorni* (koji su uglavnom spori), i
- *superiorni* (koji su uglavnom znatno brži).

Inferiorni postupci se zasnivaju na jednostavnim idejama i lako se implementiraju. Superiorni postupci su mnogo komplikovaniji. U ovom odeljku ćemo videti nekoliko standardnih inferiornih postupaka za sortiranje (dva superiorna postupka ćemo videti naredne godine). U narednim procedurama pretpostavljamo da je

```
const
  MaxEl = 1000;
type
  Niz = array [1 .. MaxEl] of integer;
```

Selection sort. *Selection sort* radi ovako: nađe se najmanji element u segmentu `a[1 .. n]` i on se razmeni sa prvim elementom niza. Potom se nađe najmanji element u segmentu `a[2 .. n]` i on razmeni mesto sa drugim elementom niza, i tako do kraja. Uz gornje deklaracije, procedura koja sortira segment `a[1 .. n]` niza `a` izgleda ovako:

```
procedure Swap(var a : Niz; p, q : integer);
var
  pom : integer;
begin
  pom := a[p]; a[p] := a[q]; a[q] := pom
end;

procedure SelectionSort(var a : Niz; n : integer);
var
  min, i, j : integer;
begin
  for i := 1 to n - 1 do
    begin
      { trazenje minimuma u segmentu a[i .. n] }
      min := i;
      for j := i + 1 to n do
        if a[j] < a[min] then min := j;

      Swap(a, i, min)
    end
end;
```

Bubble sort. *Bubble* (engl. mehuričasti) *sort* razmenjuje mesta susednim elementima niza koji stoje u pogrešnom redosledu, sve dok se ne postigne da za svaki par susednih elemenata važi da je prvi manji ili jednak sa drugim. Tada je i ceo niz sortiran (`a[1] <= a[2]` i `a[2] <= a[3]` i `a[3] <= a[4]`, itd). Bubble sort je dobio takvo ime zato što kada radimo sortiranje u nerastućem poretku, najmanji elementi odmah isplivavaju na površinu, poput mehurića.

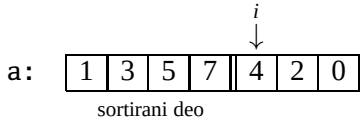
```
procedure BubbleSort(var a : Niz; n : integer);
var
  kraj : boolean;
  i    : integer;
begin
  repeat
    kraj := true;
    for i := 1 to n - 1 do
      if a[i] > a[i+1] then
```

```
        begin
          kraj := false;
          Swap(a, i, i+1)
        end
  until kraj;
end;
```

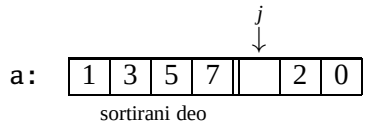
Insertion sort. *Insertion sort* (= sortiranje sa umetanjem) radi ovako: pretpostavimo da smo već sortirali segment `a[1 .. s]`; element `a[s+1]` umećemo na odgovarajuće mesto u segment `a[1 .. s]`. Vrednost `a[s+1]` se prepiše u pomoćnu promenljivu `pom`, onda se elementi segmenta `a[1 .. s]` počev od kraja pomeraju jedno mesto udesno dok se ne napravi praznina na pravom mestu, gde se potom ubaci vrednost iz promenljive `pom`.

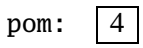
```
procedure InsertionSort(var a : Niz; n : integer);
var
  i, j, pom : integer;
  kraj : boolean;
begin
  for i := 2 to n do
    if a[i-1] > a[i] then
      begin
        { umetni a[i] u segment a[1 .. i-1] }
        pom := a[i];
        j := i;
        repeat
          a[j] := a[j-1];
          j := j - 1;
          if j > 1 then
            kraj := a[j-1] <= pom
          else
            kraj := true
        until kraj;
        a[j] := pom
      end
end;
```

Pogledajmo proces umetanja na jednom primeru. Neka je tokom sortiranja polazni niz stigao do oblika

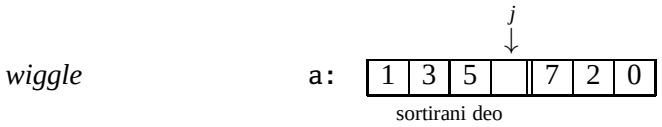


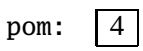
Kako je `a[i]` manji od svog prethodnika, potrebno je umetnuti ga na odgovarajuće mesto u nizu. Prvi korak se sastoji u tome da se vrednost iz `a[i]` prebaci u pomoćnu promenljivu `pom`:

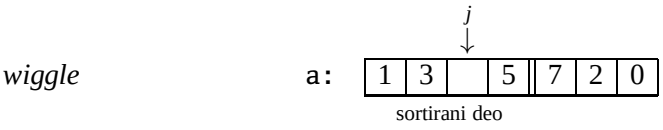


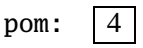


Potom se pravi mesto za vrednost iz `pom` tako što se sadržaji kućica iz sortiranog dela premeštaju jedna po jedna udesno, dok se praznina ne pojavi na pravom mestu.

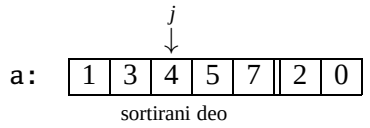


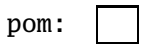






Na kraju se vrednost iz `pom` prepiše u `a[j]` i time je sortirani deo niza uvećan za jednu kućicu:



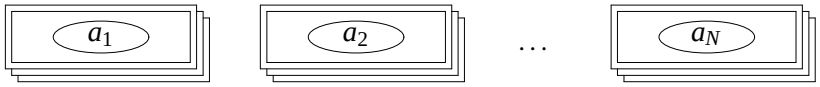


Zadaci.

1.1. Na takmičenju iz računarstva je učestvovalo n takmičara koji su rešavali 4 zadatka. Napisati Paskal program koji od korisnika učitava broj n, potom za svakog takmičara učitava 4 broja – rezultate po zadacima, a potom određuje i štampa rang listu poena. Rezultati za jednog takmičara se unose u jednoj liniji razdvojemi jednim razmakom. Na primer,

```
Broj takmicara ->  4
Takmicar 1 ->   10 10 30 50
Takmicar 2 ->   0 0 0 0
Takmicar 3 ->   50 50 50 50
Takmicar 4 ->   0 50 0 10

Rang lista poena:
Takmicar 3: 200
Takmicar 1: 100
Takmicar 4:  60
Takmicar 2:   0
```

1.2. Na raspolaganju imamo k_1 novčanica u vrednosti od a_1 dinara, k_2 novčanica u vrednosti od a_2 dinara, ..., k_N novčanica u vrednosti od a_N dinara.



k_1 k_2 k_N

Napisati Paskal program koji od korisnika učitava N, $1 \leqslant N \leqslant 100$, potom parove $(k_1, a_1), \ldots, (k_N, a_N)$ i na kraju pozitivan ceo broj m koji predstavlja količinu novca, a onda "isplaćuje" korisniku iznos od m dinara koristeći najmanji mogući broj novčanica.

Ukoliko to nije moguće, isplati korisniku najbolje što može, i obavesti ga o iznosu koji preostaje.

1.3. Napisati Paskal program koji od korisnika učitava paran ceo broj n, potom n po parovima različitih realnih brojeva $x_1, \ldots, x_n$ i određuje i štampa realan broj y sa osobinom da je tačno polovina učitanih brojeva strogo manja od y (što onda znači da je druga polovina učitanih brojeva strogo veća od y). Na primer, za $n = 10$ i za brojeve

```
1.5 3.7 2.25 9.81 3.1415 -0.26 2.9 8.11 10.12 -5.41
```

jedna mogucnost za y je $y = 3.02075$ zato što je tačno pet od navedenih deset brojeva strogo manje od y, a preostalih pet je strogo veće od y.

1.4. *Testerasti sort* je proces kojim se niz brojeva uređuje na sledeći način: na prvo mesto niza se dovede najveći element niza, na drugo najmanji od preostalih elementa, na treće najveći od preostalih elementa, na četvrto najmanji od onih elemenata niza koji su ostali, i tako dalje. Na primer, testerastim sortiranjem se od niza

$$4, \quad -1, \quad 15, \quad -40, \quad 22, \quad -10$$

dobija niz

$$22, \quad -40, \quad 15, \quad -10, \quad 4, \quad -1.$$

Napisati Paskal program koji testerasto sortira niz brojeva.

1.5. Napisati Paskal program koji od korisnika učitava cele brojeve n i k, $1 \leqslant k \leqslant n \leqslant 1000$, potom učitava n realnih brojeva za koje znamo da su svi različiti i određuje i štampa onaj od tih brojeva koji je k-ti po veličini.

1.6. Napisati Paskal program koji od korisnika učitava nekoliko realnih brojeva (ne više od 2000) među kojima može biti i istih, i određuje i štampa koliko se puta koji broj pojavio. Na primer, za niz brojeva

$$1, \quad 2, \quad 1, \quad 3, \quad 1, \quad 2, \quad 1, \quad 2$$

program ispisuje:

```
Broj 1 se pojavio 4 puta.
Broj 2 se pojavio 3 puta.
Broj 3 se pojavio 1 puta.
```

1.7. Rang liste najboljih američkih i evropskih atletičara date su nizovima `a[1 .. n]` i `e[1 .. m]`. To znači da su navedeni segmenti sortirani u nerastućem poretku. Napisati program kojim se *bez ponovnog sortiranja* formira zajednička rang lista `s[1 .. n+m]`. (Uputstvo: prisetiti se mešanja karata!)

1.8. Napisati program koji učitava imena i prezimena učenika jednog odeljenja za zatim ih sortira po prezimenima. Ime i prezime učenika se unosi kao string u kome su ime i prezime odvojeni jednim razmakom. Na primer, `'Petar Petrovic'`.

1.9. (Statističko razbijanje šifri) Sifrovana poruka se sastoji od niza velikih slova i praznina, bez znakova interpunkcije ili nekih drugih znakova. Jedan od metoda za dešifrovanje takvih poruka se sastoji u tome da se odrede frekvencije slova u poruci i da se uporede sa frekvencijama koje su standardne za jezik na kome je napisana poruka. Tada najfrekventnije slovo

u datoteci najverovatnije odgovara najfrekventnijem slovu u jeziku, i tako dalje.

U datoteci `frekv.txt` nalaze se frekvencije pojedinih slova karakteristične za neki jezik. Napisati Paskal program koji učitava datoteku `poruka.txt` u kojoj se nalazi šifrovana poruka, određuje frekvencije slova iz poruke, i na osnovu tog niza frekvencija i niza frekvencija iz datoteke `frekv.txt` predlaže ključ za dešifrovanje poruke.

1.2 Pretraživanje

Problem pretraživanja se sastoji u sledećem. Dat je niz `a[1 .. n]` i dat je broj `b`. Naći indeks `k` takav da je `a[k] = b`, a ukoliko takvo `k` ne postoji, vratiti 0 kao rezultat. Naivno rešenje ovog problema je veoma jednostavno:

```
function Nadji(a : Niz; n, b : integer) : integer;
var
    i : integer;
    nasao : boolean;
begin
    nasao := false; i := 1;
    while not nasao and (i <= n) do
        if a[i] = b then
            nasao := true
        else
            i := i + 1;

    if nasao then Nadji := i
    else Nadji := 0
end;
```

Postoji jedna vesela dosetka koja omogućuje da se prethodni program znatno pojednostavi. Osnovni razlog što je prethodno rešenje relativno komplikovano je taj što mi zapravo ne znamo da li se element nlazi u nizu ili ne. Ako bismo bili sigurni da se trazeni element nalazi u nizu, while-petlja bi bila mnogo jednostavnija.

Dosetka koju ćemo opisati se zove *pretrazivanje sa graničnikom*, *tehnika anđela čuvara* ili engleski *sentinel*, a sastoji se u tome da se na poziciju $n+1$ postavi element koga tražimo. Tako indeks ne može da "spadne sa niza" zato što se pretraživanje uvek srećno završi. Ako se završilo sa pozicijom $n+1$, onda znači da u polaznom nizu traženi element ne postoji. Odgovarajuća funkcija sada izgleda ovako:

```
function NadjiSent(a : Niz; n, b : integer) : integer;
var
    i : integer;
begin
    a[n+1] := b;
    i := 1;
    while a[i] <> b do i := i + 1;

    if i <= n then NadjiSent := i
    else NadjiSent := 0
end;
```

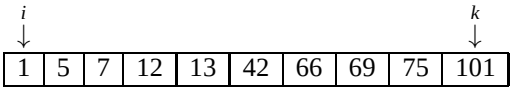
Ukoliko je polazni niz a sortiran, odgovarajući element se može naći *mnogo brže* koristeći ideju koja se zove binarno pretraživanje (engl. binary search).

Ideja binarnog pretraživanja je takođe jednostavna. Posmatramo srednji element niza, ako je on manji od elementa koga tražimo, pretraživanje treba nastaviti u desnoj polovini. Ako je veći, pretraživanje se nastavlja u levoj polovini. U oba slučaja , pretraživanje se nastavlja na isti način: ponovo se posmatra srednji element i na osnovu njegovog odnosa prema elementu koga tražimo pretraživanje se nastavlja u levoj ili desnoj četvrtini.

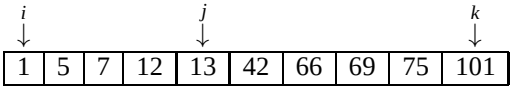
```
    function BinSearch(a : Niz; n, b : integer) : integer;
    var
        i, j, k : integer;
        nasao : boolean;
    begin
        nasao := false;
        i := 1;
        k := n;
        while (k - i > 1) and not nasao do
            begin
                j := (i + k) div 2;
                if b <= a[j] then k := j
                else i := j;
                nasao := b = a[j]
            end;

        if nasao then         BinSearch := j
        else if b = a[i] then BinSearch := i
        else if b = a[k] then BinSearch := k
        else                  BinSearch := 0
end;
```

Pogledajmo jedan primer. Pretpostavimo da u sortiranom nizu 1, 5, 7, 12, 13, 42, 66, 69, 75, 101 treba naći poziciju elementa 42.

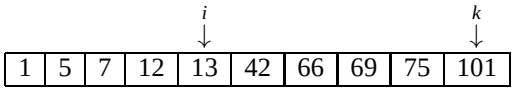
Na početku je $i = 1$ i $k = n = 10$.



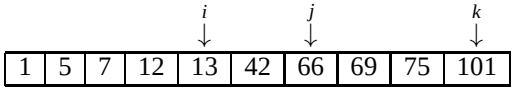
Potom stavimo $j = \left\lfloor \frac{i+k}{2} \right\rfloor = 5$ i poredimo element na poziciji j sa brojem koji se traži.



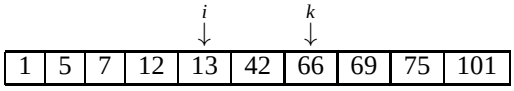
Kako je $13 < 42$, pretraživanje se nastavlja u desnom delu niza tako što se izvrši naredba `i := j`.



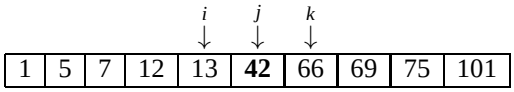
Ponovo posmatramo srednji element $j = \lfloor \frac{i+k}{2} \rfloor = 7$ i poredimo element na poziciji j sa brojem koji se traži.



Kako je $66 > 42$, pretraživanje se nastavlja u levom delu podniza tako što se izvrši naredba `k := j`.

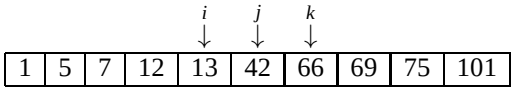


Ovaj put, srednji element je $j = \lfloor \frac{i+k}{2} \rfloor = 6$. Kako je element na poziciji j jednak 42, pretraživanje se završava.

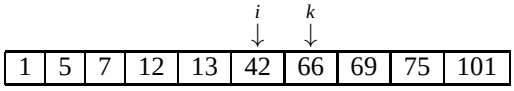


Pogledajmo sada malo drugačiji primer. Recimo da smo u istom nizu tražili broj 43.

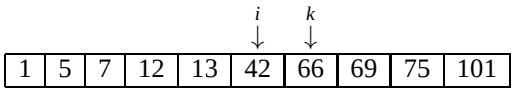
Kao i gore, u nekoliko koraka došli bismo do pozicije prikazane desno.



Zbog $42 < 43$ pretraživanje se nastavilja u desnom otsečku, što znači da se izvršava naredba `i := j`.



Međutim, $k - i = 1$, pa se petlja završava. Sada "if" nakon petlje utvrdi da nijedan od brojeva na pozicijama i, k nije jednak traženom broju, pa funkcija vraća 0.



1.3 Traženje podstringa u stringu

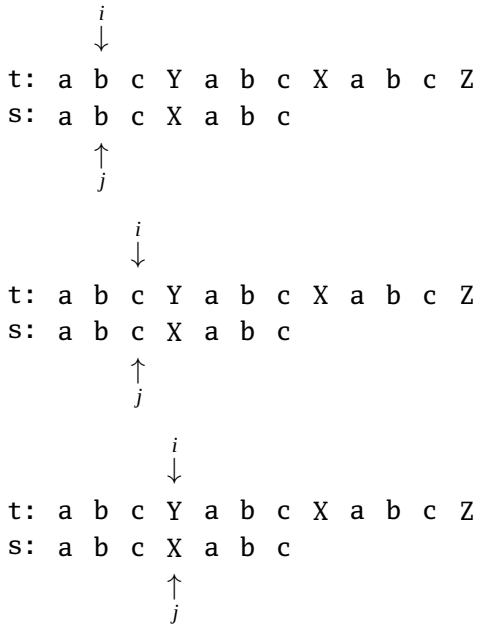
Poseban oblik pretraživanja predstavlja traženje podstringa u datom stringu. Kao što smo videli, postoji ugrađena funkcija `pos` koja obavlja taj posao. Pogledajmo sada ona radi. Napisaćemo funkciju `Find` koja vraća 0 ako `s` nije podstring stringa `t`, a ako je `s` podstring stringa `t` onda vraća poziciju na kojoj se `s` prvi put javlja kao podstring stringa `t`. Na primer,

```
Find('pera', 'mika') = 0
Find('ma', 'sa mamama') = 4.
```

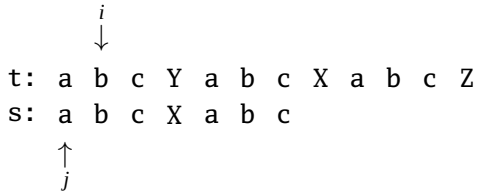
```
function Find(s, t : string) : integer;
{ pretpostavljamo da su s i t neprazni }
var
  i, j : integer;
begin
  i := 1; j := 1;
  repeat
    if t[i] = s[j] then
      begin
        i := i + 1; j := j + 1
      end
    else
      begin
        i := i - j + 2; j := 1
      end
  until (j > length(s)) or (i > length(t));
  if j > length(s) then
    Find := i - length(s)
  else
    Find := 0
end;
```

Indeks `i` šeta duž stringa `t`, a indeks `j` duž stringa `s`. Sve dok se odgovarajući karakteri poklapaju, indeksi simultano napreduju za po jednu kućicu:

```
        i
        ↓
  t: a b c Y a b c X a b c Z
  s: a b c X a b c
        ↑
        j
```



Kada prvi put naiđemo na situaciju da je `t[i]` $\neq$ `s[j]` indeks `j` oborimo na 1, a indeks `i` postavimo na novu početnu vrednost koja je za jedan veća od prethodne početne vrednosti:



Zadaci.

1.10. Napisati funkciju

```
function NadjiOd(a : Niz; n, m, b : integer) : integer;
```

koja u segmentu `a[m .. n]` niza `a` nalazi prvo pojavljivanje broja `b`. Funkcija kao rezultat vraća `0` ako se broj `b` ne nalazi u navedenom segmentu.

1.11. Napisati proceduru

```
procedure NadjiSVE(a : Niz; n, b : integer; var p : Niz;
                   var m : integer);
```

koja u segmentu `a[1 .. n]` niza `a` nalazi sva pojavljivanja broja `b`. Procedura kao rezultat vraća niz `p` pozicija na kojima se nalazi broj `b`, dok `m` sadrži broj pojavljivanja boraj `b` u segmentu `a[1 .. n]`.

1.12. Napisati funkciju koja određuje broj pojavljivanja stringa `s` u stringu `t`.

1.13. Morzeov kōd izgleda ovako:

A	·–	H	····	O	–––	U	··–
B	–···	I	··	P	·––·	V	···–
C	–·–·	J	·–––	Q	––·–	W	·––
D	–··	K	–·–	R	·–·	X	–··–
E	·	L	·–··	S	···	Y	–·––
F	··–·	M	––	T	–	Z	––··
G	––·	N	–·				

U tekstualnoj datoteci `poruka.txt` nalazi se poruka kodirana Morzeovim kodom. Između kodova dva uzastopna slova u poruci nalazi se tačno jedna praznina, a između kodova dve uzastopne reči nalaze se tačno dve praznine. Napisati Paskal program koji čita kodiranu poruku iz datoteke, dekodira je i originalnu poruku prikazuje na ekranu.

‡**1.14.** Programi koji se zovu *spell-checker*i imaju zadatak da u tekstu pronađu reči koje su možda pogrešno napisane. Ovi programi rade tako što svaku reč iz teksta koji se proverava potraže u rečniku, što je unapred pripremljen spisak svih korektno zapisanih reči kojih je sistem svestan. Ako se reč ne nalazi u rečniku, sistem je na neki način označi kao pogrešno napisanu reč.

Napisati Paskal program koji proverava da li su sve reči u datoj tekstualnoj datoteci korektno napisane. Pretpostavljamo da tekstualna datoteka sadrži tekst na engleskom jeziku. Za ovaj zadatak će vam trebati rečnik korektno zapisanih engleskih reči, što se može pronaći na Internetu. Program prvo treba da učita ceo rečnik u jedan veliki niz u memoriji, i onda treba svaku reč iz tekstualne datoteke koja se proverava da potraži u rečniku. Ako reč ne postoji u rečniku treba je ispisati na ekran. Obzirom da se očekuje da će rečnik sadržati veliki broj reči, preporučljivo je implementirati binarno pretraživanje.

1.4 Quicksearch algoritam

Proces traženja podstringa unutar drugog (znatno dužeg) stringa je izuzetno značajan posao koji se može sresti kako u programima za obradu teksta (*word processors*), tako u algoritmima koji se koriste u istraživanjima kao što je nalaženje određenih segmenata DNK sekvence. Međutim, naivno rešenje koje smo videli u prethodnom poglavlju često nije dovoljno efikasno za realne primene. Problem je u tome što pri prelasku na naredni pokušaj naivni algoritam zaboravi sve što je saznao o karakterima stringa `s` dok je proveravao da li se poklapaju sa odgovarajućim delom stringoa `t`. Napredni algoritmi za traženje stringa koriste dodatne informacije kako bi preskočili pokušaje za koje se unapred može zaključiti da će biti neuspešni. Na taj način se povećava efikasnost algoritma, ali se zato dobijaju algoritmi koji su složeniji i koji pre nego što počnu da traže podstring unutar datog stringa moraju da potroše neko vreme na fazu preprocesiranja u kojoj se analizira struktura stringa `s`. U ovoj glavi ćemo razmotriti *Quicksearch* algoritam,[1] što je jedan od najefikasnijih algoritama za traženje podstringa unutar stringa.

Pogledajmo, prvo, na jednom primeru kako radi naivno rešenje dato funkcijom `Find` na kraju prethodnog poglavlja. Recimo da tražimo string `s = 'problem'` unutar stringa `t = 'programi za resavanje svih problema'`.

Prva tri karaktera stringa `s` se poklapaju sa odgovarajućim karakterima stringa `t`, a neslaganje se javlja na poziciji 4 stringa `s`.

```
programi za resavanje svih problema
problem
   ↑
```

Pomerimo string `s` jedno mesto u desno i pokušamo ponovo. Do neslaganja dolazi već na prvoj poziciji stringa `s`.

```
programi za resavanje svih problema
 problem
 ↑
```

Pomerimo string `s` jedno mesto u desno i pokušamo ponovo. Do neslaganja opet dolazi na prvoj poziciji stringa `s`.

```
programi za resavanje svih problema
  problem
  ↑
```

[1] D. M. Sunday, *A very fast substring search algorithm*, Communications of the Association for Computing Machinery, Vol. 33, No. 8, 132–142, August 1990; ovaj algoritam se često može sresti i pod imenom *Synday* algoritam

Neuspesi se ređaju jedan za drugim, a do uspešne situacije

```
programi za resavanje svih problema
                           problem
```

stižemo tek u 28. pokušaju.

Ako pažljivo pogledamo ovaj proces možemo lako da primetimo da su neki pokušaji u startu bili osuđeni na propast. Na primer, kada prvi pokušaj

```
programi za resavanje svih problema
problem
   ↑
```

nije uspeo, odmah je bilo jasno da ni drugi pokušaj neće uspeti, zato što će se pomeranjem za jedno mesto poslednji karakter stringa `s`, a to je `'m'`, poklopiti u stringu `t` sa karakterom `'i'`:

```
       ↓
programi za resavanje svih problema
 problem
       ↑
```

Štaviše, pošto se karakter `'i'` ne javlja nigde u stringu `s`, kao drugi korak smo slobodno mogli da uzmemo situaciju u kojoj smo string `s` "gurnuli" iza pozicije sudbonosnog karaktera `'i'`:

```
       ↓
programi za resavanje svih problema
        problem
```

Ovo je osnovna ideja Quicksearch algoritma! Uz još nekoliko dodatnih ideja Quicksearch algoritam radi na sledeći način.

Quicksearch algoritam:

(1) Proverimo da li se string `s` u tekućoj poziciji poklapa sa odgovarajućim delom striga `t`.

(2) Ako to nije slučaj, uočimo onaj karakter stringa `t` koji se nalazi *odmah iza* poslednjeg karaktera stringa `s` na tekućoj poziciji. Neka je to karakter `c`.

(3) Ako se karakter `c` ne javlja nigde u stringu `s`, pomerimo string `s` u desno za $m+1$ mesta, gde je m dužina stringa `s`. Tako smo "preskočili" karakter `c` i time postigli da se prvi karakter u stringu `s` poravna sa karakterom stringa `t` koji se nalazi odmah iza `c`.

(4) Ako se karakter `c` javlja u stringu `s`, pomerimo string `s` u desno tako da se najdesnije pojavljivanje karaktera `c` u stringu `s` poravna sa karakterom `c` u stringu `t`.

Primer. Neka je `t = 'oprobana torta od banana'` i `s = 'banana'`.

```
oprobana torta od banana
banana
```

Pošto se string `s` u tekućoj poziciji ne poklapa sa odgovarajućim delom stringa `t` i pošto se u stringu `t` iza poslednjeg karaktera stringa `s` nalazi slovo `'n'`:

```
      ↓
oprobana torta od banana
banana
```

vidimo da je nastao slučaj (4), pa pomerimo string `s` za dva mesta u desno tako da se najdesnija pojava slova `'n'` u stringu `s` poklopi sa uočenom pojavom slova `'n'` u stringu `t`.

```
oprobana torta od banana
  banana
```

I dalje string `s` ne odgovara odgovarajućem delu stringa `t`. Ovaj put se u stringu `t` iza poslednjeg karaktera stringa `s` nalazi praznina:

```
        ↓
oprobana torta od banana
  banana
```

Kako string `s` ne sadrži prazninu, vidimo da je nastao slučaj (3) pa pomeramo string `s` odmah za sedam mesta u desno, tako da se prvo slovo stringa `s` na novoj poziciji poravna sa prvim slovom stringa `t` odmah iza uočene praznine:

```
        ↓
oprobana torta od banana
         banana
```

I dalje nema poklapanja, i pri tome se slovo `'o'` ne nalazi u stringu `s`, pa string `s` ponovo pomeramo za sedam mesta u desno (slučaj (3)):

```
                      ↓
oprobana torta od banana
                banana
```

U sledećem koraku opet imamo situaciju (4) pa string `s` pomeramo dva mesta u desno:

```
oprobana torta od banana
                  banana
```

i time algoritam uspešno završava rad.

Implementacija Quicksearch algoritma. Iz prethodnog primera se vidi i kako se računa pomeraj stringa `s` u zavisnosti od toga koje slovo se nalazi u stringu `t` neposredno iza tekuće pozicije stringa `s`. Neka je m dužina stringa `s` i neka je `c` onaj karakter stringa `t` koji se nalazi odmah iza poslednjeg karaktera stringa `s` na tekućoj poziciji.

- Ako se `c` ne pojavljuje u `s`, stavimo `pomeraj[c] :=` $m+1$.
- Ako se `c` pojavljuje u `s`, stavimo `pomeraj[c] :=` $m-k+1$, gde je k pozicija najdesnijeg pojavljivanja slova `c` u stringu `s`.

Primer. Neka je `s = 'banana'`. Tada je $m = 6$, a vrednosti pomeraja su date u sledećoj tabeli:

`c`	`'a'`	`'b'`	`'n'`	ostala slova
k	6	1	5	nije definisano
`pomeraj`	1	6	2	$7 = m+1$

Naravno, prilikom traženja stringa `s` u stringu `t` nećemo fizički pomerati string `s` kako bismo ga preklopili sa odgovarajućim stringa `t` već "pomeranje", kao i kod funkcije `Find` iz prethodnog poglavlja, realizujemo promenom vrednosti dva indeksa od kojih jedan ide po stringu `s`, a drugi po stringu `t`. Konačno, evo i procedure koja implementira algoritam:

```
function QuickSearch(s, t : string) : integer;
{ pretpostavljamo da su s i t neprazni }
var
  i, j, k : integer;
  pomeraj : array[char] of integer;
begin
  for k := 0 to 255 do pomeraj[chr(k)] := length(s) + 1;
  for k := 1 to length(s) do pomeraj[s[k]] := length(s) - k + 1;
```

```
  i := 1; j := 1;
  repeat
    if t[i] = s[j] then
      begin
        i := i + 1; j := j + 1
      end
    else
      begin
        i := i - j + 1 + pomeraj[t[i - j + 1 + length(s)]]; j := 1
      end
  until (j > length(s)) or (i > length(t));
  if j > length(s) then
    QuickSearch := i - length(s)
  else
    QuickSearch := 0
end;
```

Glava 2

Skupovi

U ovom poglavlju se srećemo sa dve novom strukturom podataka: sa skupovima. Skupovi predstavljaju modele konačnih skupova elemenata istog nabrojivog tipa.

2.1 Skupovi

Programski jezik Pascal podržava tip podataka `set` koji predstavlja model konačnog skupa. Iz razloga koje ćemo objasniti nešto kasnije elementi skupa u programskom jeziku Pascal mogu biti elementi nekog prostog tipa kao što je `integer`, `Boolean` i `char`, ali *to ne sme biti* tip `real`. Promenljiva skupovnog tipa se deklariše na jedan od ova dva načina:

```
var
  s : set of ⟨prost-tip⟩;
  t : set of ⟨konstantna-vrednost⟩ .. ⟨konstantna-vrednost⟩;
```

Primer. Uz deklaracije pored na snazi, slementi skupova `a` i `b` mogu biti samo celi brojevi između 0 i 99, elementi skupa `Blanks` mogu biti proizvoljni karakteri, dok je `c` skup čiji elementi mogu biti proizvoljni celi brojevi.

```
var
  a, b : set of 0..99;
  c : set of integer;
  Blanks : set of char;
```

Konstantne vrednosti skupovnog tipa se pišu tako što se između uglastih zagrada navedu elementi skupa. Tako, uz deklaracije iz prethodnog primera na snazi sledeće naredbe dodele su korektne:

```
a := [1, 3, 5, 9];  b := [0, 1, 20..50, 90]; c := [-10 .. 10];
Blanks := [chr(9), chr(10), chr(13), ' '];
```

Naredbom `a := []` se skupovnoj promenljivoj `a` dodeljuje prazan skup.

Programski jezik Pascal poznaje sledeće operacije sa promenljivim skupovnog tipa:

Operacija	Značenje (rezultat je uvek skup)
`s + t`	unija skupova
`s * t`	presek skupova
`s - t`	razlika skupova

kao i sledeće operatore poređenja:

Operacija	Značenje (rezultat je uvek tipa Boolean)
`s = t`	skupovi su jednaki
`s <> t`	skupovi su različiti
`s < t`	`s` je strogi podskup od `t`
`s <= t`	`s` je podskup od `t`
`s > t`	`s` je strogi nadskup od `t`
`s >= t`	`s` je nadskup od `t`
`el in s`	`el` je element skupa `s`

Na primer:

Izraz	Značenje
`s <= t1 * t2`	skup `s` je podskup i skupa `t1` i skupa `t2`
`c in Blanks`	znak `c` je element skupa `Blanks`

U memoriji računara skup je predstavljen svojom karakterističnom funkcijom, dakle, kao niz bitova. Na primer, ako je

```
var
    a : set of 0..15;
```

za promenljivu `a` biće rezervisano 16 bitova i nakon dodele `a := [0..3, 12, 15]` stanje memorije dodeljene promenljivoj `a` će izgledati ovako:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a:	1	1	1	1	0	0	0	0	0	0	0	0	1	0	0	1

Operacije na skupovima se sada lako realizuju kao logičke operacije bit-po-bit: unija skupova logičko “or” (bit-po-bit), presek skupova je logičko “and” (bit-po-bit), itd.

Ako želimo da napišemo proceduru ili funkciju koja prima skup kao argument, moramo prvo formirati odgovarajući imenovani tip. Na primer deklaracija

```
type
  SmallInts = set of 0 .. 255;
```

uvodi novi tip koji se zove `SmallInts` i koji predstavlja skup čiji elementi mogu biti celi brojevi iz intervala 0, ..., 255. Funkcija koja utvrđuje broj elemenata ovakvog skupa sada može biti deklarisana recimo ovako:

```
function Card(S : SmallInts) : integer;
```

Za kraj napomenimo još samo to da funkcija *ne može* da vrati skup kao svoj rezultat.

Primer. Napisati proceduru koja ispisuje sve elemente skupa tipa `SmallInts`.

```
procedure WrSet(s : SmallInts);
var
  i : integer;
begin
  for i := 0 to 255 do
    if i in s then
      write(i : 4);
  writeln
end;
```

Kviz.

1. Označiti korektne deklaracije:

- ☐ `var a : string;`
- ☐ `var d : string [1..80] of char;`
- ☐ `var e : set of integer;`
- ☐ `var f : set of string;`
- ☐ `var g : set of char;`
- ☐ `var h : set [1..10] of char;`
- ☐ `var i : set of 'a'..'z';`
- ☐ `var j : set of ['$', '#', '?'];`
- ☐ `var l : set of real;`
- ☐ `var m : set [1..10] of real;`
- ☐ `var n : set of 0..99;`
- ☐ `var o : set of 0,1,3,7,12;`
- ☐ `var p : set of (0,1,3,7,12);`
- ☐ `var q : set of [0,1,3,7,12];`
- ☐ `var r : set of [0..99];`
- ☐ `var s : set [-1..3] of [0..99];`

2. Uz sledeće deklaracije:

```
var
  s, t : string;
  a, b : set of 0..99;
  c    : char;
  ok   : Boolean;
```

označiti korektne naredbe dodele:

☐ `s := 'Zdravo';`
☐ `s := "Zdravo";`
☐ `t := '';`
☐ `s[3] := '!';`
☐ `s := s + t + c;`
☐ `t := s - c;`
☐ `c := 'Proba';`
☐ `s := 'a';`
☐ `s[4] := '';`
☐ `c := '''';`
☐ `a := [];`
☐ `a := {};`
☐ `b := a + 3;`
☐ `b := a + [3];`
☐ `a := [1,1,2,2,2,3,3];`
☐ `b := [0,1..5,90..99];`
☐ `b := (b + a) * (b - a);`
☐ `ok := 1 in b;`
☐ `ok := a <= b;`
☐ `ok := 9 in (a + b);`

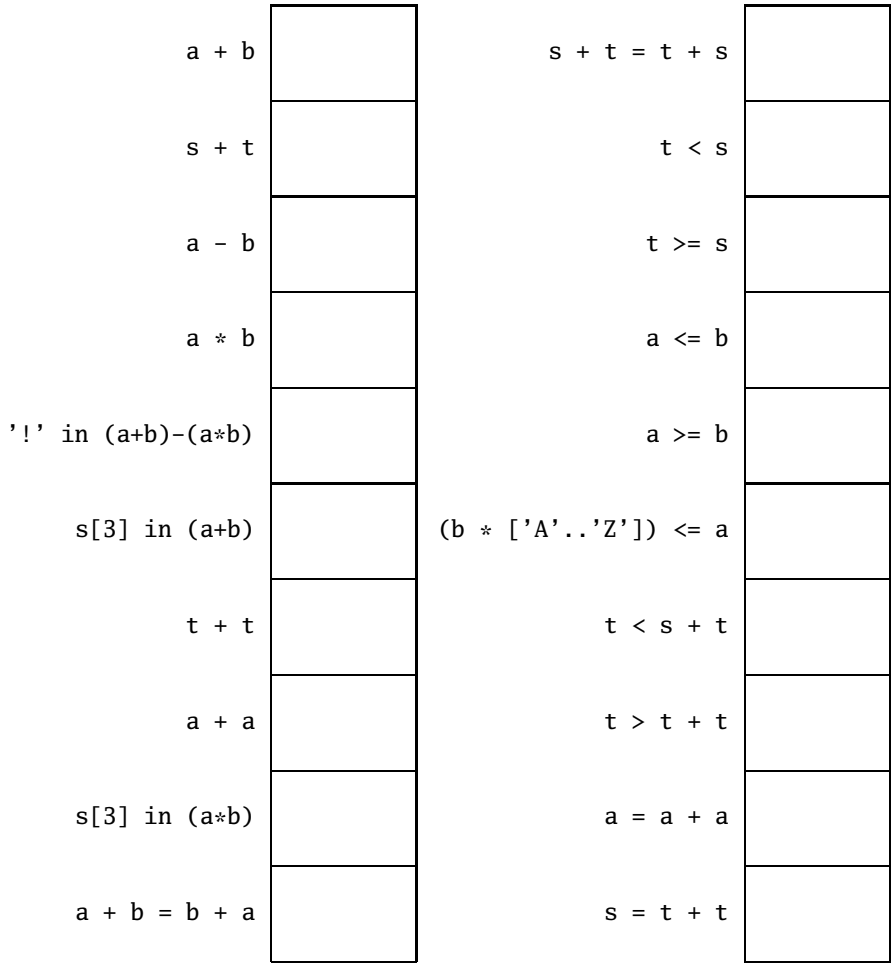
3. Uz deklaracije

```
var
  a, b : set of char;
  s, t : string;
```

i nakon sledećih naredbi dodele:

```
a := ['!', 'A'..'Z'];       s := 'barbara';
b := ['a'..'z', 'A'..'Z'];  t := 'bara';
```

izračunati vrednosti navedenih izraza:



Zadaci.

2.1. Napisati program koji učitava string i utvrđuje koliko znakova tog stringa pripada svakoj od sledećih kategorija znakova:

- slova: `'a'..'z'` i `'A'..'Z'`;
- praznine: tab, CR, LF, space;
- znaci interpunkcije: `!`, `?`, `,`, `.`, `:`, `;`, `'`, `"`;
- matematički simboli: `(`, `)`, `+`, `-`, `*`, `/`, `=`, , `<`, `>`, `%`;
- specijalni simboli: `@`, `#`, `$`, `^`, `&`, `_`, `[`, `]`, `{`, `}`.

2.2. Neka je data deklaracija

```
type
  SmallInts = set of 0 .. 255;
```

Napisati funkciju `function Card(S : SmallInts) : integer` koja utvrđuje broj elemenata datog skupa malih celih brojeva.

2.3. Neka je data deklaracija

```
type
  Chars = set of 'a' .. 'z';
```

Napisati proceduru `procedure PrintChars(S : Chars)` koja ispisuje skup `S` tako što ispiše elemente skupa `S` u abecednom poretku, razdvojene zarezima i sa zagradama { i }. Na primer, nakon naredbe dodele `S := ['b', 'a', 'd'];` procedura `PrintChars(S : Chars)` ispisuje `{a, b, d}`.

2.4. Napisati program koji učitava dva stringa i ispisuje sve znakove koji se pojavljuju u jednom od njih, ali ne i u drugom.

Glava 3

Slogovi i datoteke

Slog (engl. *record*) u programskom jeziku Pascal predstavlja način da se nekoliko podataka različitog tipa tretira kao jedna celina sa stanovišta programskog jezika. Tako se, recimo, podaci o nekoj osobi kao što je ime, prezime, datum rođenja, adresa, pol i JMBG mogu spakovati u jedan "paket podataka", umesto da ih tretiramo kao šest nezavisnih promenljivih. Slogovi se mogu koristiti i za modelovanje nekih tipova podataka koji nisu standardno podržani programskim jezikom Pascal, kao što su kompleksni i racionalni brojevi.

U saradnji sa datotekama dolazimo do druge važne primene slogova, a to je rad sa ogromnim količinama podataka za potrebe poslovnih primena. Datoteka je proizvoljno dugačak niz podataka istog tipa koji se nalazi u spoljašnjoj memoriji. Tip elemenata datoteke je najčešće neki slog, dakle, heterogeni paket podataka koji opisuje neki objekt (ili osobu).

3.1 Slogovi

U svakodnevnom programiranju se često javlja potreba da se nekoliko podataka različitog tipa tretira kao jedna celina sa stanovišta programskog jezika. Mehanizam koji to obezbeđuje zove se *slog*, ili `record` u terminologiji programskog jezika Pascal. Promenljiva tipa `record` ima sledeću deklaraciju

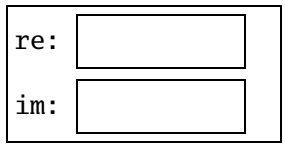
```
var
    a : record
          ⟨ime11⟩, ..., ⟨ime1k⟩ : ⟨tip1⟩;
          ⟨ime21⟩, ..., ⟨ime2l⟩ : ⟨tip2⟩;
            ⋮
          ⟨imen1⟩, ..., ⟨imens⟩ : ⟨tipn⟩
        end;
```

Na primer kompleksan broj je par realnih brojeva i zato je prirodno kompleksne brojeve opisati kao slogove sa dva polja:

Deklaracija promenljive:

```
var
    z : record
      re : real;
      im : real
    end;
```

Memorija:

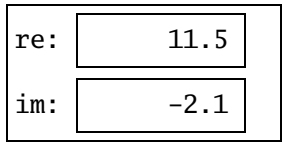


Slog je jedna složena promenljiva koja u sebi sadrži "sitnije promenljive". "Delovi sloga" se zovu *polja sloga*. Poljima sloga se pristupa tako što se iza imena promenljive navede tačka i onda ime polja

$$\langle \textit{ime sloga} \rangle \,.\, \langle \textit{ime polja} \rangle$$

Tako, ako želimo da postavimo vrednost promenljive z, to možemo učiniti ovako:

```
z.re := 11.5;
z.im := -2.1;
```



Pošto polja sloga nisu nezavisne promenljive, imena polja sloga *nisu vidljiva izvan sloga!* Zato će sledeće naredbe

```
re := 3;
```

i

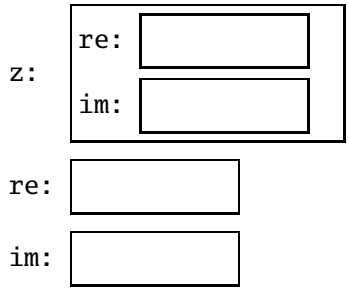
```
im := -1;
```

uzrokovati grešku pri kompilaciji (naravno, ukoliko ne postoje promenljive koje su tako deklarisane). Upravo zato što polja sloga ne postoje kao imena izvan sloga, sledeće je *dozvoljeno i sasvim korektno*:

Deklaracija promenljivih:

```
var
  z : record
    re : real;
    im : real
  end;
  re : char;
  im : Boolean;
```

Memorija:



Veoma retko ćemo promenljive deklarisati direktno kao slogove. Mnogo korisnije je prvo definisati novi tip, pa onda deklaristi promenljivu kao promenljivu tog tipa:

```
type
  Complex = record
    re, im : real
  end;
var
  z : Complex;
```

Ne samo što se na taj način dobija jasniji program, već možemo i direktno dodeljivati promenljive istog tipa jednu drugoj:

```
var
  w, z : Complex;
begin
  ...
  w := z;
  ...
end.
```

Prenos struktura kao argumenata procedura i funkcija se pokorava istim pravilima kao prenos nizova. Da bismo strukturu uneli u proceduru ili funkciju kao njen argument, ona mora biti imenovana. Strukture mogu biti prenete u potprogram po referenci (`var` argumenti) ili po vrednosti ("obični" argumenti, bez var). U prvom slučaju se promene na elementima strukture odslikavaju na strukturi koja je navedena u pozivu potprograma, a u drugom slučaju ne zato što će Pascal prevodilac napraviti kopiju i nju preneti u potprogram. Za kraj ćemo istaknuti da

☞ *Rezultat funkcije* ne može biti slog!

Ukoliko je rezultat rada potprograma neka struktura, ona se mora vratiti kao `var` argument procedure kao u sledećem primeru:

```
procedure Add(var c : Complex; a, b : Complex); { c := a + b }
```

3.2 Kompleksni brojevi

Kompleksani brojevi se u programskom jeziku Pascal mogu na prirodan način predstaviti kao slogovi:

```
type
  Complex = record
    re, im : real
  end;
```

Pošto Pascal ne ume da operiše sa kompleksnim brojevima, programer mora da obezbedi osnovne operacije. Učitavanje i ispis kompleksnih brojeva su jednostavni:

```
procedure ComplexRead(var z : Complex);
begin
  readln(z.re, z.im)
end;

procedure ComplexWrite(z : Complex);
begin
  if (z.re = 0) and (z.im = 0) then write('0')
  else if z.re = 0 then write(z.im : 10 : 2, 'i')
  else if z.im = 0 then write(z.re : 10 : 2)
  else { z.re i z.im su razliciti od nule }
    if z.im > 0 then
      write(z.re : 10 : 2, '+', z.im : 10 : 2, 'i')
    else
      write(z.re : 10 : 2, z.im : 10 : 2, 'i')
end;
```

Naredna procedura konstruiše kompleksan broj ako su dati njegov realni i imaginarni deo. I ona je veoma jednostavna:

```
procedure ComplexAssign(var z : Complex; a, b : real);
begin
  z.re := a;
  z.im := b
end;
```

Od operacija sa kompleksnim brojevima pokazaćemo svega nekoliko: sabiranje kompleksnih brojeva, množenje kompleksnih brojeva i deljenje kompleksnog broja realnim brojem koji nije nula, dok ćemo ostale operacije sa kompleksnim brojevima videti u zadacima:

```
procedure ComplexAdd(var z : Complex; a, b : Complex);
begin
  z.re := a.re + b.re;
  z.im := a.im + b.im
end;

procedure ComplexMul(var z : Complex; a, b : Complex);
begin
  z.re := a.re * b.re - a.im * b.im;
  z.im := a.re * b.im + a.im * b.re
end;
```

```
procedure ComplexDivReal(var z : Complex; a: Complex; b: real);
{ pretpostavlja se da je b <> 0 }
begin
  z.re := a.re / b;
  z.im := a.im / b
end;
```

Primer. Napisati Pascal program koji od korisnika učitava kompleksan broj z i pozitivan ceo broj n i potom računa i štampa vrednost sledećeg izraza

$$1+z+\frac{z^2}{2!}+\frac{z^3}{3!}+\ldots+\frac{z^n}{n!}$$

Rešenje. Kada bi Pascal umeo direktno da radi sa kompleksnim brojevima, onda bi program mogao da izgleda ovako $\longrightarrow$

```
program VoleliBismoDaMozeOvako;
{ ali ne moze! }
var
  z, sum, pom : complex;
  k, n : integer;
begin
  readln(z);
  readln(n);
  sum := 1;
  pom := 1;
  for k := 1 to n do begin
    pom := pom * z / k;
    sum := sum + pom
  end;
  writeln(z)
end.
```

Srećom, ovu ideju je lako implementirati koristeći se tipom `Complex` i operacijama koje smo do sada napisali.

```
program OvakoMoze;
type
  Complex = record
    re, im : real
  end;

var
  z, sum, pom : Complex;
  k, n : integer;

procedure ComplexRead(var z : Complex);
begin
  readln(z.re, z.im)
end;

procedure ComplexWrite(z : Complex);
begin
```

```
  if (z.re = 0) and (z.im = 0) then write('0')
  else if z.re = 0 then write(z.im : 10 : 2, 'i')
  else if z.im = 0 then write(z.re : 10 : 2)
  else { z.re i z.im su razliciti od nule }
    if z.im > 0 then
      write(z.re : 10 : 2, '+', z.im : 10 : 2, 'i')
    else
      write(z.re : 10 : 2, z.im : 10 : 2, 'i')
end;

procedure ComplexAssign(var z : Complex; a, b : real);
begin
  z.re := a; z.im := b
end;

procedure ComplexAdd(var z : Complex; a, b : Complex);
begin
  z.re := a.re + b.re; z.im := a.im + b.im
end;

procedure ComplexMul(var z : Complex; a, b : Complex);
begin
  z.re := a.re * b.re - a.im * b.im;
  z.im := a.re * b.im + a.im * b.re
end;

procedure ComplexDivReal(var z : Complex; a: Complex; b: real);
{ pretpostavlja se da je b <> 0 }
begin
  z.re := a.re / b; z.im := a.im / b
end;

begin
  ComplexRead(z);
  readln(n);
  ComplexAssign(sum, 1, 0);        { sum := 1 + 0*i   }
  ComplexAssign(pom, 1, 0);        { pom := 1 + 0*i   }
  for k := 1 to n do
    begin
      ComplexMul(pom, pom, z);     { pom := pom * z   }
      ComplexDivReal(pom, pom, k); { pom := pom / k   }
      ComplexAdd(sum, sum, pom)    { sum := sum + pom }
    end;
  ComplexWrite(z);
  writeln
```

```
end.
```

Zadaci.

3.1. Napisati sledeće procedure i funkcije za rad sa kompleksnim projevima:

- `procedure ComplexSub(var z : Complex; a, b : Complex);` koja oduzima kompleksne brojeve `a` i `b`: $z = a - b$
- `procedure ComplexDiv(var z : Complex; a, b : Complex);` koja deli kompleksne brojeve `a` i `b`: $z = a/b$; proceduru napisati pod pretpostavkom da `b` nije nula
- `function ComplexAbs(z : Complex) : real;` koja računa modul kompleksnog broja `z`
- `procedure ComplexAddReal(var z : Complex; a : Complex; b : real);` koja na kompleksni broj `a` dodaje realni broj `b`
- `procedure ComplexSubReal(var z : Complex; a : Complex; b : real);` koja od kompleksnog broja `a` oduzima realni broj `b`
- `procedure ComplexMulReal(var z : Complex; a : Complex; b : real);` koja kompleksni broj `a` množi realnim brojem `b`.

3.2. Napisati Pascal program koji od korisnika učitava kompleksan broj z i pozitivan ceo broj n i potom računa i štampa vrednost sledećeg izraza

$(a)\ z - \frac{z^3}{3!} + \frac{z^5}{5!} - \frac{z^7}{7!} + \ldots + (-1)^n \frac{z^{2n+1}}{(2n+1)!}$

$(b)\ 1 - \frac{z^2}{2!} + \frac{z^4}{4!} - \frac{z^6}{6!} + \ldots + (-1)^n \frac{z^{2n}}{(2n)!}$

$(c)\ z - \frac{z^3}{3} + \frac{z^5}{5} - \frac{z^7}{7} + \ldots + (-1)^n \frac{z^{2n+1}}{2n+1}$

3.3. Napisati Pascal program koji od korisnika učitava pozitivan ceo broj $n \geqslant 2$, potom n kompleksnih brojeva $z_1, \ldots, z_n$ i onda računa i štampa vrednost sledećeg izraza:

$$z_1 \cdot z_2 + z_2 \cdot z_3 + \ldots + z_{n-1} \cdot z_n.$$

3.4. Koren kompleksnog broja je vrlo komplikovana funkcija, zato što nije jednoznačna: n-ti koren kompleksnog broja ima n vrednosti i definiše se ovako:

$$\sqrt[n]{z} = \{w \in \mathbf{C} : w^n = z\}.$$

Da bismo izračunali svih n vrednosti n-tog korena datog broja, koristimo trigonometrijsku reprezentaciju kompleksnog broja i de Moivreov obrazac. Naime, svaki kompleksni broj $z = a + bi$ se može predstaviti u *trigonometrijskom obliku* na sledeći način:

$$z = r(\cos\varphi + i\sin\varphi)$$

gde je $r = |z|$, a φ je ugao takav da je $\operatorname{tg}\varphi = \frac{b}{a}$. Tada se na osnovu de Moivreovog obrasca lako vidi da je n-ti koren iz z dat sledećim skupom vrednosti:

$$\sqrt[n]{z} = \left\{ \sqrt[n]{r}\left(\cos\frac{\varphi+2j\pi}{n} + i\sin\frac{\varphi+2j\pi}{n}\right) : j = 0,1,\ldots,n-1 \right\}.$$

Napisati Pascal program koji od korisnika učitava pozitivan ceo broj n i kompleksni broj z i ispisuje svih n n-tih korena broja z. (n-ti koren realnog broja r računati koristeći formulu $\sqrt[n]{r} =$ `exp(ln(r) / n)`.)

3.3 Razlomci

Razlomak $\frac{p}{q}$ je određen sa dva cela broja, svojim brojiocem i imeniocem, pri čemu se uzima da je imenilac uvek pozitivan ceo broj. Slično kompleksnim brojevima, i razlomci se u programskom jeziku Pascal mogu predstaviti kao slogovi:

```
type
  Frac = record
    num, den : integer
  end;
```

Za razliku od kompleksnih brojeva, pri radu sa razlomcima moramo da vodimo malo više računa iz dva razloga:

- imenilac razlomka mora uvek biti pozitivan ceo broj, i
- zato što više parova brojeva određuje jedan isti razlomak ($\frac{3}{2} = \frac{15}{10} = \frac{21}{14} = \ldots$), razlomak ćemo uvek čuvati u skraćenom obliku.

Prvo ćemo pokazati proceduru koja skraćuje dati razlomak i za koju nam je potrebna procedura koja računa NZD dva pozitivna cela broja:

```
function NZD(a, b : integer) : integer;
{ pretpostavlja se da je a > 0 i b > 0 }
var
  r : integer;
begin
  repeat
    r := a mod b; a := b; b := r
  until r = 0;
  NZD := a
end;

procedure FracLowerTerms(var r : Frac);
{ pretpostavlja se da je r.den > 0 }
var
  d : integer;
begin
  if r.num = 0 then r.den := 1
  else begin
    d := NZD(abs(r.num), r.den);
    r.num := r.num div d;
    r.den := r.den div d
  end
end;
```

Procedura za ispis razlomka je krajnje jednostavna, dok procedure koja učitava razlomak vodi računa o tome da imenilac ne sme biti nula, da promeni znak i imeniocu i brojiocu ukoliko je imenilac negativan i da na kraju skrati razlomak:

```
procedure FracWrite(r : Frac);
begin
  write(r.num, '/', r.den)
end;

procedure FracRead(var r : Frac);
begin
  readln(r.num, r.den);
  while r.den = 0 do begin
    writeln('Greska: imenilac = 0');
    readln(r.num, r.den)
  end;
  if r.den < 0 then begin
    r.num := -r.num;
    r.den := -r.den
  end;
  FracLowerTerms(r)
end;
```

Naredna procedura formira razlomak na osnovu dva cela broja koji predstavljaju brojilac i imenilac razlomka. Pretpostavlja se da je imenilac različit od nule:

```
procedure FracAssign(var r : Frac; p, q : integer);
{ pretpostavlja se da je q <> 0 }
begin
  if q < 0 then begin
    p := -p; q := -q
  end;
  r.num := p; r.den := q;
  FracLowerTerms(r)
end;
```

Aritmetičke operacije sa razlomcima se implementiraju direktno. Na primer, evo procedure koja sabira dva razlomka:

```
procedure FracAdd(var r : Frac; a, b : Frac); { r := a + b }
begin
  r.num := a.num * b.den + a.den * b.num;
  r.den := a.den * b.den;
  FracLowerTerms(r)
end;
```

Na kraju navodimo funkciju koja poredi dva razlomka. Ona vraća `-1` ako je prvi razlomak manji od drugog, `0` ako su jednaki i `1` ako je prvi razlomak veći od drugog.

```
function  FracCompare(a, b : Frac) : integer;
{ -1 za a < b, 0 za a = b, 1 za a > b }
begin
  if (a.num = b.num) and (a.den = b.den) then
    FracCompare := 0
  else if a.num * b.den < a.den * b.num then
    FracCompare := -1
  else
    FracCompare := 1
end;
```

Primer. Napisati Pascal program koji od korisnika učitava pozitivan ceo broj n i potom računa i štampa vrednost sledećeg izraza

$$\frac{1}{2\cdot 3}+\frac{2}{3\cdot 4}+\frac{3}{4\cdot 5}+\ldots+\frac{n}{(n+1)(n+2)}.$$

Rešenje. Program je klasičan primer programa koji sumira n brojeva, s tim da su brojevi implementirani tipom `Frac`:

```
program PrimerSaRazlomcima;
type
  Frac = record
    num, den : integer
  end;

var
  n, i : integer;
  sum, r : Frac;

function NZD(a, b : integer) : integer;
{ pretpostavlja se da je a > 0 i b > 0 }
var
  r : integer;
begin
  repeat
    r := a mod b; a := b; b := r
  until r = 0;
  NZD := a
end;

procedure FracLowerTerms(var r : Frac);
var
  d : integer;
begin
  if r.num = 0 then r.den := 1
  else begin
    d := NZD(abs(r.num), r.den);
    r.num := r.num div d;
    r.den := r.den div d
  end
end;

procedure FracWrite(r : Frac);
begin
  write(r.num, '/', r.den)
end;

procedure FracAssign(var r : Frac; p, q : integer);
begin
  if q < 0 then begin
    p := -p; q := -q
  end;
  r.num := p; r.den := q;
  FracLowerTerms(r)
end;
```

```
procedure FracAdd(var r : Frac; a, b : Frac); { r := a + b }
begin
  r.num := a.num * b.den + a.den * b.num;
  r.den := a.den * b.den;
  FracLowerTerms(r)
end;

begin
  readln(n);
  if n <= 0 then writeln('Greska')
  else begin
    FracAssign(sum, 0, 1); { sum := 0 }
    for i := 1 to n do begin
      FracAssign(r, i, (i + 1) * (i + 2));
      FracAdd(sum, sum, r)
    end;
    FracWrite(sum)
  end
end.
```

Primer. Na planeti $\exists\theta\ddot{u}$ žive stvorenja koja su astrozoolozi nazvali *besmrtni puževi sporaći*. Besmrtni puž sporać je puž koji je besmrtan, ali svakim danom gubi snagu, tako da prvog dana svog života može da pređe 1m, drugog dana svog života može da pređe $\frac{1}{2}$m, trećeg $\frac{1}{3}$m, ..., n-tog dana svog života $\frac{1}{n}$m, i tako dalje. I pored toga, besmrtni puž sporać može da stigne gdegod poželi, samo ako mu se da dovoljno vremena. Evo zašto. Primetimo da je

$$1 > \frac{1}{2}$$
$$\frac{1}{2} = \frac{1}{2}$$
$$\frac{1}{3}+\frac{1}{4} > \frac{1}{4}+\frac{1}{4} = \frac{1}{2}$$
$$\frac{1}{5}+\frac{1}{6}+\frac{1}{7}+\frac{1}{8} > \frac{1}{8}+\frac{1}{8}+\frac{1}{8}+\frac{1}{8} = \frac{1}{2}$$
$$\frac{1}{9}+\frac{1}{10}+\frac{1}{11}+\frac{1}{12}+\frac{1}{13}+\frac{1}{14}+\frac{1}{15}+\frac{1}{16} > \frac{1}{16}+\frac{1}{16}+\frac{1}{16}+\frac{1}{16}+\frac{1}{16}+\frac{1}{16}+\frac{1}{16}+\frac{1}{16} = \frac{1}{2}$$

i tako dalje. Koristeći, dakle, činjenicu da je

$$\frac{1}{2^n+1}+\frac{1}{2^n+2}+\ldots+\frac{1}{2^{n+1}} > \frac{2^n}{2^{n+1}} = \frac{1}{2},$$

lako zaključujemo da će besmrtni puž sporać moći da pređe $n \cdot \frac{1}{2}$m za svaki prirodan broj n, ako mu damo dovoljno vremena.

Napisati Paskal program koji od korisnika učitava prirodan broj k i određuje kog dana nakon svog rođenja će besmrtni puž sporać preći k metara.

Rešenje.

```
program BesmrtniPuzSporac;
uses FracUnit;

var
  k : integer; {put koji puz treba da predje}
  d : longint; {broj dana}
  sum, kFrac, pom : Frac;

begin
  repeat
    readln(k)
  until k > 0;
  FracAssign(kFrac, k, 1); {kFrac := k / 1}
  FracAssign(sum, 0, 1); {sum := 0 / 1}
  d := 0;
  while FracCompare(sum, kFrac) = -1 do begin
    inc(d);
    FracAssign(pom, 1, d); {pom := 1 / d}
    FracAdd(sum, sum, pom); {sum := sum + pom}
  end;
  writeln(d)
end.
```

Zadaci.

3.5. Napisati sledeće procedure i funkcije za rad sa razlomcima:

- `procedure FracSub(var r : Frac; a, b : Frac);` koja oduzima dva razlomka,
- `procedure FracMul(var r : Frac; a, b : Frac);` koja množi dva razlomka,
- `procedure FracDiv(var r : Frac; a, b : Frac);` koja deli razlomak `a` razlomkom `b`; proceduru napisati pod pretpostavkom da `b` nije nula,
- `function FracIsZero(r : Frac) : Boolean;` koja proverava da li je dati razlomak jednak nuli.

3.6. Napisati proceduru `procedure Mid3(a, b, c : Frac; var m : Frac);` koja vraća srednji od tri data razlomka.

3.7. Napisati Pascal program koji od korisnika učitava pozitivan ceo broj n, potom n razlomaka $a_1, \ldots, a_n$ i računa i štampa vrednost sledećeg izraza u obliku skraćenog razlomka: $-a_1 + a_2 - a_3 + a_4 + \ldots + (-1)^n a_n$.

3.8. Napisati Pascal program koji od korisnika učitava ceo broj $n \geqslant 3$, potom n razlomaka $q_1, \ldots, q_n$ i računa i štampa vrednost izraza

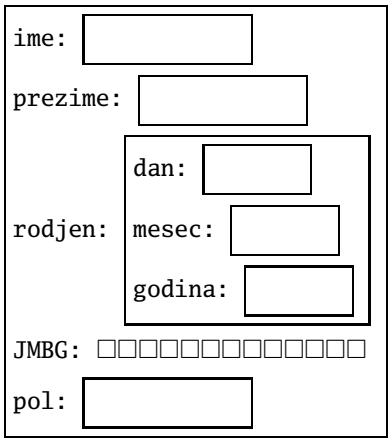
$$q_1 + q_1 q_2 + q_1 q_2 q_3 + \ldots + q_1 q_2 \ldots q_n.$$

(Napomena: Ne znamo gornje ograničenje za broj n, tako da razlomke ne možemo smeštati u niz!)

3.4 Složenije strukture

Polje sloga može imati proizvoljan tip, dakle, to može biti prost tip, ali i bilo koja druga proizvoljno komplikovana struktura. Na primer

```
type
  Datum = record
    dan, mesec, godina : integer
  end;
  Osoba = record
    ime, prezime : string;
    rodjen : Datum;
    JMBG : array [1 .. 13] of integer;
    pol : integer {1 = muski, 2 = zenski}
  end;
var
  Pera : Osoba;
```



Ako je polje sloga ponovo struktura, njegovim elementima se pristupa u zavisnosti od vrste strukture o kojoj je reč. Navodimo jedan način da se popuni promenljiva `Pera` tipa `Osoba`.

```
Pera.ime := 'Petar';          Pera.prezime := 'Petrovic';
Pera.rodjen.dan := 3;         Pera.rodjen.mesec := 3;
Pera.rodjen.godina := 1999;   Pera.pol := 1;
Pera.JMBG[ 1] := 0;           Pera.JMBG[ 2] := 3;
Pera.JMBG[ 3] := 0;           Pera.JMBG[ 4] := 3;
Pera.JMBG[ 5] := 9;           Pera.JMBG[ 6] := 9;
Pera.JMBG[ 7] := 9;           Pera.JMBG[ 8] := 8;
Pera.JMBG[ 9] := 0;           Pera.JMBG[10] := 0;
Pera.JMBG[11] := 0;           Pera.JMBG[12] := 9;
Pera.JMBG[13] := 5;
```

Ne samo da se nizovi mogu javljati kao delovi drugih struktura, već se i svaka druga struktura može pojaviti kao tip elementa niza. Dakle, elementi niza mogu biti neki slogovi, neki drugi nizovi itd. Opšta deklaracija niza u programskom jeziku Pascal ima strukturu:

```
var
    a : array [⟨nabrojivi tip⟩] of ⟨proizvoljan tip⟩;
```

Zadaci.

3.9. Napisti procedure i funkcije:

- `procedure OsobaRead(var o : Osoba);` koja od korisnika učitava podatke o nekoj osobi;
- `procedure OsobaWrite(var o : Osoba);` koja ispisuje podatke o nekoj osobi;
- `function OsobaStarija(p, q : Osoba) : Boolean;` koja vraća `true` ukoliko je osoba `p` starija od osobe `q`.

3.10. Vreme se može opisati ovakvim tipom podataka:

```
type
  Time = record
    hrs : 0..23;
    min : 0..59;
    sec : 0..59
  end;
```

Napisati sledeće procedure i funkcije za rad sa vremenom:

- `procedure TimeRead(var t : Time);` koja učitava vreme,
- `procedure TimeWrite(t : Time);` koja ispisuje vreme u obliku `h:m:s`,

- `procedure TimeAssign(var t : Time; h, m, s : integer);` koja na osnovu vrednosti celobrojnih promenljivih `h`, `m` i `s` postavlja odgovarajuće vreme u `t`; pretpostavlja se da su `h`, `m` i `s` nenegativni brojevi *koji ne moraju biti u očekivanom opsegu!*
- `procedure TimeAdd(var t : Time; a, b : Time);` koja sabira dva vremena,
- `procedure TimeSub(var t : Time; a, b : Time);` koja oduzima dva vremena,
- `procedure TimeIncSec(var t : Time);` koja uvećava dato vreme za jednu sekundu,
- `function TimeInBetween(a, b, c : Time) : Boolean;` koja proverava da li je `b` između `a` i `c`,
- `function TimeBefore(a, b : Time) : Boolean;` koja proverava da li je `a` pre `b`, i
- `function TimeEqual(a, b : Time) : Boolean;` koja proverava da li je `a` jednako sa `b`.

3.11. Napisati program koji od korisnika učitava pravougaonu šemu brojeva i sortira njene vrste prema prvom elementu.

3.12. Tip `Poligon` je definisan na sledeći način:

```
const
  MaxN = 500;
type
  Tacka = record
    x, y : real
  end;
Poligon = record
  N : integer;
  A : array [1 .. MaxN] of Tacka
end;
```

(a) Napisati Pascal funkciju koja računa obim datog poligona.

(b) Napisati Pascal funkciju koja računa površinu datog poligona koristeći sledeću formulu:

$$P = \frac{1}{2} \cdot \left| \sum_{i=1}^{N} (x_i y_{i\oplus 1} - x_{i\oplus 1} y_i) \right|,$$

gde je sa $\oplus 1$ označeno "cikličko" uvećavanje za 1 kod koga je $N \oplus 1 = 1$, dok je $k \oplus 1 = k+1$ za $k \in \{1, \ldots, N-1\}$.

(c) Napisati Pascal funkciju koja računa dijametar poligona (dijametar figure je najveće rastojanje koje postižu dve tačke te figure).

(d) Napisati proceduru `procedure BoundingBox(P : Poligon; var B : Poligon)` koja za dati prost poligon `P` određuje njegov *bounding box*, što je najmanji pravougaonik koji sadrži poligon `P`, a čije strane su paralelne koordinatnim osama.

3.13. Napisati Pascal program koji od korisnika učitava pozitivan ceo broj n, potom n razlomaka $a_1, \ldots, a_n$ i uređuje i štampa date razlomke po veličini, od najmanjeg ka najvećem (sortiranje razlomaka).

3.5 Datoteke

Datoteka je proizvoljno dugačak niz podataka istog tipa koji se nalazi u spoljašnjoj memoriji. Prema načinu pristupa njenim elementima, datoteke se dele na dve velike organizacione grupe: sekvencijalne datoteke i rasute datoteke. Podacima iz sekvencijalne datoteke se pristupa redom, od prvog ka poslednjem, bez mogućnosti preskakanja, dok se podacima iz rasute datoteke može pristupati proizvoljnim redosledom. Programski jezik Pascal u svojoj osnovnoj verziji podržava samo sekvencijalne datoteke, mada skoro sve implementacije nude dodatne pakete za rad sa raznim drugim tipovima datoteka.

Podacima iz neke datoteke se iz Pascal programa pristupa preko promenljive tipa datoteka čija deklaracija izgleda ovako:

```
var
    f : file of ⟨proizvoljan tip⟩;
```

U programskom jeziku Pascal se podaci iz datoteke mogu ili samo čitati, ili je moguć samo upis u datoteku. Pre početka rada sa datotekom potrebno je vezati promenljivu za neku fizičku datoteku (koja se najčešće nalazi na hard disku) i naglasiti programu da li će podaci biti samo čitani iz datoteke ili samo upisivani u datoteku. Dodela fizičke datoteke promenljivoj nije standardizovana, tako da svaka implementacija nudi svoje rešenje. Implementacija Pascala koju mi koristimo ima posebnu komandu `assign` koja vezuje datotečku promenljivu za neku konkretnu datoteku. Operacije za rad sa datotekama su date u Tabeli 3.1.

Operacija	Značenje	Primer
`assign`	dodela fizičke datoteke promenljivoj	`assign(f, 'p.dat');`
`reset`	priprema datoteke za čitanje	`reset(f);`
`rewrite`	priprema datoteke za pisanje (ukoliko datoteka nije prazna, stari sadržaj se briše)	`rewrite(f);`
`write`	upis u datoteku	`write(f, slog);`
`read`	čitanje podatka iz datoteke	`read(f, slog);`
`close`	zatvaranje datoteke	`close(f);`
`eof`	proverava da li je su pročitani svi podaci iz datoteke	`if eof(f) then ...`

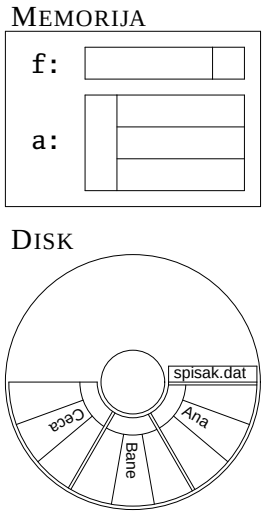
Tabela 3.1: Operacije sa datotekama

☞ *Upis i čitanje podataka u/iz datoteke se obavlja naredbama* `read` *odnosno* `write`. *Naredbe* `readln` *i* `writeln` *se ne mogu koristiti za rad sa datotekama tipa* `file of T`.

Evo jednog tipičnog programa koji čita podatke iz neke datoteke. Prvo se naredbom `assign` promenljiva `f` veže za konkretnu datoteku na disku. Komanda `reset(f)` sprema datoteku za čitanje. U `while` ciklusu koji sledi testom `eof(f)` proveravamo da li smo stigli do kraja datoteke i ako nismo, prelazimo na izvršavanje tela ciklusa. U telu ciklusa se naredbom `read(f, a)` iz datoteke `f` pročita sledeći podatak, smesti u promenljivu `a` i dalje obrađuje.

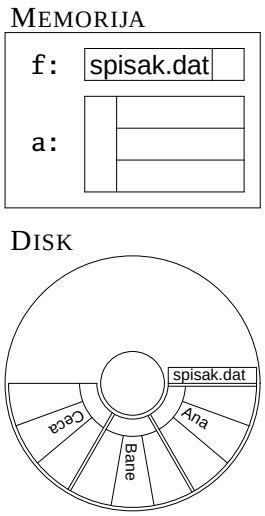
```
program CitamIzDat;
type
  Osoba = record ... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  reset(f);
  while not eof(f) do begin
    read(f, a);
    ....
  end;
  close(f)
end.
```

Pogledajmo na jednom primeru kako ovaj program radi. Pretpostavimo da na disku imamo datoteku `spisak.dat` tipa `file of Osoba`. U radnoj memoriji računara rezerviše se prostor za dve promenljive: za promenljivu `f` tipa `file of Osoba` i za promenljivu `a` tipa `Osoba`.



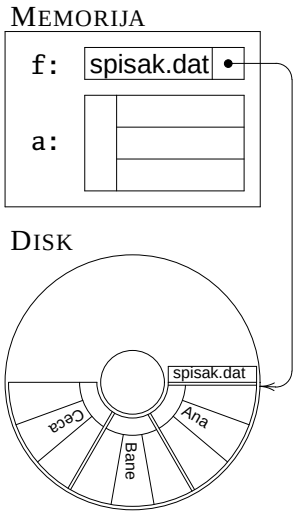

```
program CitamIzDat;
type
  Osoba = record ... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  reset(f);
  while not eof(f) do begin
    read(f, a);
    ....
  end;
  close(f)
end.
```

Naredba `assign` veže promenljivu `f` za datoteku `spisak.dat`.



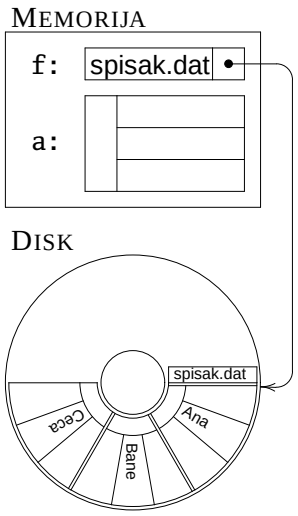

```
program CitamIzDat;
type
  Osoba = record ... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  reset(f);
  while not eof(f) do begin
    read(f, a);
    ....
  end;
  close(f)
end.
```

Naredba `reset` pripremi datoteku za čitanje i pozicionira pokazivač na početak datoteke. Pokazivač je jedna strelica koja pokazuje na sledeći podatak u datoteci koji je spreman za čitanje.



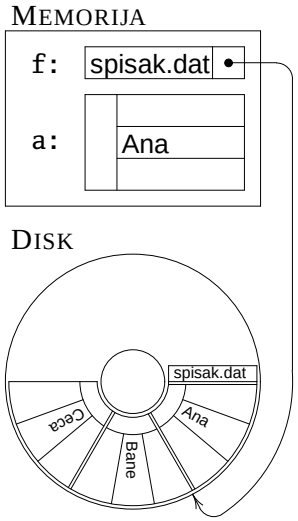

```
program CitamIzDat;
type
  Osoba = record ... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  reset(f);
  while not eof(f) do begin
    read(f, a);
    ....
  end;
  close(f)
end.
```

Test `eof(f)` proveri da li smo stigli do kraja datoteke. Pošto to nije slučaj, program će izvršiti telo petlje.



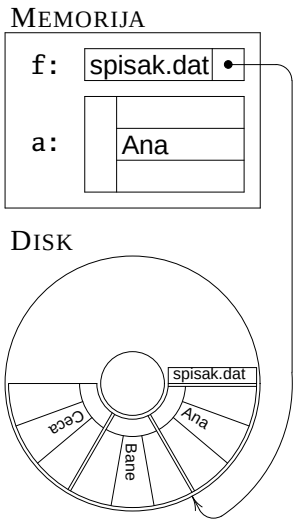

```
program CitamIzDat;
type
  Osoba = record ... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  reset(f);
  while not eof(f) do begin
    read(f, a);
    ....
  end;
  close(f)
end.
```

Naredba `read(f, a)` pročita iz datoteke `f` podatak i upiše ga u promenljivu `a`. Da bi ova naredba mogla da se izvrši, tip datoteke i tip promenljive moraju biti usaglašeni: ako je datoteka tipa `file of` *T*, onda i promenljiva mora imati tip *T*. U ovom slučaju, datoteka je tipa `file of Osoba`, promenljiva `a` je tipa `Osoba` i naredba može da se izvrši. Pri tome se pokazivač datoteke pomeri na naredni podatak, tako da će sledeći poziv naredbe `read(f, a)` iz datoteke pročitati sledeći podatak.



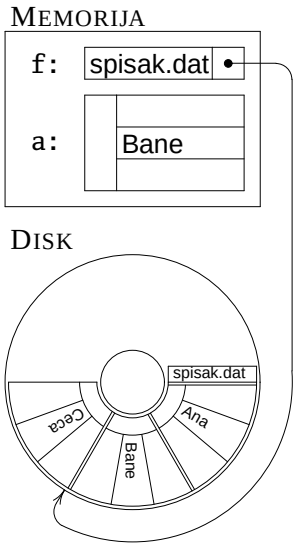

```
program CitamIzDat;
type
  Osoba = record ... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  reset(f);
  while not eof(f) do begin
    read(f, a);
    ....
  end;
  close(f)
end.
```

Kada se izvrši telo petlje, vraćamo se na početak i ponovo proveravamo da li smo stigli do kraja datoteke. Nismo. Zato će ponovo biti izvršeno telo petlje.



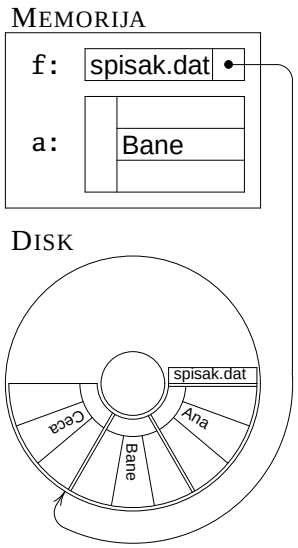

```
program CitamIzDat;
type
  Osoba = record ... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  reset(f);
  while not eof(f) do begin
    read(f, a);
    ....
  end;
  close(f)
end.
```

Naredba `read(f, a)` čita iz datoteke sledeći podatak i smešta ga u promenljivu `a`.



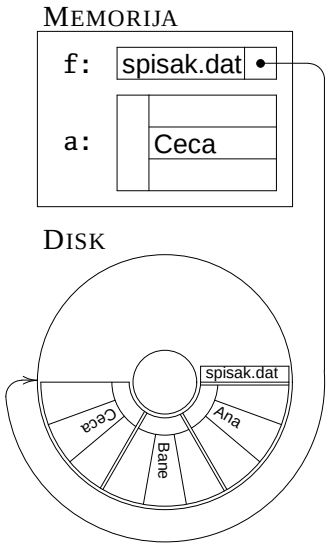

```
program CitamIzDat;
type
  Osoba = record ... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  reset(f);
  while not eof(f) do begin
    read(f, a);
    ....
  end;
  close(f)
end.
```

Kada se izvrši telo petlje, još jednom se vraćamo na početak i proveravamo da li smo stigli do kraja datoteke. Nismo. Još jednom će biti izvršeno telo petlje.



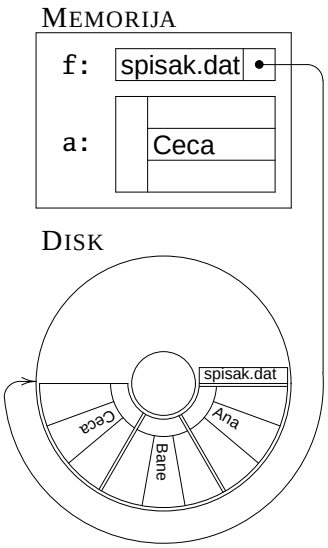

```
program CitamIzDat;
type
  Osoba = record ... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  reset(f);
  while not eof(f) do begin
    read(f, a);
    ....
  end;
  close(f)
end.
```

Naredba `read(f, a)` čita iz datoteke sledeći podatak i smešta ga u promenljivu `a`.



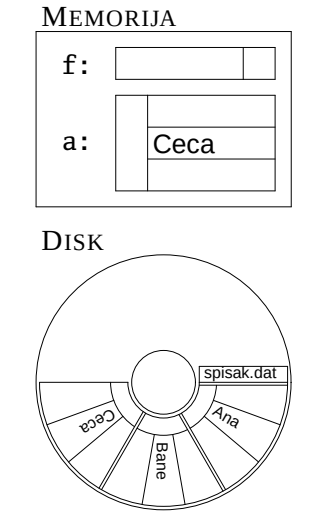

```
program CitamIzDat;
type
  Osoba = record ... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  reset(f);
  while not eof(f) do begin
    read(f, a);
    ....
  end;
  close(f)
end.
```

Kada se izvrši telo petlje, vraćamo se na početak i proveravamo da li smo stigli do kraja datoteke. JESMO! Ciklus se završava.




```
program CitamIzDat;
type
  Osoba = record ... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  reset(f);
  while not eof(f) do begin
    read(f, a);
    ....
  end;
  close(f)
end.
```

Naredba `close(f)` zatvara datoteku i tako raskida vezu promenljive `f` i datoteke `spisak.dat`.

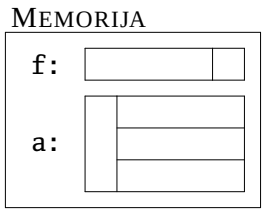


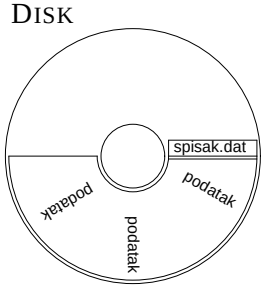

```
program CitamIzDat;
type
  Osoba = record ... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  reset(f);
  while not eof(f) do begin
    read(f, a);
    ....
  end;
  close(f)
end.
```

Tipičan program koji upisuje podatke u neku datoteke je dat pored. Prvo se naredbom `assign` promenljiva `f` veže za neku datoteku na disku. Komanda `rewrite(f)` sprema datoteku za pisanje u sledećem smislu: ako na disku postoji datoteka čije ime je navedeno u naredbi `assign`, računar izbriše staru datoteku i napravi praznu datoteku sa datim imenom; ako na disku ne postoji datoteka čije ime je navedeno u naredbi `assign`, računar napravi praznu datoteku sa datim imenom. Potom se, najčešće u nekom ciklusu, od korisnika učitavaju podaci i naredbom `write` upisuju u datoteku jedan za drugim.

```
program PisemUDat;
type
  Osoba = record .... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  rewrite(f);
  while not KrajUnosa do begin
    ... (* formiranje sloga a *)
    write(f, a)
  end;
  close(f)
end.
```

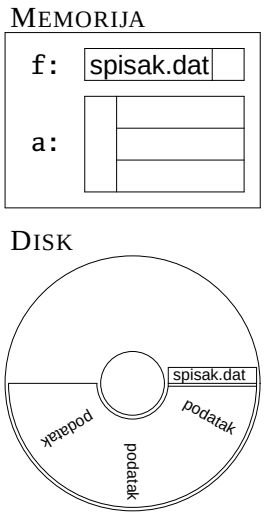
Pogledajmo na jednom primeru kako program radi. U radnoj memoriji računara rezerviše se prostor za dve promenljive: za promenljivu `f` tipa `file of Osoba` i za promenljivu `a` tipa `Osoba`. Pretpostavimo da na disku postoji datoteka `spisak.dat` koja sadrži neke podatke.





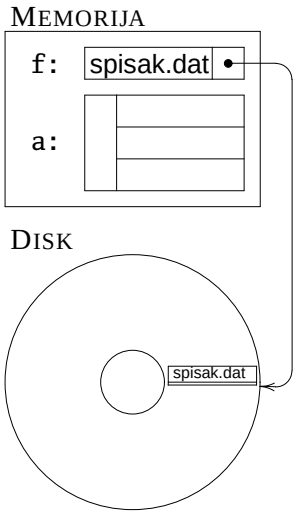

```
program PisemUDat;
type
  Osoba = record .... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  rewrite(f);
  while not KrajUnosa do begin
    ... (* formiranje sloga a *)
    write(f, a)
  end;
  close(f)
end.
```

Naredba `assign` veže promenljivu `f` za datoteku `spisak.dat`.



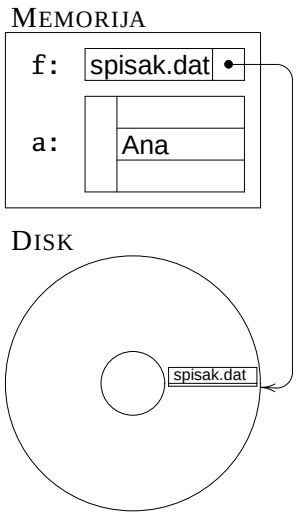

```
program PisemUDat;
type
  Osoba = record .... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  rewrite(f);
  while not KrajUnosa do begin
    ... (* formiranje sloga a *)
    write(f, a)
  end;
  close(f)
end.
```

Naredba `rewrite` pripremi datoteku za pisanje tako što obriše stari sadržaj datoteke `spisak.dat` i pozicionira pokazivač na početak prazne datoteke.



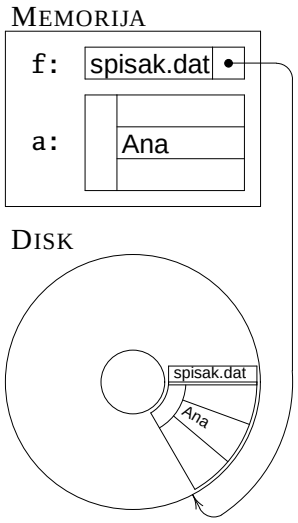

```
program PisemUDat;
type
  Osoba = record .... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  rewrite(f);
  while not KrajUnosa do begin
    ... (* formiranje sloga a *)
    write(f, a)
  end;
  close(f)
end.
```

Program potom formira slog a tako što od korisnika učita podatke o nekoj osobi.



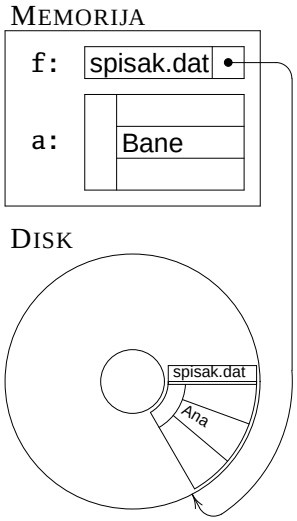

```
program PisemUDat;
type
  Osoba = record .... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  rewrite(f);
  while not KrajUnosa do begin
    ... (* formiranje sloga a *)
    write(f, a)
  end;
  close(f)
end.
```

A onda se naredbom `write(f, a)` slog a dopiše na kraj datoteke `f`.



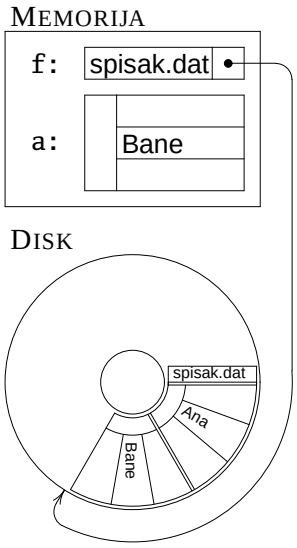

```
program PisemUDat;
type
  Osoba = record .... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  rewrite(f);
  while not KrajUnosa do begin
    ... (* formiranje sloga a *)
    write(f, a)
  end;
  close(f)
end.
```

Recimo da korisnik želi da unosi još podataka. Program formira slog a tako što od korisnika učita nove podatke o nekoj osobi.



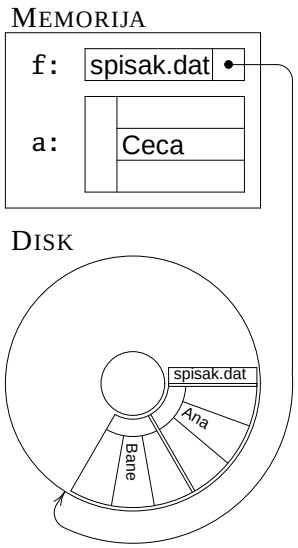

```
program PisemUDat;
type
  Osoba = record .... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  rewrite(f);
  while not KrajUnosa do begin
    ... (* formiranje sloga a *)
    write(f, a)
  end;
  close(f)
end.
```

Pa se naredbom `write(f, a)` slog a dopiše na kraj datoteke f.



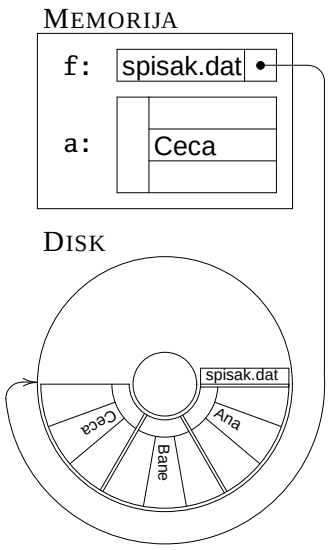

```
program PisemUDat;
type
  Osoba = record .... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  rewrite(f);
  while not KrajUnosa do begin
    ... (* formiranje sloga a *)
    write(f, a)
  end;
  close(f)
end.
```

Pretpostavimo da korisnik želi da unese još jedan podatak u datoteku. Program na isti način formira slog a.



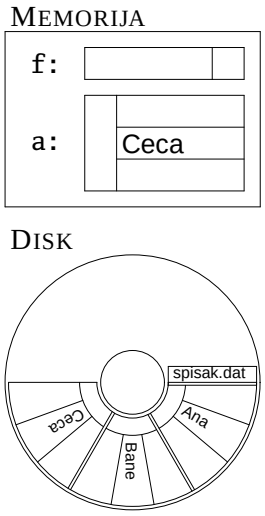

```
program PisemUDat;
type
  Osoba = record .... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  rewrite(f);
  while not KrajUnosa do begin
    ... (* formiranje sloga a *)
    write(f, a)
  end;
  close(f)
end.
```

I naredbom `write(f, a)` slog a dopiše na kraj datoteke `f`.




```
program PisemUDat;
type
  Osoba = record .... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  rewrite(f);
  while not KrajUnosa do begin
    ... (* formiranje sloga a *)
    write(f, a)
  end;
  close(f)
end.
```

Na kraju kada korisnik više ne želi da unosi podatke, naredba `close(f)` zatvara datoteku i tako raskida vezu promenljive `f` i datoteke `spisak.dat`.




```
program PisemUDat;
type
  Osoba = record .... end;
var
  f : file of Osoba;
  a : Osoba;
begin
  assign(f, 'spisak.dat');
  rewrite(f);
  while not KrajUnosa do begin
    ... (* formiranje sloga a *)
    write(f, a)
  end;
  close(f)
end.
```

Elementi datoteke mogu biti proizvoljnog tipa, ali su to najčešće prosti tipovi ili slogovi. Naravno, niko nam ne brani da u datoteku upisujemo nizove brojeva ako nam je to potrebno, kao u sledećem primeru:

```
program PrimerDatSaNiz;
const
  Max = 100;
type
  Niz = array [1 .. Max] of real;
var
  f : file of Niz;
  s : Niz;
begin
  assign(f, 'c:merenja.dat');
  rewrite(f);
  ...
  write(f, s);
  ...
  close(f);
end.
```

Datoteke mogu da budu argumenti procedura i funkcija, ali *datoteka ne može biti rezultat funkcije.* Ako se datoteke javljaju kao argumenti procedura i funkcija, postoji jedno važno ograničenje:

☞ *Datoteke se u potprogram* uvek *prenose kao* `var` *argumenti.*

Razlog za ovo ograničenje je jednostavan: datoteke se nalaze u spoljašnjoj memoriji i prevodilac nema načina da obezbedi da se u potprogram unese kopija datoteke. Svaka izmena koju procedura izvrši nad datotekom biće vidljiva iz glavnog programa. Na primer, funkcija koja računa broj elemenata neke datoteke tipa `file of Osoba` može da se napiše ovako:

```
type
  Osoba = record ... end;
  Spisak = file of Osoba;

function DuzinaSpiska(var f : Spisak) : integer;
var
  a : Osoba;
  n : integer;
begin
  n := 0;
  reset(f);
  while not eof(f) do begin
    read(f, a);
    n := n + 1
  end;
  close(f);
  DuzinaSpiska := n
end;
```

Kviz.

1. Sa sledećim deklaracijama na snazi:

```
type
  JmbgType = array [1 .. 13] of integer;
  Osoba = record
    ime, prezime : string;
    rodj : record
      dd, mm, gggg : integer
    end;
    jmbg : JmbgType;
    brak : Boolean;
    supruznik : JmbgType
  end;
```

```
var
  a, b : Osoba;
  c : char;
  s : string;
  f : file of Osoba;
  t : text;
```

označiti korektne naredbe:

- ☐ `assign(f, 'c:\spisak.dat');`
- ☐ `assign(f, c:\spisak.dat);`
- ☐ `assign(f, s);`
- ☐ `assign(t, 'spisak.txt');`
- ☐ `reset('c:\spisak.dat');`
- ☐ `reset(f);`
- ☐ `rewrite(s);`
- ☐ `rewrite(t);`
- ☐ `read(f, a);`
- ☐ `readln(f);`
- ☐ `write(f, a);`
- ☐ `writeln(f, s);`
- ☐ `writeln(f, 'kraj spiska');`
- ☐ `writeln('kraj spiska');`
- ☐ `writeln(t, b);`
- ☐ `writeln(t, 2 + 2);`
- ☐ `readln(t, a);`
- ☐ `readln(t, a.ime, a.prezime);`
- ☐ `read(f, a.prezime);`
- ☐ `if a.supruznik = b.jmbg then write('!');`
- ☐ `a.ime := 'Petar';`
- ☐ `a.rodj := '29.02.2010.';`
- ☐ `b := a;`
- ☐ `b.ime := a.ime;`
- ☐ `b.jmbg := a.jmbg;`
- ☐ `b.rodj := a.rodj;`
- ☐ `a.jmbg[4] := 12;`
- ☐ `a.jmbg[20] := 9;`

Zadaci.

3.14. Data je sledeća definicija tipa `Osoba`:

```
type
  Datum = record
    dan, mesec, godina : integer
  end;
  Osoba = record
    ime, prezime : string;
    rodjen : Datum;
    pol : integer {1 = muski, 2 = zenski}
  end;
```

(*a*) Napisati program koji formira datoteku `osobe.dat` tipa `file of Osoba`.

(*b*) U datoteci `osobe.dat` tipa `file of Osoba` se nalaze podaci o osobama. Ne znamo unapred koliko u ovoj datoteci ima podataka. Napisati Pascal program koji od korisnika učitava tekuću godinu, potom učitava podatke iz ove datoteke i utvrdjuje i štampa broj muških osoba koje imaju $\geqslant 18$ godina.

(*c*) Napisati program koji učitava u niz slogove datoteke tipa `Osoba` (kojih ima najviše 1000), prema zahtevu korisnika sortira niz po prezimenu ili datumu rođenja, i tako sortiran niz upisuje u novu datoteku.

3.15. Na disku se nalaze dve datoteke tipa `file of Artikl` gde je

```
type
  Artikl = record
    naziv : string;
    sifra : integer;
    kolicina : integer;
    cenaPoKom : real
  end;
```

Jedna od njih, `magacin.dat`, sadrži informacije o stanju u magacinu, a druga, `promene.dat`, informacije o promenama u nazivu, količini ili ceni artikla. Napisati Pascal program koji na osnovu informacija o promenama ažurira datoteku `magacin.dat`. Imati na umu da je svaki artikl jednoznačno određen svojom šifrom koja je jedino obeležje proizvoda koje ne može da se menja.

3.16. Data je sledeća struktura podataka

```
const
  MaxBrPredmeta = 10;
type
  Ocene = array [1 .. MaxBrPredmeta] of integer;
  Ucenik = record
    ime, prezime : string;
    ocena : Ocene;
    izost : record
      opravdani, neopravdani : integer
    end
  end;
```

- Napisati Pascal program koji formira datoteku `razred.dat` tipa `file of Ucenik` čiji sadržaj su podaci o učenicima tvog razreda.
- Napisati Pascal program koji generiše spisak svih učenika sa više od 5 neopravdanih izostanaka.
- Napisati Pascal program koji generiše spisak učenika sortiran prema proseku ili izostancima (od korisnika učitati kriterijum sortiranja).
- Napisati Pascal program koji za svakog učenika određuje kojoj kategoriji pripada, prema sledećim pravilima: kategoriju A čine učenici koji nemaju neopravdane izostanke i po uspehu spadaju u gornjih 25%; kategoriju C čine učenici koji imaju 6 ili više neopravdanih izostanaka, ili po uspehu spadaju u donjih 25%; kategoriju B čine svi ostali učenici.
- Napisati Pascal program koji sortira datoteku tipa `file of Ucenik` prema prezimenu.
- Napisati Pascal program koji spaja dve datoteke tipa `file of Ucenik` sortirane po prezimenu u treću, koja takođe treba da bude sortirana po prezimenu.

3.17. Napisati Pascal program koji u datoj tekstualnoj datoteci traži sve reči koje odgovaraju datom uzorku (engl. *pattern*). Uzorak je string čija slova imaju sledeće značenje:

`?`	džoker umesto koga može da se nalazi bilo koji znak
`^`	džoker umesto koga može da se nalazi bilo koje veliko slovo
`_`	džoker umesto koga može da se nalazi bilo koje malo slovo.

Za svaku nađenu reč ispisuje se ta reč, kao i red u kome se pojavljuje.

3.18. Napisati program koji implementira malu “bazu podataka” koja sadrži informacije o prijateljima:

```
type
  Prijatelj = record
    ime, prezime : string;
    rodjendan : record
      dan, mesec, godina : integer
    end;
    brTel : array [1 .. 10] of 0..9;
    adresa : string
  end;
```

“Baza” podataka poznaje sledeće operacije:

`open`	otvara bazu: učitava u niz sve slogove iz datoteke
`close`	zatvara bazu: upisuje u datoteku slogove iz internog niza
`add`	dodaje novi slog
`remove`	izbacuje slog iz niza
`list`	ispisuje sve slogove datoteke
`today`	ispisuje imena svih prijatelja čiji rođendan je danas
`count`	utvrđuje broj elemenata u “bazi”
`@name`	sortira “bazu” po prezimenu
`@date`	sortira “bazu” po rođendanu
`@adr`	sortira “bazu” po adresi
`?name`	ispisuje podatke za osobu čije ime ili prezime je navedeno ime ili prezime ne mora biti navedeno u celosti, već se može navesti i uzorak (pattern; prethodni zadatak)
`?date`	ispisuje podatke za osobu čiji rođendan je naveden
`?tel`	ispisuje podatke za osobu čiji telefon je naveden
`?adr`	ispisuje podatke za osobu čija adresa je navedena

Glava 4

Matrice

Matrica je pravougaona šema brojeva ili nekih drugih objekata. Dimenzije matrice zovemo još i *format matrice*. Na primer,

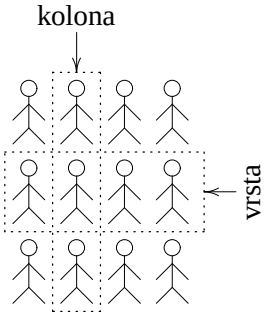
Matrica brojeva formata 3×5:

$$\begin{bmatrix} 0.12 & 3.35 & 12.47 & -1.12 & 9.96 \\ -2.44 & 0.00 & 9.67 & 12.36 & 3.37 \\ 9.85 & 3.31 & -1.00 & -2.25 & 2.78 \end{bmatrix}$$

Matrica slova formata 5×3:

$$\begin{bmatrix} a & b & c \\ f & e & d \\ g & h & i \\ l & k & j \\ m & n & o \end{bmatrix}$$

Matrica formata $m \times n$ ima m vrsta i n kolona. Na slici dole levo prikazana je matrica (čovečuljaka) formata 3×4.



$$\begin{bmatrix} 1 & 2 & 0 & 4 & 5 \\ 3 & 1 & 0 & 5 & 2 \\ 1 & 0 & 9 & 7 & 1 \\ 6 & 0 & 2 & 5 & 3 \\ 7 & 4 & 1 & 0 & 1 \end{bmatrix}$$

← sporedna dijagonala

← glavna dijagonala

Matrica kod koje su obe dimenzije jednake zove se *kvadratna matrica*. Kvadratna matrica ima dve dijagonale, *glavnu* i *sporednu*.

☞ *Elementi matrice na glavnoj dijagonali imaju oblik* a_{ii}*, dok elementi na sporednoj dijagonali imaju oblik* $a_{i,(n+1)-i}$.

4.1 Matrice u Pascalu

U programskom jeziku Pascal matrice se deklarišu kao višedimenzioni nizovi. Promenljiva koja sadrži matricu deklariše se na sledeći način:

```
var
    m : array [⟨nabrojivi tip1⟩, ⟨nabrojivi tip2⟩] of ⟨proizvoljan tip⟩;
```

Na primer:

```
const
  MaxLen = 50;

type
  Niz = array [1 .. MaxLen] of integer;
  Osoba = record
    ime : string;
    d, m, g : integer
  end;

var
  a : array [-5 .. 5, 0 .. 10] of Niz;
  b : array ['a' .. 'z', -3 .. 3] of Osoba;
  c : array [boolean, char] of real;
```

Elementu matrice se pristupa tako što se u zagradi iza imena navedu dva indeksa razdvojena zarezom, kao u sledećim primerima:

```
c[false, '@'] := 3.1415;
b['a', -2].ime := 'Petar Petrovic-Njegos';
a[-1, 9][3] := 6;
```

Iako se matrice u memoriju smeštaju tako da odgovarajuće kućice idu jedna za drugom:

```
var  m : array [1 .. 3, -1 .. 1] of integer;
```

m[1,-1]	m[1,0]	m[1,1]	m[2,-1]	m[2,0]	m[2,1]	m[3,-1]	m[3,0]	m[3,1]

nama je lakše da o njima razmišljamo kao da su smeštene u obliku tabele što ćemo rado i činiti:

```
var  m : array [1 .. 3, -1 .. 1] of integer;
```

m[1,-1]	m[1,0]	m[1,1]
m[2,-1]	m[2,0]	m[2,1]
m[3,-1]	m[3,0]	m[3,1]

Matrica se od korisnika učitava tako što se prvo učita njen format, a onda se učitavaju elementi, jedan po jedan. Na primer, učitavanje matrice koja je deklarisana sa

```
const
  MaxN = 50;
var
  a : array [1..MaxN, 1..MaxN] of real;
  m, n : integer; {format matrice a}
```

se može realizovati ovako:

```
repeat
  write('Format matrice m x n -> ');
  readln(m, n)
until (1 <= m) and (m <= MaxN) and (1 <= n) and (n <= MaxN);

writeln('Elementi matrice:');
for i := 1 to m do
  for j := 1 to n do begin
    write('a[', i, ',', j, '] -> ');
    readln(a[i, j])
  end;
```

Ispis matrice možemo realizovati ili tako da svaki element matrice ispišemo u novi red:

```
for i := 1 to m do
  for j := 1 to n do
    writeln('a[', i, ',', j, '] = ', a[i, j]:10:2);
```

ili, ukoliko se radi o matrici manjeg formata, u obliku tabele:

```
for i := 1 to m do begin
  for j := 1 to n do write(a[i, j]:10:2);
  writeln
end;
```

Primer. Trag kvadratne matrice $A = [a_{ij}]$ formata $n \times n$ je zbir elemenata glavne dijagonale, dakle, broj $\sum_{i=1}^{n} a_{ii}$. Napisati Pascal program koji učitava kvadratnu matricu i računa i ispisuje njen trag.

```
program TragKvMat;
const
  MaxN = 50;
var
  i, j, n : integer;
  a : array [1..MaxN, 1..MaxN] of real;
  tr : real;
begin
  {ucitavanje matrice}
  readln(n);
  for i := 1 to n do
    for j := 1 to n do
      readln(a[i, j]);

  {racunanje traga}
  tr := 0;
  for i := 1 to n do
    tr := tr + a[i, i];
  writeln(tr)
end.
```

Zadaci.

4.1. Napisati program koji za datu kvadratnu matricu računa vrednost najvećeg elementa na sporednoj dijagonali.

4.2. Matrica se množi brojem tako što se svaki element matrice pomnoži brojem. Na primer:

$$3 \cdot \begin{bmatrix} 1 & 2 & 1 & 3 \\ 0 & 1 & 1 & 1 \\ 2 & 1 & 3 & 0 \end{bmatrix} = \begin{bmatrix} 3 & 6 & 3 & 9 \\ 0 & 3 & 3 & 3 \\ 6 & 3 & 9 & 0 \end{bmatrix}.$$

Napisati Pascal program koji od korisnika učitava matricu i broj i potom računa i štampa proizvod broja i matrice.

4.3. Napisati program koji za datu kvadratnu matricu računa sumu elemenata ispod glavne dijagonale.

4.4. Napisati program koji od korisnika učitava niz brojeva $a_1, \ldots, a_n$ gde je $1 \leqslant n \leqslant 50$, i onda formira i ispisuje matricu B formata $n \times n$ čija prva vrsta je $a_1, \ldots, a_n$, a svaka naredna vrsta se dobija od prethodne cikličkim pomeranjem njenih elemenata za jedno mesto u levo.

4.5. Kvadratna matrica A je *strogo dijagonalno dominantna* ako je

$$|a_{ii}| > \sum_{\substack{j=1\ldots n \\ j\neq i}} |a_{ij}|$$

za sve i. Napisati Pascal program koji utvrđuje da li je data kvadratna matrica strogo dijagonalno dominantna.

†**4.6.** Napisati Pascal program koji od korisnika učitava matricu formata $m \times n$, gde je $2 \leqslant m,n \leqslant 50$, za koju se zna da je popunjena je na neki način brojevima 1, 2, 4, 8, i potom štampa broji podmatrica formata 2×2 kod kojih su svi elementi različiti.

4.7. Školski dnevnik nije ništa drugo do jedna matrica kod koje vrste odgovaraju učenicima jednog odeljenja, a kolone predmetima. U preseku i-te vrste i j-te kolone nalazi se broj koji predstavlja ocenu koja je i-tom učeniku zaključena iz j-tog predmeta. Napisati Pascal program kod iz tekstualne datoteke `ocene.txt` učitava podatke o učenicima jednog odeljenja i njihovim ocenama, a onda računa prosek za svakog učenika i svaki predmet, i njih upisuje u tekstualnu datoteku `proseci.txt`, svaki prosek u novom redu. Datoteka `ocene.txt` ima sledeću strukturu: u prvom redu se nalaze dva pozitivna cela broja N (broj učenika u odeljenju) i K (broj predmeta) koji su razdvojeni jednim razmakom. Nakon prvog reda, u datoteci sledi još N redova, a svaki od njih sadrži tačno K brojeva koji su svi međusobno razdvojeni po jednim razmakom (ocene odgovarajućeg učenika).

4.8. Napisati program koji za datu kvadratnu matricu reda n (što znači da je matrica formata $n \times n$) računa niz $d_0, \ldots, d_{n-1}$, gde je: d_0 =suma elemenata na glavnoj dijagonali, d_1 =suma elemenata na prvoj dijagonalnoj paraleli donjeg trougla matrice, d_2 =suma elemenata na drugoj dijagonalnoj paraleli donjeg trougla matrice, itd.

4.9. Proizvod matrice A formata $m \times n$ i vektora b dimenzije n (a to je matrica formata $n \times 1$) je vektor c dimenzije m:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}$$

čiji elementi se računaju po formuli

$$c_i = \sum_{j=1}^{n} a_{ij} b_j.$$

Napisati Pascal program koji od korisnika učitava matricu i vektor i ako su kompatibilni računa i štampa vektor koji je njihov proizvod.

4.10. Element neke matrice se zove *sedlasta tačka* ako je on istovremeno najmanji u svojoj vrsti i najveći u svojoj koloni (sedlasta tačka tipa α), ili obrnuto, najmanji u svojoj koloni i najveći u svojoj vrsti (sedlasta tačka tipa β). Napisati program koji od korisnika učitava realnu matricu A formata $n \times n$ i računa i štampa matricu S istog formata koja je definisana na sledeći način:

$$s_{ij} = \begin{cases} 1, & a_{ij} \text{ je sedlasta tačka tipa } \alpha, \\ -1, & a_{ij} \text{ je sedlasta tačka tipa } \beta, \\ 0, & \text{inače.} \end{cases}$$

4.11. Data je matrica A formata $m \times n$ čija i-ta vrsta sadrži ocene koje je i-ti gimnastičar na nekom takmičenju dobio od svakog od n sudija. Napisati program koji formira niz $c_1, \ldots, c_m$ tako da c_j bude redni broj gimnastičara koji se plasirao na j-to mesto.

4.12. Matrica $n \times n$ celih brojeva je magični kvadrat ako su zbirovi svih vrsta i svih kolona isti. Magični kvadrat je *jak magični kvadrat* ako su elementi matrice svi brojevi iz skupa $\{1, 2, \ldots, n^2\}$. Napisati program koji proverava da li je data matrica magični kvadrat, i ako jeste, da li je jak magični kvadrat.

4.13. "Zarotirati" kvadratnu matricu za 90°.

4.14. Upisati brojeve 1, 2, ..., n^2 u kvadratnu matricu $n \times n$ po kolonama. Na primer, za $n = 4$ dobija se

$$\begin{array}{cccc} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{array}$$

4.15. Upisati brojeve 1, 2, ..., n^2 spiralno u kvadratnu matricu $n \times n$. Na primer, za $n = 4$ dobija se

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 12 & 13 & 14 & 5 \\ 11 & 16 & 15 & 6 \\ 10 & 9 & 8 & 7 \end{array}$$

4.16. Napisati Pascal program koji od korisnika učitava matricu realnih brojeva, sortira je i potom ispisuje sortiranu matricu. Matrica je sortirana ako su njeni elementi, kada se čitaju red po red, poređani od manjih vrednosti ka većim. Na primer:

$$\begin{bmatrix} 4 & 1 & 3 \\ 7 & 5 & 9 \\ 6 & 2 & 8 \end{bmatrix} \mapsto \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

†**4.17.** *Latinski kvadrat reda n* je kvadratna matrica $S = [s_{ij}]$ formata $n \times n$ čiji elementi su iz skupa $\{1,2,\ldots,n\}$ i koja ima osobinu da su joj u svakoj vrsti svi elementi različiti i da su joj u svakoj koloni svi elementi različiti.

(a) Napisati Pascal program koji ispituje da li je data kvadratna matrica latinski kvadrat.

(b) Za latinski kvadrat S kažemo da je *dvostruko standardan* ako su mu elementi prve vrste i prve kolone poređani po veličini. Na primer,

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{bmatrix}$$

Svaki latinski kvadrat se premeštanjem vrsta i kolona može dovesti u dvostruko standardan oblik. Napisati program koji dati latinski kvadrat dovodi u dvostruko standardan oblik.

(c) Za latinske kvadrate $S = [s_{ij}]$ i $T = [t_{ij}]$, oba reda n, kažemo da su *ortogonalni* ako je $\{(s_{ij}, t_{ij}) : 1 \leqslant i, j \leqslant n\} = \{(i, j) : 1 \leqslant i, j \leqslant n\}$. Na primer, latinski kvadrati

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{bmatrix} \quad \text{i} \quad \begin{bmatrix} 3 & 1 & 2 \\ 2 & 3 & 1 \\ 1 & 2 & 3 \end{bmatrix}$$

su ortogonalni, dok

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \end{bmatrix} \quad \text{i} \quad \begin{bmatrix} 2 & 3 & 1 \\ 1 & 2 & 3 \\ 3 & 1 & 2 \end{bmatrix}$$

to nisu. Napisati Pascal program koji proverava da li su dva latinska kvadrata istog reda ortogonalni.

(d) Za latinske kvadrate $S = [s_{ij}]$ i $T = [t_{ij}]$, oba reda n, kažemo da su *izotopni* ako postoji bijekcija α skupa $\{1,2,\ldots,n\}$ takva da je $t_{ij} = \alpha(s_{ij})$ za sve i i j. Na primer, sledeća dva latinska kvadrata su izotopni

$$\begin{bmatrix} 4 & 1 & 3 & 2 \\ 3 & 2 & 1 & 4 \\ 1 & 4 & 2 & 3 \\ 2 & 3 & 4 & 1 \end{bmatrix} \sim \begin{bmatrix} 3 & 4 & 2 & 1 \\ 2 & 1 & 4 & 3 \\ 4 & 3 & 1 & 2 \\ 1 & 2 & 3 & 4 \end{bmatrix},$$

jer permutacije

$$\alpha = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 2 & 3 \end{pmatrix}$$

daje

$$\begin{bmatrix} \alpha(4) & \alpha(1) & \alpha(3) & \alpha(2) \\ \alpha(3) & \alpha(2) & \alpha(1) & \alpha(4) \\ \alpha(1) & \alpha(4) & \alpha(2) & \alpha(3) \\ \alpha(2) & \alpha(3) & \alpha(4) & \alpha(1) \end{bmatrix} = \begin{bmatrix} 3 & 4 & 2 & 1 \\ 2 & 1 & 4 & 3 \\ 4 & 3 & 1 & 2 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

Napisati Pascal program koji za latinske kvadrate S i T utvrđuje da li su izotopni.

†**4.18.** Na izborima za parlament Republike Srbije struktura parlamenta se utvrđuje po principu najvećih količnika (Dontov proporcionalni sistem; dobio ime po belgijskom matematičaru Viktoru Dontu) koji radi na sledeći način. Neka parlament ima N mesta, neka je na parlamentarnim izborima učestvovalo K stranaka/koalicija, i neka su one osvojile $g_1, g_2, \ldots, g_K$ glasova, redom. Tada se formira tabela sa N vrsta i K kolona koja se popuni tako što se u j-tu vrstu upišu brojevi

$$\frac{g_1}{j}, \frac{g_2}{j}, \ldots, \frac{g_K}{j},$$

što ne moraju nužno biti celi brojevi. Potom se u tako formiranoj tabeli brojeva zaokruži najvećih N brojeva, a stranka/koalicija je osvojila onoliko mesta u parlamentu koliko u njenoj koloni ima zaokruženih brojeva. Pri tome, ako su neka dva količnika jednaka, prednost dajemo onom količniku koji je u koloni sa manjim rednim brojem (drugim rečima, prednost dajemo onoj stranki/koaliciji koja se ranije javlja na izbornom spisku).

Ulazna datoteka `ZAD.TXT` ima tačno dva reda:

$$\begin{array}{cccc} N & K & & \\ g_1 & g_2 & \ldots & g_K \end{array}$$

1:	(120.00)	(100.00)	(40.00)	(20.00)
2:	(60.00)	(50.00)	(20.00)	10.00
3:	(40.00)	(33.33)	(13.33)	6.67
4:	(30.00)	(25.00)	10.00	5.00
5:	(24.00)	(20.00)	8.00	4.00
6:	(20.00)	(16.67)	6.67	3.33
7:	(17.14)	(14.29)	5.71	2.86
8:	(15.00)	12.50	5.00	2.50
9:	(13.33)	11.11	4.44	2.22
10:	12.00	10.00	4.00	2.00
11:	10.91	9.09	3.64	1.82
12:	10.00	8.33	3.33	1.67
13:	9.23	7.69	3.08	1.54
14:	8.57	7.14	2.86	1.43
15:	8.00	6.67	2.67	1.33
16:	7.50	6.25	2.50	1.25
17:	7.06	5.88	2.35	1.18
18:	6.67	5.56	2.22	1.11
19:	6.32	5.26	2.11	1.05
20:	6.00	5.00	2.00	1.00
Broj mesta:	9	7	3	1

Tabela 4.1: Dontov proporcionalni sistem za skupštinu sa 20 mesta i četiri stranke/koalicije

U svakom redu brojevi su razdvojeni tačno jednom prazninom i pri tome je $1 \leqslant N \leqslant 500$, $1 \leqslant K \leqslant 25$ i $1 \leqslant g_j \leqslant 6\,000\,000$ za sve $j \in \{1, \ldots, K\}$.

Izlazna datoteka `RES.TXT` ima tačno jedan red u kome se nalazi K brojeva

$$s_1 \quad s_2 \quad \ldots \quad s_K$$

gde su svaka dva susedna razdvojena tačno jednom prozninom. Broj s_j predstavlja broj poslaničkih mesta u parlamentu koja su pripala j-toj stranki/koaliciji.

Na primer, ako skupština ima 20 mesta, a na izbore je izašlo 4 stranke/koalicije koje su osvojile, redom, 120, 100, 40 i 20 glasova, odgovarajuća situacija je prikazana u Tabeli 4.1.

4.2 Matrice kao strukture podataka

Da bi se matrica mogla pojaviti kao argument procedure ili funkcije, potrebno je prvo deklarisati odgovarajući imenovani tip, kako smo već navikli. *Funkcija ne može da vrati matricu kao rezultat*. I to smo očekivali.

Primer. Dve matrice $A = [a_{ij}]$ i $B = [b_{ij}]$ istog formata $m \times n$ se mogu sabirati tako što se saberu odgovarajući elementi, i rezultat je ponovo matrica formata $m \times n$. Na primer:

$$\begin{bmatrix} 2 & 3 & -1 \\ 0 & 1 & 6 \end{bmatrix} + \begin{bmatrix} -2 & 0 & 2 \\ 1 & 9 & 4 \end{bmatrix} = \begin{bmatrix} 0 & 3 & 1 \\ 1 & 10 & 10 \end{bmatrix}.$$

Napisati proceduru

```
procedure MAdd(m, n: integer; a, b: Matrix; var c: Matrix);
```

koja sabira matrice `a` i `b` formata $m \times n$ i rezultat vraća kroz argument `c`.

```
const
  MaxN = 50;
type
  Matrix = array [1 .. MaxN, 1 .. MaxN] of real;

procedure MAdd(m, n : integer; a, b : Matrix; var c : Matrix);
var
  i, j : integer;
begin
  for i := 1 to m do
    for j := 1 to n do
      c[i, j] := a[i, j] + b[i, j]
end;
```

Kao što vidimo iz ovog primera, da bismo u proceduru preneli podatke o nekoj matrici, moramo preneti tri podatka: njene dimenzije `m` i `n`, kao i "tabelu" sa elementima matrice. U prethodnom primeru smo bili srećni zato što su sve tri matrice imale isti format $m \times n$. No, može se desiti da sve matrice sa kojima radimo imaju različite formate. U tom slučaju bi procedura morala da izgleda otprilike ovako:

```
procedure P(m, n: integer; a: Matrix; p, q: integer; b: Matrix;
            var s, t: integer; var c: Matrix)
```

što je užasno nepregledno. Pošto već moramo da tretiramo matrice kao imenovane tipove podataka, zašto ne bismo u isti paket podataka zapakovali i informacije o formatu matrice? Tako dobijamo strukturu podataka koja sve svoje sa sobom nosi, što nam daje veoma prirodan i kompaktan zapis!

Primer. Matrica A formata $m \times p$ i matrica B formata $q \times n$ su *kompatibilne* ako je $p = q$.

Proizvod matrice A formata $m \times p$ i matrice B formata $p \times n$ (koje moraju biti kompatibilne!) je matrica C formata $m \times n$:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1p} \\ a_{21} & a_{22} & \cdots & a_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mp} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{p1} & b_{p2} & \cdots & b_{pn} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{bmatrix}$$

čiji elementi se računaju po formuli

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}.$$

Napisati Pascal proceduru koja računa proizvod dve matrice za koje se pretpostavlja da su kompatibilne.

```
const
  MaxN = 50;
type
  Mat = record
    m, n : integer;
    el : array [1 .. MaxN, 1 .. MaxN] of real
  end;

procedure MatMul(A, B : Mat; var C : Mat);
{pretpostavlja se da su A i B kompatibilne}
{matrice, tj. da je A.n = B.m}
var
  i, j, k : integer;
  sum : real;
begin
  C.m := A.m;
  C.n := B.n;
  for i := 1 to C.m do
    for j := 1 to C.n do begin
      sum := 0;
      for k := 1 to A.n do
        sum := sum + A.el[i, k] * B.el[k, j];
      C.el[i, j] := sum
    end
end;
```

Zadaci.

4.19. Napisati procedure

```
procedure MatRead(var A : Mat);
procedure MatWrite(A : Mat);
```

koje učitavaju, odnosno, ispisuju matricu A.

4.20. Napisati funkciju

```
function SumAll(A : Mat) : real;
```

koja za datu matricu A računa sumu svih elementa matrice.

4.21. Neka je A matrica formata $m \times n$. Matrica B formata $n \times m$ se zove *transponovana matrica matrice A*, i označava sa A^T, ako je $a_{ij} = b_{ji}$ za sve i i j. Napisati proceduru

```
procedure MatTranspose(A : Mat; var B : Mat);
```

koja transponuje matricu A i rezultat vraća kroz svoj argument B.

4.22. Kvadratna matrica H reda n (tj. formata $n \times n$) se zove *Adamarova matrica* ako su njeni elementi samo 1 ili -1 i ako je

$$H \cdot H^T = \begin{bmatrix} n & 0 & 0 & \dots & 0 \\ 0 & n & 0 & \dots & 0 \\ 0 & 0 & n & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & n \end{bmatrix}.$$

Napisati Pascal program koji od korisnika učitava kvadratnu matricu i proverava da li je to Adamarova matrica.

4.23. Napisati proceduru

```
procedure Sub(A : Mat; k, l : integer; var B : Mat);
```

koja od date matrice $A = [a_{ij}]_{m \times n}$ formira matricu $B = [b_{ij}]_{(m-1)\times(n-1)}$ izbacivanjem elemenata k-vrste i l-te kolone matrice A.

4.24. Napisati Pascal funkciju

```
function SubMat(A, S : Mat) : integer;
```

koja utvrđuje koliko se puta matrica S javlja kao podmatrica matrice A.

4.25. Napisati Pascal program koji od korisnika učitava kvadratnu matricu A, pozitivan ceo broj k i potom računa i štampa vrednost matrice B, gde je

(a) $B = E + A + A^2 + A^3 + \ldots + A^k$

(b) $B = E + \frac{1}{1!}A + \frac{1}{2!}A^2 + \frac{1}{3!}A^3 + \ldots + \frac{1}{k!}A^k$

(Napomena: sa E je označena *jedinična matrica*, tj. matrica koja na glavnoj dijagonali ima jedinice, a svi ostali elementi su joj nule; pored toga, $A^2 = A \cdot A$, $A^3 = A \cdot A \cdot A$, itd.)

4.26. Ispitati jednakost dve matrice. (Matrice $A = [a_{ij}]_{m \times n}$ i $B = [b_{ij}]_{p \times q}$ su jednake ako su istih dimenzija i ako je $a_{ij} = b_{ij}$ za sve vrednosti i i j.)

4.27. Ispitati da li je data kvadratna matrica simetrična. (Kvadratna matrica $A = [a_{i,j}]_{n \times n}$ je simetrična ako je $a_{ij} = a_{ji}$ za sve vrednosti i i j.)

4.28. Datoteka `kvmat.dat` tipa `file of KvMat` sadrži kvadratne matrice, pri čemu je tip `KvMat` dat ispod (polje `N` predstavlja red kvadratne matrice, a polje `A` sadrži elemente matrice). Znamo da u datoteci ima najviše 1000 matrica. Napisati Pascal program koji učitava kvadratne matrice iz ove datoteke u niz, sortira niz po tragu matrice i potom sortirani niz upisuje u datoteku `sorkvmat.dat` tipa `file of KvMat`.

```
const
  MaxN = 50;
type
  KvMat = record
    N : integer;
    A : array [1..MaxN, 1..MaxN] of real
  end;
```

†**4.29.** Kvadratna matrica je simetrična ako i samo ako je jednaka svojoj transponovanoj matrici. Dokazati.

†**4.30.** Napisati program koji za dato n generiše magični kvadrat reda n.

†**4.31.** Napisati program koji rešava sistem linearnih jednačina koristeći Gaussov metod eliminacije.

4.3 Struktura statičkih tipova podataka u Pascalu

Programski jezik Pascal ima veoma razgranatu strukturu tipova podataka koja se ukratko može podeliti na

- proste tipove, koji mogu biti:
  - nabrojivi, koji se dalje dele na
    - standardne (`integer`, `longint`, `real`, `boolean`, `char`),
    - one koje je definisao korisnik, i
    - poddomenske (ili intervalne),
  - skup (`set`), i
  - string;
- statičke strukture, koji se realizuju sledećim standardnim metodama struktuiranja:
  - niz (`array`),
  - slog (`record`), i
  - promenljivi slog[1] (`record-case`);
- datoteke (`file`, `text`); i
- dinamičke strukture.

Dinamičke strukture podataka ćemo učiti kasnije. Evo sada rezimea statičkih struktura podataka:

⟨*Skup*⟩ ≡ `set of` ⟨*Nabrojivi tip*⟩
⟨*Niz*⟩ ≡ `array [`⟨*Nabrojivi tip*$_1$⟩`, ...,` ⟨*Nabrojivi tip*$_k$⟩`] of` ⟨*Proizvoljan tip*⟩
⟨*Slog*⟩ ≡ `record`
⟨*ime*$_{11}$⟩`, ...,` ⟨*ime*$_{1n}$⟩ `:` ⟨*Proizvoljan tip*⟩`;`
⋮
⟨*ime*$_{s1}$⟩`, ...,` ⟨*ime*$_{sm}$⟩ `:` ⟨*Proizvoljan tip*⟩
`end`
⟨*Datoteka*⟩ ≡ `file of` ⟨*Proizvoljan tip*⟩
| `text`

[1] promenljive slogove nismo obradili u ovom kursu

Podsetimo se da strukture mogu biti prenete u potprogram po referenci (`var` argumenti) ili po vrednosti ("obični" argumenti, bez var). U prvom slučaju se promene na elementima strukture odslikavaju na strukturi koja je navedena u pozivu potprograma, a u drugom slučaju ne zato što će Pascal prevodilac napraviti kopiju i nju preneti u potprogram.

Izuzetak od pravila čine datoteke koje se u potporgram *uvek* prenose kao `var` argumenti. Razlog za ovo ograničenje je jednostavan: datoteke se nalaze u spoljašnjoj memoriji i prevodilac nema načina da obezbedi da se u potprogram unese kopija strukture. Uz to, *statička struktura ne može biti rezultat funkcije.*

Glava 5

Pokazivači

Promenljive sa kojima smo do sada radili su imale osobinu da o njima brine računar: u deklaraciji promenljive se navede njen tip, a program sam kreira kućicu u odgovarajućem delu memorije kada se aktivira potprogram koji je deklarisao promenljivu, i sam oslobodi prostor u tom delu memorije kada se potprogram koji je deklarisao promenljivu završi. Takve promenljive se zovu *statičke* promenljive.

Programski jezik Pascal poznaje i drugu vrstu promenljivih, *dinamičke promenljive*, kod kojih za rezervisanje prostora i oslobađanje prostora kada nam promenljiva više ne treba mora da brine programer. Neko se u ovom trenutku može upitati, šta će nam koncept promenljive kod koga moramo ručno da rezervišemo i oslobađamo prostor, kada već postoji jednostavan automatizovani sistem koji sam brine o svemu? Videćemo da je koncept dinamičkih promenljivih jedan *izuzetno* koristan koncept (inače mu ne bismo posvetili čitavu jednu glavu)!

5.1 Dinamičke promenljive

Da bismo shvatili dinamičke promenljive, treba prvo da objasnimo koncept *pokazivača*. Pokazivač (engl. *pointer*) je "strelica" koja pokazuje na neki deo memorije i deklariše se ovako:

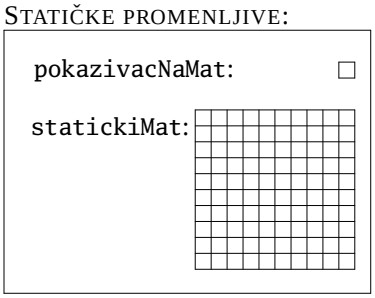
```
var
    p : ↑⟨proizvoljan tip⟩;
```

Na primer,

```
const
  MaxN = 10;
type
  Osoba = record
    ime, prezime : string;
    JMBG : array[1 .. 13] of integer
  end;
  Mat = array [1 .. MaxN, 1 .. MaxN] of real;
var
  pokazivacNaOsobu : ↑Osoba;
  pokazivacNaMat : ↑Mat;
  pokazivacNaReal : ↑real;
```

Razlika između promenljive tipa `Mat` i promenljive tipa `↑Mat` se najbolje vidi na sledećoj slici:

```
const
  MaxN = 10;
type
  Mat = array [1 .. MaxN,
               1 .. MaxN] of real;
var
  pokazivacNaMat : ↑Mat;
  statickiMat : Mat;
```



Eto sad! Gde je druga matrica?

Pokazivačka promenljiva ne postoji dok korisnik *ručno* ne rezerviše prostor za njen sadržaj. Kada naiđe na pokazivačku promenljivu, računar automatski rezerviše samo prostor za *strelicu* koja će jednog dana pokazivati na matricu. Matricu moramo ručno da napravimo. Naredba koja rezerviše prostor za dinamičke promenljive zove se `new` i koristi se ovako:

`new(`⟨*neka pokazivačka promenljiva*⟩`);`

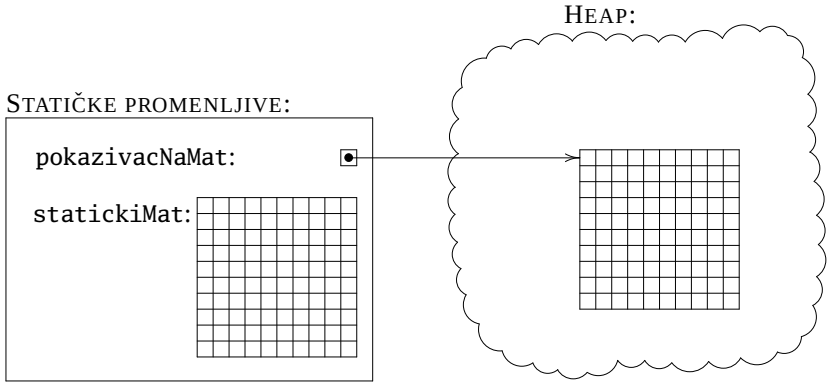
na primer,

```
new(pokazivacNaMat);
```

Naredba `new` u posebnom delu memorije koji se zove *heap* (engl. gomila, hrpa) rezerviše odgovarajući prostor, pa u pokazivačku promenljivu stavi samo strelicu (pokazivač) na taj prostor. Tako, nakon naredbe

```
new(pokazivacNaMat);
```

situacija u memoriji izgleda ovako:



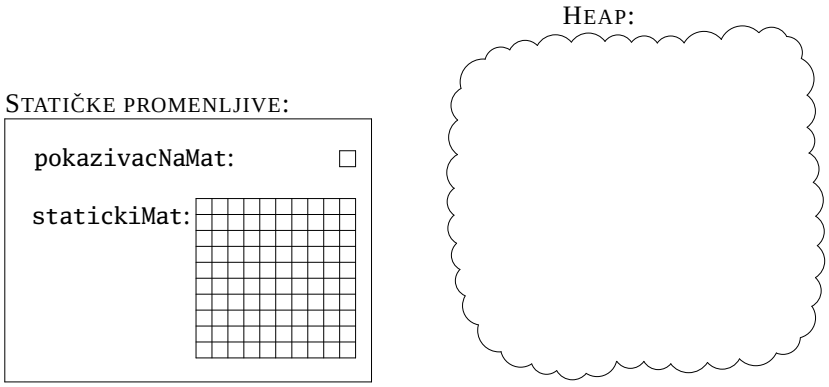
Naredba koja oslobađa prostor rezervisan za pokazivačku promenljivu zove se `dispose` i koristi se ovako:

`dispose(`⟨*neka pokazivačka promenljiva*⟩`);`

na primer,

```
dispose(pokazivacNaMat);
```

Ona uništi sadržaj na heapu koji je bio rezervisan za pokazivačku promenljivu. Situacija nakon `dispose(pokazivacNaMat)` izgleda ovako:

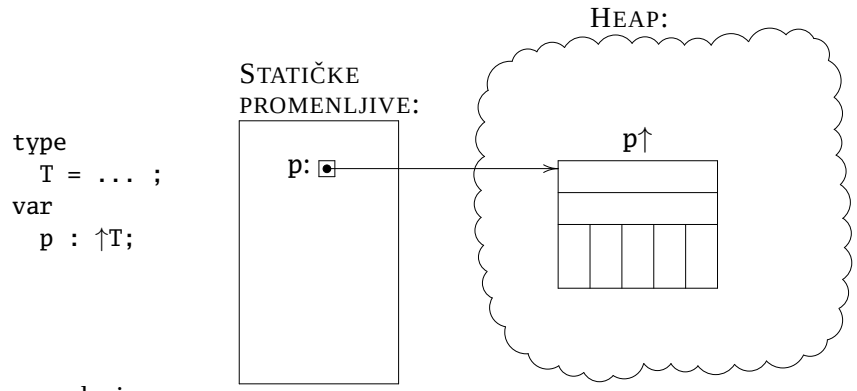


5.2 Pristupanje dinamičkim promenljivim

Kada rezervišemo prostor za dinamičku promenljivu, naravno da bismo želeli da ga koristimo kao svaku drugu promenljivu. Postoji jednostavno pravilo koje

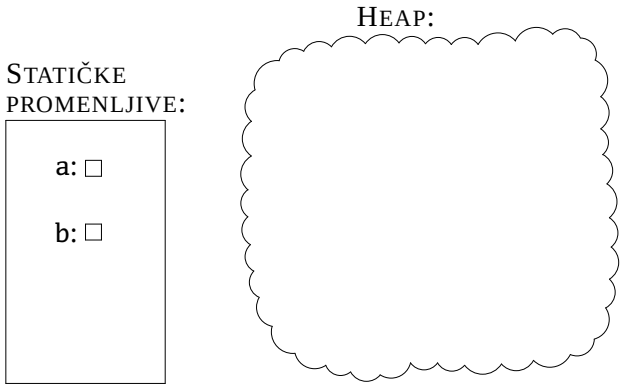
reguliše pristupanje pokazivačkim promenljivim. Ako je T neki tip i p promenljiva koja pokazuje na taj tip, tada

p	je "obična" promenljiva	tipa ↑T	(strelica)
p↑	je promenljiva na heapu	tipa T	(ono na šta p pokazuje)



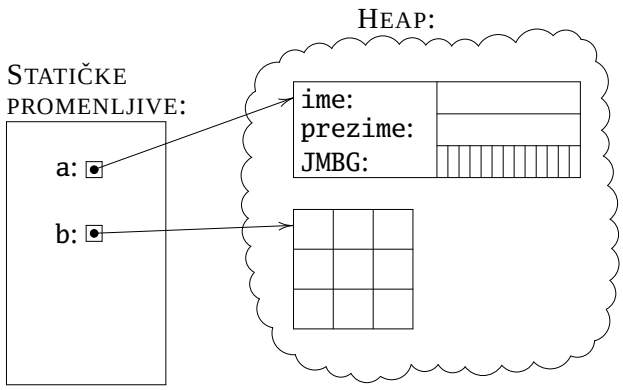
Na primer, neka je

```
const
  MaxN = 3;
type
  Osoba = record
    ime, prezime : string;
    JMBG : array [1 .. 13] of integer
  end;
  Mat = array [1 .. MaxN, 1 .. MaxN] of real;
var
  a : ↑Osoba;
  b : ↑Mat;
```



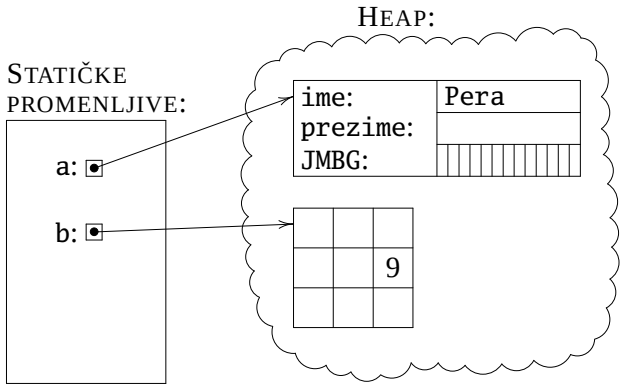
Tada je a strelica koja pokazuje na dinamičku promenljivu tipa Osoba, a b je strelica koja pokazuje na dinamičku promenljivu tipa Mat i računar će automatski rezervisati samo prostor za dve strelice. Dinamičke promenljive a↑ tipa Osoba i b↑ tipa Mat *još uvek ne postoje i moraju se kreirati ručno:*

```
begin
  new(a); new(b);
```



Da bismo pristupili ovim promenljivim, samo treba da se setimo da je a↑ običan slog, a b↑ obična matrica:

```
  a↑.ime := 'Pera';
  b↑[2,3] := 9;
  ...
end.
```



5.3 Pridruživanje i jednakost pokazivača

Neka je T proizvoljan tip i neka su p i q promenljive koje pokazuju na taj tip:

```
type
  T = ... ;
  PtrT = ↑T;
var
  p, q : PtrT;
```

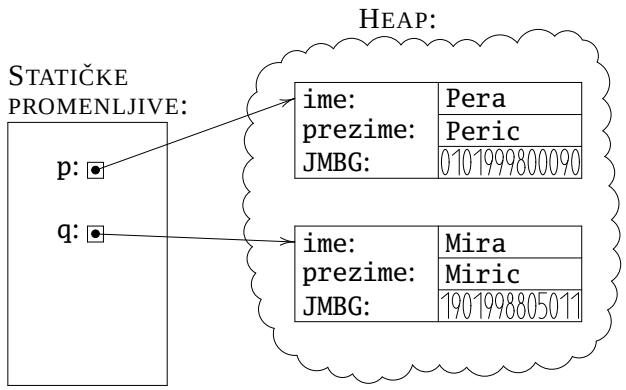
Činjenica da su p i q strelice, a p↑ i q↑ promenljive tipa T ima važne posledice:

`q := p`	znači: *strelica* q *postaje strelica* p, pa nakon dodele strelica q pokazuje tamo gde pokazuje strelica p
`q↑ := p↑`	znači: *promenljiva* q↑ *postaje promenljiva* p↑, pa nakon dodele strelice q i p pokazuju i dalje na različite promenljive koje sada imaju isti sadržaj
`q = p`	proverava *da li strelice* p i q *pokazuju na istu promenljivu*

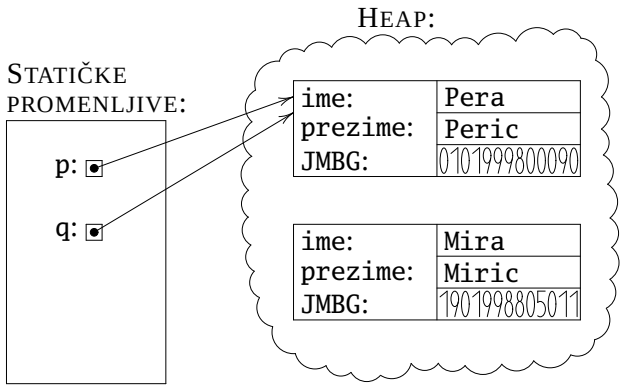
Na primer, neka su p i q pokazivači na tip Osoba:

```
type
  Osoba = record
    ime, prezime : string;
    JMBG : array[1 .. 13] of integer
  end;
  PtrOsoba = ↑Osoba;
var
  p, q : PtrOsoba;
```

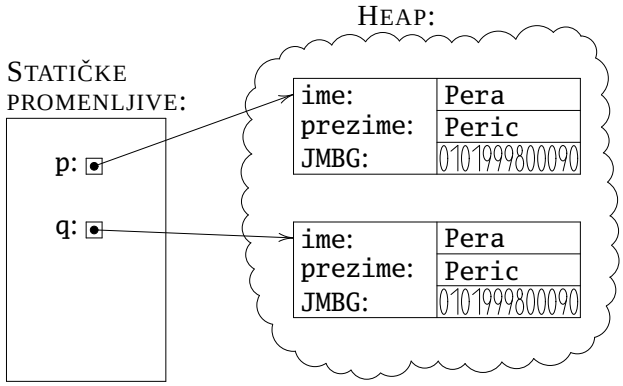
i neka su na heapu kreirane odgovarajuće dinamičke promenljive, kao na slici ispod. U ovom trenutku je `q <> p` zato što p i q pokazuju na različite kućice na heapu.



Kada kažemo `q := p;` to znači da strelica `q` dobija vrednost strelice `p`, pa nakon ove naredbe *q pokazuje na ono na šta pokazuje p.* U ovom trenutku je `q = p` zato što `p` i `q` pokazuju na istu kućicu na heapu. *Pažnja: ako niko drugi ne pokazuje na Miru Mirić, tim podacima više nikako ne možemo pristupiti!*



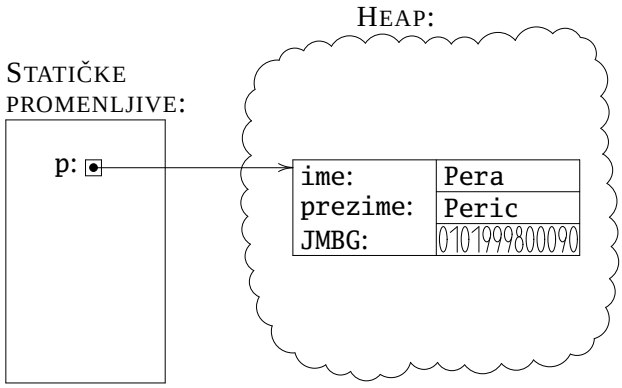
S druge strane, kada kažemo `q↑:= p↑;` tada se sadržaj promenljive `p↑` (koja je tipa `T`) kopira u sadržaj promenljive `q↑`. Iako su sadržaji obe dinamičke promenljive isti, u ovom trenutku važi `q <> p` zato što `q` i `p` pokazuju na *različite kućice* na heapu.



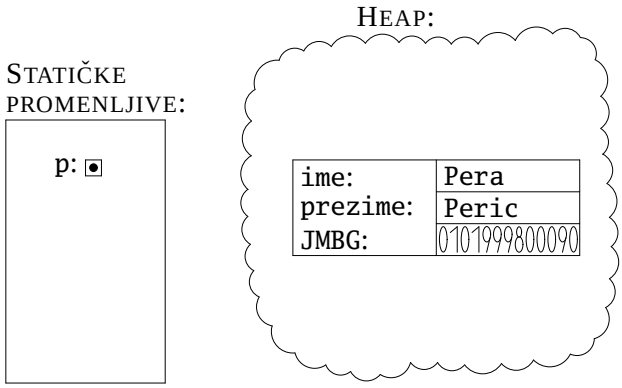
Postoji posebna vrednost `nil` koja se može pridružiti svakom pokazivaču. Ako je `p` pokazivačka promenljiva kao u primeru pored tada naredba `p := nil;` znači da se promenljivoj `p` dodeljuje "prazna strelica" koja nigde ne pokazuje.

```
type
  T = ... ;
  PtrT = ↑T;
var
  p : PtrT;
```

Na primer, ako je u jednom trenutku situacija bila kao na sledećoj slici:



onda posle `p := nil;` promenljiva `p` ne pokazuje ni na šta. I u ovom slučaju, ako niko drugi ne pokazuje na Peru Perić, tim podacima više nikako ne možemo pristupiti!

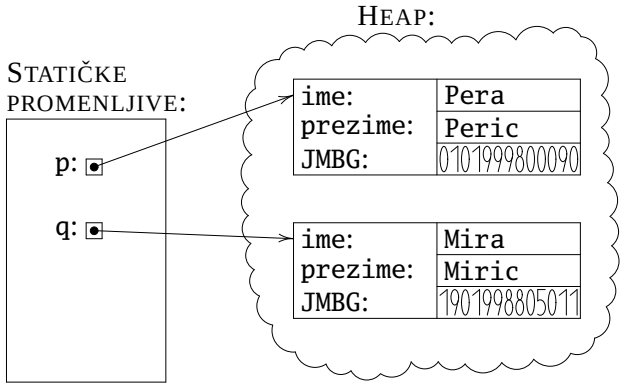


Do sada smo videli već dve prilike kada može da se desi da posle nekih naredbi na heapu može da ostane đubre, što će reći, podaci koji zauzimaju prostor, a do kojih nikako ne možemo da dobacimo. Kada znamo da niko drugi ne pokazuje na neki podatak na heapu, potrebno je počistiti đubre za sobom i osloboditi prostor za neke druge dinamičke promenljive. Tome služi naredba `dispose`. Na primer, neka je:

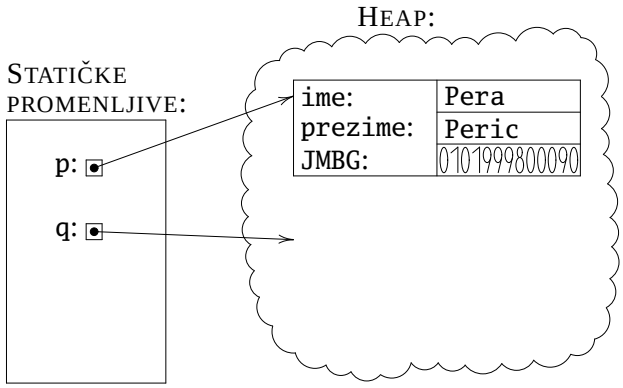
```
type
  Osoba = record
    ime, prezime : string;
    JMBG : array[1 .. 13] of integer
  end;
  PtrOsoba = ↑Osoba;
```

```
var
  p, q : PtrOsoba;
```

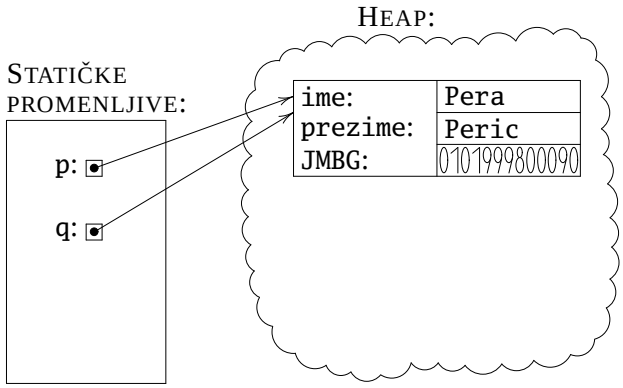
i neka su na heapu kreirane odgovarajuće dinamičke promenljive:



Tada se prvo naredbom `dispose(q);` počisti đubre:



pa naredba `q := p;` prebacuje pokazivač tamo gde mi to želimo, s tim da je sada sve čisto:



5.4 Pokazivači kao argumenti procedura i funkcija

Prva radosna vest u ovom odeljku je da funkcija može da vrati pokazivač kao svoj rezultat! Zato, ako želimo da vratimo strukturu kao rezultat nekog računanja, imamo dve mogućnosti: možemo je vratiti kao var argument procedure, ili kao pokazivač na strukturu koji je rezultat funkcije:

```
type
  T = record
    ...
  end;

procedure P(c: char; var rez: T);
begin
  ...
end;
```

```
type
  T = record
    ...
  end;
  PtrT = ↑T;

function F(c: char): PtrT;
begin
  ...
end;
```

Evo funkcije koja računa transponovanu matricu i vraća je kao svoj rezultat:

```
const
  MaxN = 20;
type
  Mat = record
    M, N : integer;
    el : array [1 .. MaxN, 1 .. MaxN] of real
  end;
  PtrMat = ↑Mat;

function Transpose(a : Mat) : PtrMat;
var
  b : PtrMat;
  i, j : integer;
begin
  {prvo kreiramo dinamicku promenljivu na heapu}
  new(b);
  {koju potom popunjavamo odgovarajucim vrednostima}
  b↑.M := a.N;
  b↑.N := a.M;
  for i := 1 to a.M do
    for j := 1 to a.N do
      b↑.el[j, i] := a.el[i, j];
  {i na kraju kao rezultat vratimo pokazivac na tu promenljivu}
  Transpose := b
end;
```

Sa ovim pristupom moramo biti pažljivi, zato što je potrebno ručno rezervisati prostor na heapu, i *ručno ga oslobađati*. Ako pri tome nismo dovoljno pažljivi i ako dopustimo da se na heapu nagomila đubre, program može da pukne iz čistog mira, iako na prvi pogled sve deluje korektno!

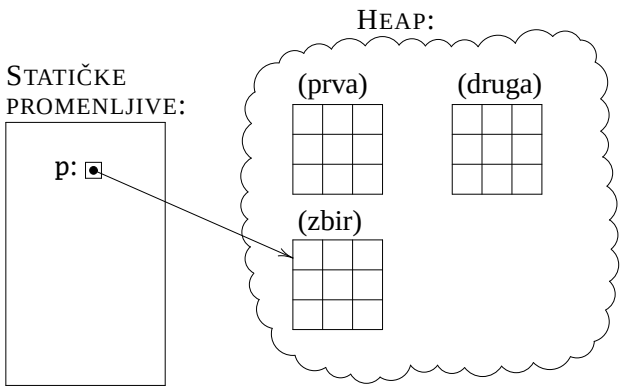
Na primer, neka je `ReadPtrMat` funkcija koja od korisnika učita matricu i vrati pokazivač na odgovarajući objekt na heapu, a neka je `AddPtrMat` funkcija koja sabira dve matrice na heapu i vraća zbir kao matricu na heapu:

```
function ReadPtrMat() : PtrMat;
var
  b : PtrMat;
  i, j : integer;
begin
  new(b);
  readln(b↑.M, b↑.N);
  for i := 1 to b↑.M do
    for j := 1 to b↑.N do
      readln(b↑.el[i, j]);
  ReadPtrMat := b
end;

function AddPtrMat(a, b : PtrMat) : PtrMat;
{pretpostavlja se da su a i b kompatibilne}
var
  c : PtrMat;
  i, j : integer;
begin
  new(c);
  c↑.M := a↑.M;
  c↑.N := a↑.N;
  for i := 1 to a↑.M do
    for j := 1 to a↑.N do
      c↑.el[i, j] := a↑.el[i, j] + b↑.el[i, j];
  AddPtrMat := c
end;
```

Tada naredba

```
p := AddPtrMat(ReadPtrMat(), ReadPtrMat());
```

učita dve matrice, kreira odgovarajuće objekte na heapu, sabere ih i zbir vrati kao rezultat svog rada, a na heapu ostaju dve početne matrice u svojstvu đubreta, zato što na njih više niko ne pokazuje:



Pokazivač se, naravno, može pojaviti i kao argument procedure ili funkcije i to ćemo intenzivno koristiti u sledećem odeljku. Na primer, procedura koja transponuje matricu može da se napiše i ovako:

```
procedure Transpose2(a : PtrMat);
var
  P, i, j : integer;
  pomi : integer;
  pomr : real;
begin
  {prvo u P postavimo maksimum brojeva a↑.M i a↑.N}
  if a↑.M > a↑.N then P := a↑.M else P := a↑.N;
  {sada razmenimo a↑.M i a↑.N}
  pomi := a↑.M; a↑.M := a↑.N; a↑.N := pomi;
  {i na kraju razmenimo a↑.el[i,j] sa a↑.el[j,i] za sve i, j}
  for i := 1 to P - 1 do
    for j := i + 1 to P do begin
      pomr := a↑.el[i, j];
      a↑.el[i, j] := a↑.el[j, i];
      a↑.el[j, i] := pomr
    end
end;
```

Ovaj primer ima jedan važan aspekt: proceduri je prosleđen pokazivač na matricu, ali kao običan argument, ne kao `var` argument, a promene koje smo učini na matrici su ostale na snazi i nakon rada procedure! To je zato što se tokom aktivacije procedure ili funkcije ne prave kopije promenljivih koje su na heapu.

☞ *Kada se u proceduru/funkciju unese pokazivač na neku promenljivu, čak i kada argument nije deklarisan kao `var` argument, promene koje smo izvršili na heapu ostaju na snazi i posle završetka rada procedure/funkcije!*

Zadaci.

5.1. Neka je dat sledeći tip:

```
type
  Osoba = record
    ime, prezime : string;
    JMBG : array[1 .. 13] of integer
  end;
  PtrOsoba = ↑Osoba;
```

Napisati funkciju `function UcitajOsobu() : PtrOsoba;` koja od korisnika učitava podatke o nekoj osobi i vraća pokazivač na odgovarajući slog.

5.2. Kompleksni broj može da se opiše ovako:

```
type
  StaticComplex = record
    re, im : real
  end;
  Complex = ↑StaticComplex;
```

Napisati sledeće procedure i funkcije koje manipulišu kompleksnim brojevima:

- `function RdComplex(): Complex;` koja učitava kompleksan broj;
- `procedure WrComplex(z: Complex);` koja ispisuje kompleksan broj;
- `function NewComplex(re, im: real): Complex;` koja konstruiše kompleksan broj čiji realni deo je `re`, a imaginarni deo je `im`;
- `function AddComplex(u, v: Complex): Complex;` koja sabira kompleksne brojeve;
- `function SubComplex(u, v: Complex): Complex;` koja oduzima kompleksne brojeve;
- `function MulComplex(u, v: Complex): Complex;` koja množi kompleksne brojeve;
- `function DivComplex(u, v: Complex): Complex;` koja deli kompleksne brojeve; pretpostavlja se da `v` nije nula.

5.3. (a) Napisati Pascal program koji od korisnika učitava prirodan broj n, potom kompleksne brojeve $z_1, \ldots, z_n$ i računa i štampa kompleksni broj

$$z_1 + \ldots + z_n.$$

(b) Napisati Pascal program koji od korisnika učitava prirodan broj n, potom kompleksne brojeve $z_1, \ldots, z_n$ i računa i štampa kompleksni broj

$$\frac{1}{z_1} + \ldots + \frac{1}{z_n}.$$

(c) Napisati Pascal program koji od korisnika učitava prirodan broj n i kompleksni broj z i računa i štampa kompleksni broj

$$1 + z + \frac{z^2}{2} + \frac{z^3}{3} + \ldots + \frac{z^n}{n}.$$

(d) Napisati Pascal program koji od korisnika učitava pozitivan ceo broj n, potom kompleksne brojeve $a_0, a_1, \ldots, a_n, z$ i računa i štampa kompleksni broj

$$a_0 + a_1 z + a_2 z^2 + \ldots + a_n z^n.$$

5.4. Neka je matrica opisana ovako:

```
const
  MaxN = 10;
type
  Mat = record
    M, N : integer;
    el : array [1 .. MaxN, 1 .. MaxN] of real;
  end;
  PtrMat = ↑Mat;
```

Napisati sledeće procedure i funkcije:

- `function RdMat() : PtrMat;` koja učitava matricu;
- `procedure WrMat(m : PtrMat);` koja ispisuje matricu;
- `function AddMat(a, b : PtrMat) : PtrMat;` koja sabira matrice istog formata; ako matrice nisu istog formata vraća `nil`;
- `function MulMat(a, b : PtrMat) : PtrMat;` koja množi kompatibilne matrice; ako matrice nisu komaptibilne vraća `nil`.

5.5. Neka je dat sledeći tip:

```
const
  MaxN = 1000;
type
  Osoba = record
    ime, prezime : string;
    JMBG : array[1 .. 13] of integer
  end;
  NizOsoba = array [1 .. MaxN] of ↑Osoba;
```

Napisati proceduru

```
procedure SortNiz(var a: NizOsoba; N: integer);
```

koja sortira segment `[1 .. N]` niza a po prezimenu osobe premeštanjem pokazivača u nizu.

5.6. Neka je dat sledeći tip:

```
const
  MaxN = 1000;
  MaxDim = 50
type
  Vector = array [1 .. MaxDim] of real;
  NizVec = array [1 .. MaxN] of ↑Vector;
```

Napisati proceduru

```
procedure SortNiz(var a : NizVec; N : integer);
```

koja sortira segment `[1 .. N]` niza a po prvom elementu premeštanjem pokazivača u nizu.

5.7. Blok-matrica je matrica čiji elementi su matrice. Iako u opštem slučaju to nije tako, mi ćemo pretpostavljati da su sve matrice koje se javljaju kao elementi blok-matrice istog formata. Na primer,

$$\begin{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & \begin{bmatrix} -3 & 1 \\ 2 & 1 \end{bmatrix} \\ \begin{bmatrix} 9 & 11 \\ 0 & 11 \end{bmatrix} & \begin{bmatrix} 2 & 2 \\ 2 & 1 \end{bmatrix} & \begin{bmatrix} 3 & 1 \\ 7 & 6 \end{bmatrix} \\ \begin{bmatrix} 1 & 6 \\ 2 & 7 \end{bmatrix} & \begin{bmatrix} 4 & 2 \\ 3 & 5 \end{bmatrix} & \begin{bmatrix} 1 & 1 \\ 2 & 3 \end{bmatrix} \end{bmatrix}$$

je blok matrica 3×3 čiji elementi su matrice formata 2×2. Blok-matrice ćemo opisivati na sledeći način:

```
const
  MaxN = 20;
type
  Mat = array [1 .. MaxN, 1 .. MaxN] of real;
  PtrMat = ↑Mat;
  BlockMat = record
    M, N, K, L : integer;
    el : array [1 .. MaxN, 1 .. MaxN] of PtrMat;
  end;
```

gde su M i N dimenzije blok-matrice, a K i L dimenzije svakog elementa blok-matrice (u prethodnom primeru je $M = N = 3$ i $K = L = 2$). Napisati sledeće procedure i funkcije:

- `procedure RdBlockMat(var m: BlockMat);` koja učitava blok-matricu;
- `procedure WrBlockMat(m :BlockMat);` koja ispisuje blok-matricu;
- `procedure TrBlockMat(a :BlockMat; var b: BlockMat);` koja transponuje blok-matricu (transponovanje blok-matrice se vrši tako što se transponuje ta matrica, ali i svi njeni elementi!)
- `procedure AddBlockMat(a, b: BlockMat; var c: BlockMat);` koja sabira blok-matrice za koje pretpostavljamo da su istog formata;
- `procedure MulBlockMat(a, b: BlockMat; var c: BlockMat);` koja množi kompatibilne blok-matrice za koje pretpostavljamo da su kompatibilne (Pažnja: i blok-matrice i matrice koje se javljaju kao elementi moraju biti kompatibilne; drugim rečima, da bismo bogli da pomnožimo dve blok-matrice mora biti zadovoljeno: `a.N = b.M` i `a.L = b.K`!)

Glava 6

Liste

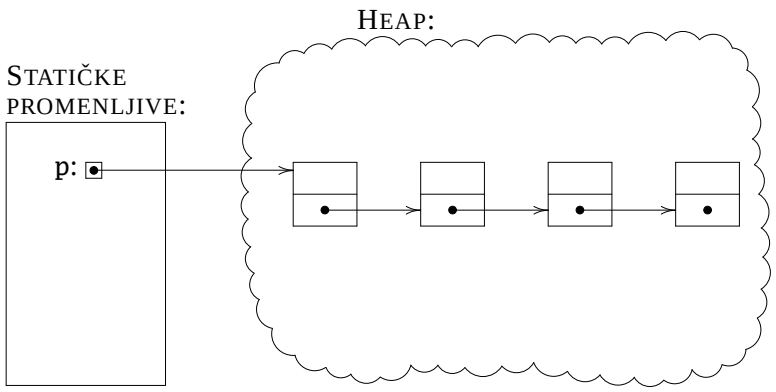
Strukture podatada sa kojima smo se do sada sretali (nizovi, slogovi) se zovu još i *statičke* strukture podataka zato što je takva struktura podataka u potpunosti poznata pre prevođenja programa i ne može se menjati tokom izvršavanja programa. Uvođenjem pokazivača stvorile su se pretpostavke da se upoznamo sa jednom novom vrstom struktura podataka koje zovemo *dinamičke* strukture podataka. Dinamičke strukture podataka se kreiraju i menjaju tokom izvršenja programa.

Liste predstavljaju najjednostavniju dinamičku strukturu podataka. U ovom poglavlju ćemo se upoznati sa jednostruko povezanim listama i operacijama sa ovim listama. Potom ćemo razmatrati neke jednostavnije rekurzivne algoritme sa jednostruko povezanim listama, a glavu zaključujemo pričom o dvostruko povezanim listama.

6.1 Jednostruko povezane liste

Jednostruko povezana lista je niz kućica na heapu koje pokazuju svaka na onu iza sebe kao na Sl. 6.1. Pokazivačka promenljiva pokazuje na prvi element liste koji se zove *glava liste*, a svaki element liste potom pokazuje na onaj iza njega. Poslednji element liste ne pokazuje ni na šta i prepoznajemo ga po tome što njegovo polje `next` ima vrednost `nil`. Na primer, spisak (shvaćen kao niz osoba) se može deklarisati ovako:

```
type
  Spisak = ↑Osoba;
  Osoba = record
    ime, prezime : string;
    JMBG : array [1 .. 13] of integer;
    sledeci : Spisak
  end;
```



Slika 6.1: Jednostruko povezana lista

Polja `ime`, `prezime` i `JMBG` sadrže informacije o osobi, dok polje `sledeci` sadrži pokazivač na sledeću kućicu u nizu, ili `nil` ako se radi o poslednjoj kućici. Slično tome, lista celih brojeva se može deklarisati ovako:

```
type
  IntList = ↑IntListElem;
  IntListElem = record
    n : integer;
    next : IntList
  end;
```

Polje `n` sadrži koristan podatak, dok polje `next` sadrži pokazivač na narednu kućicu u nizu, ili `nil` ukoliko se radi o poslednjoj kućici.

Procedura koja ispisuje sve elemente celobrojne liste izgleda ovako:

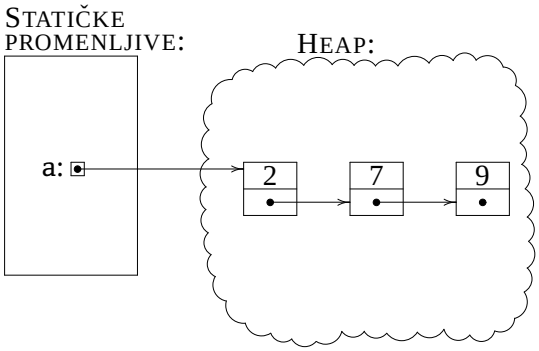
```
procedure WrIntList(p : IntList);
begin
  while p <> nil do begin
    writeln(p↑.n);
    p := p↑.next
  end
end;
```

Sve dok pokazivač `p` pokazuje na neku kućicu u listi radimo sledeće: ispišemo sadržaj polja `n` (to je podatak koji je smešten u kućici), i onda se naredbom

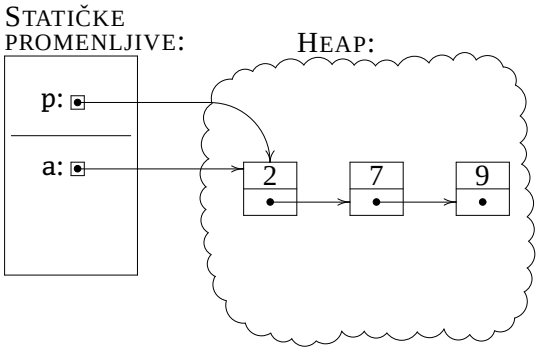
```
p := p↑.next
```

premestimo na sledeću kućicu u nizu. *Ovo je izuzetno važan idiom pri radu sa pokazivačima i treba ga zapamtiti!*

Pogledajmo sada detaljnije kako ova procedura radi. Neka je `a` globalna promenljiva tipa `IntList` i neka smo nekako uspeli da formiramo listu i da je popunimo celim brojevima kao na slici pored.



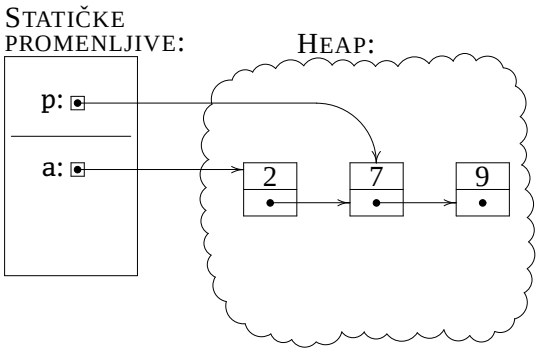
Kada iz glavnog programa pozovemo proceduru `WrIntList(a);` mehanizam aktivacije procedure će u memoriji napraviti novu kućicu za lokalnu promenljivu `p` ove procedure i kopiraće vrednost promenljive `a` u promenljivu `p`. To znači da će strelica `p` pokazivati tamo gde pokazuje strelica `a`.



Sada krećemo sa izvršavanjem `while`-petlje. Pošto je `p <> nil` program ispiše vrednost polja `p↑.n` što je 2, a naredbom

```
p := p↑.next
```

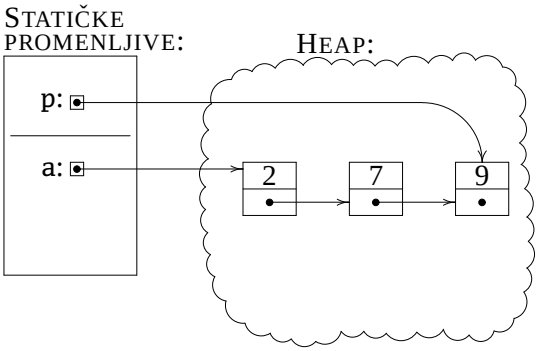
se strelica `p` premesti na narednu kućicu.



Ponovo je `p <> nil`, pa program ispiše vrednost polja `p↑.n` što je ovaj put 7 i naredbom

```
p := p↑.next
```

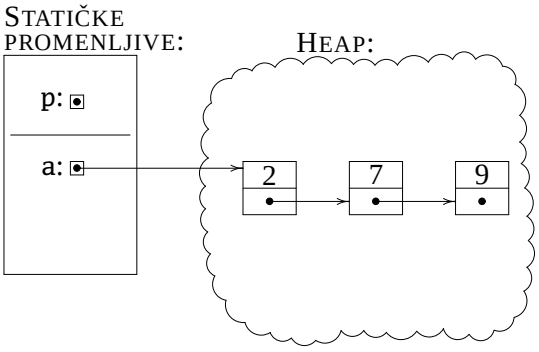
se strelica `p` premesti na narednu kućicu.



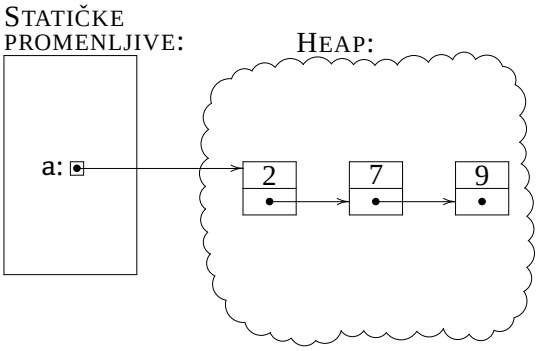
Ponovo je `p <> nil`, pa program ispiše vrednost polja `p↑.n` što je ovaj put 9. Sada je, međutim, `p↑.next = nil` tako da naredba

```
p := p↑.next
```

u kućicu `p` smešta vrednost `nil`.



Sada je `p = nil`, pa `while`-petlja završava sa radom. Time se i procedura završava i program iz memorije uklanja njene lokalne promenljive.



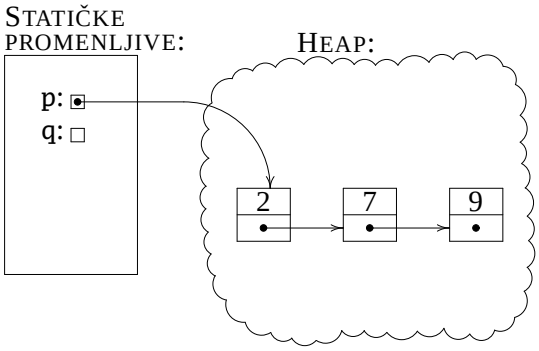
Primer. Napisati funkciju koja određuje dužinu (= broj elemenata) celobrojne liste.

```
function Len(p : IntList) : integer;
var
  n : integer;
begin
  n := 0;
  while p <> nil do begin
    n := n + 1;
    p := p↑.next
  end;
  Len := n
end;
```

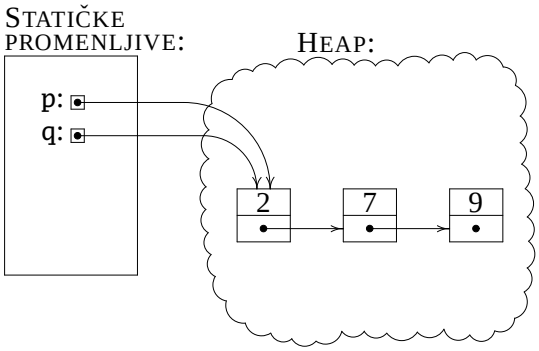
Primer. Napisati proceduru koja uništava celobrojnu listu i oslobađa prostor na heapu koji je bio rezervisan za njene elemente.

```
procedure Kill(var p : IntList);
var
  q : IntList;
begin
  while p <> nil do begin
    q := p;
    p := p↑.next;
    dispose(q)
  end
end;
```

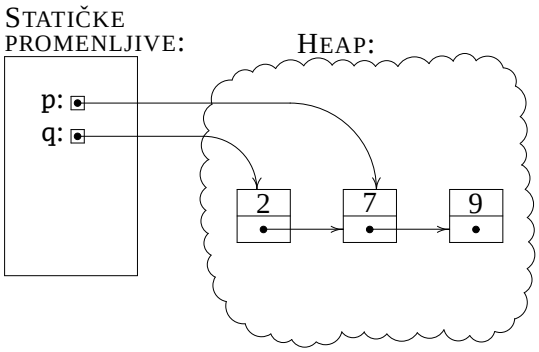
Pogledaćemo detaljno kako radi procedura `Kill`. Pošto je `p <> nil`, izvršava se telo petlje:



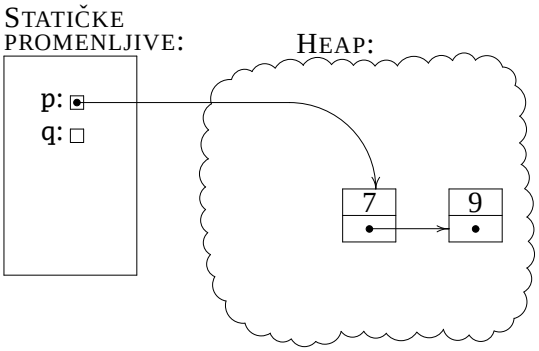
`q := p;`



`p := p↑.next;`

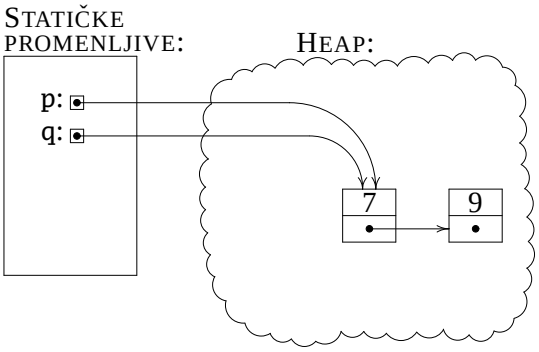


```
dispose(q)
```

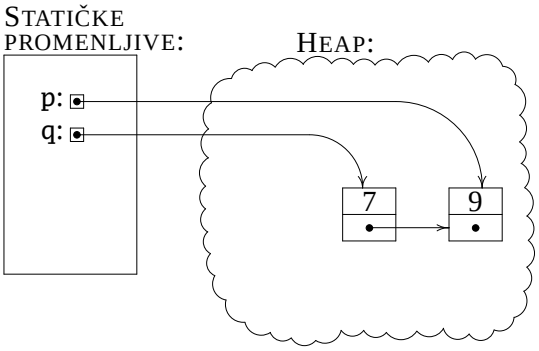


Pošto je `p <> nil` ponovo se izvršava telo petlje:

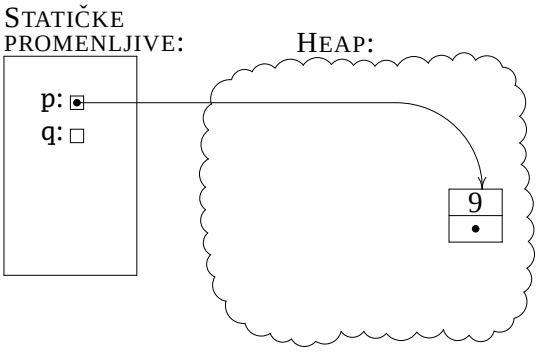
```
q := p;
```




```
p := p↑.next;
```

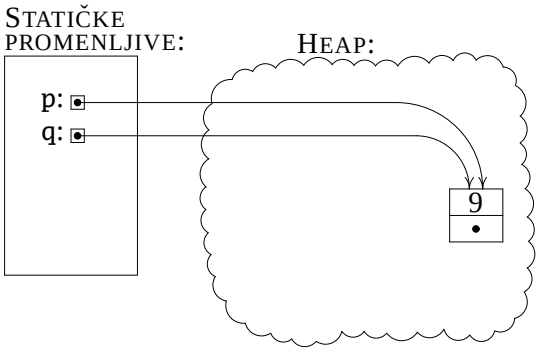



```
dispose(q)
```

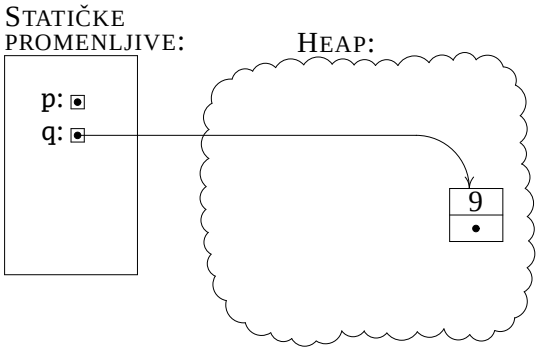


Pošto je `p <> nil` još jednom se izvršava telo petlje:

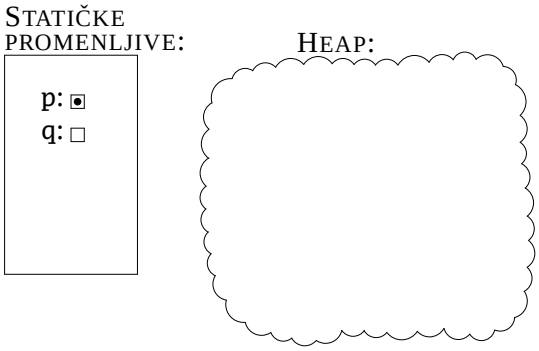
```
q := p;
```




```
p := p↑.next;
```




```
dispose(q)
```



Sada je `p = nil` i petlja se završava, a time i procedura `Kill`.

Zadaci.

6.1. Napisati funkciju

```
function Sum(p : IntList) : integer;
```

koja određuje zbir svih elemenata u listi celih brojeva. Suma prazne liste je 0.

6.2. Napisati funkciju

```
function Member(p : IntList; n : integer) : Boolean;
```

koja proverava da li se dati element nalazi u listi celih brojeva.

6.3. Napisati funkciju

```
function Max(p : IntList) : integer;
```

koja određuje najveći element u listi celih brojeva za koju pretpostavljamo da nije prazna.

6.4. Napisati funkciju

```
function Sum2(p : IntList) : integer;
```

koja određuje zbir svih elemenata koji se u datoj listi celih brojeva nalaze na parnim mestima. Suma prazne liste je 0.

6.5. Napisati funkciju

```
function SumN(p : IntList; k : integer) : integer;
```

koja određuje zbir svih elemenata koji se u datoj listi celih brojeva nalaze na mestima čiji redni broj je deljiv sa k. Suma prazne liste je 0.

6.6. Napisati funkciju

```
function CountMin(p : IntList) : integer;
```

koja određuje koliko se puta najmanji element liste celih brojeva p pojavljuje u toj listi. Za prazne liste ova funkcija vraća 0.

6.2 Osnovne operacije sa listama

U ovom odeljku ćemo videti kako se implementiraju neke od osnovnih operacija nad listama. Sve operacije koje ćemo videti mogu se primeniti na proizvoljne jednostruko povezane liste, a mi ćemo kao i do sada u primerima koristiti liste celih brojeva:

```
type
  IntList = ↑IntListElem;
  IntListElem = record
    n : integer;
    next : IntList
  end;
```

Primer. Napisati proceduru koja ubacuje dati element na početak date liste.

```
procedure Insert(var p : IntList;
                     n : integer);
var
  q : IntList;
begin
  new(q);
  q↑.n := n;
  q↑.next := p;
  p := q
end;
```

Primer. Napisati proceduru koja iz date neprazne liste izbacuje njen prvi element.

```
procedure Drop(var p : IntList);
var
  q : IntList;
begin
  q := p; p := p↑.next;
  dispose(q)
end;
```

Koristeći proceduru `Insert` možemo lako napisati programski fragment koji od korisnika učitava nekoliko celih brojeva i formira listu čiji elementi su učitani brojevi, kako je to pokazano u primeru pored.

```
p := nil;
while not eoln do begin
  read(n);
  Insert(p, n)
end;
```

U listi koja je formirana na ovaj način elementi nisu zadržali redosled kojim su učitani: element koga smo poslednjeg učitali nalazi se na početku liste. Ako želimo da formiramo listu u kojoj se elementi nalaze onim redom kojim smo ih učitali od korisnika, potrebno je malo više mašte.

```
procedure RdList(var p: IntList);
var
  n : integer;
  q, r : IntList;
begin
  p := nil;
  while not eoln do begin
    readln(n);
    new(r); r↑.n := n;
    if p = nil then begin
      p := r; q := r
    end
    else begin
      q↑.next := r;
      q := r
    end
  end;
  q↑.next := nil
end;
```

Primer. Napisati proceduru koja iz date neprazne liste izbacuje njen poslednji element.

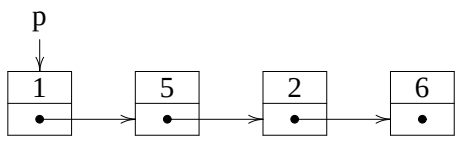
Procedura prvo proveri da li se lista sastoji od samo jednog elementa i ako je to tako, uništi jedini element liste. Ovde je na delu još jedan važan idiom pri radu sa pokazivačima:

☞ *neprazna lista na koju pokazuje* `p` *ima samo jedan element ako i samo ako je* `p↑.next = nil`

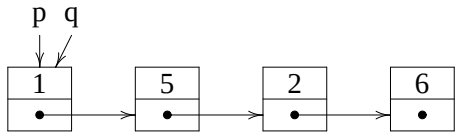
Ako lista ima više od jednog elementa, onda na neki način moramo da stignemo do njenog kraja. Uništavanjem poslednjeg elementa liste njen pretposlednji element postaje poslednji, pa i o tome moramo da vodimo računa.

```
procedure DropLast(var p : IntList);
var
  q, r : IntList;
begin
  if p↑.next = nil then begin
    dispose(p);
    p := nil
  end
  else begin
    q := p;
    repeat
      r := q;
      q := q↑.next
    until q↑.next = nil;
    r↑.next := nil;
    dispose(q)
  end
end;
```

Pogledajmo na jednom primeru kako radi procedura `DropLast` kada lista ima više od jednog elementa. Neka je `p` pokazivač na neku listu.

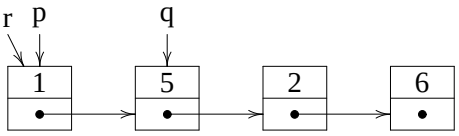



```
q := p;
```



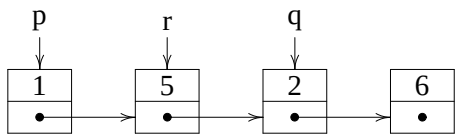
Izvrši se telo `repeat`-petlje:

```
r := q;
q := q↑.next
```



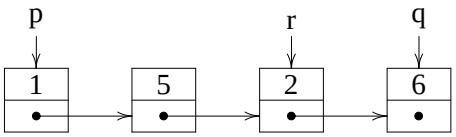
Pošto `q↑.next <> nil` ponovo se izvrši telo `repeat`-petlje:

```
r := q;
q := q↑.next
```



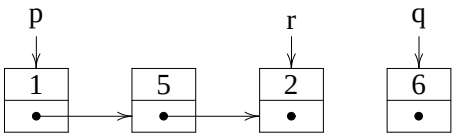
Još jednom se izvrši telo `repeat`-petlje:

```
r := q;
q := q↑.next
```

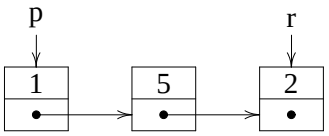


Sada je `q↑.next = nil`, što znači da smo stigli do poslednjeg elementa liste, pa se `repeat`-ciklus završava. Promenljiva `r` pokazuje na pretposlednji element liste. Na kraju "otkačimo" poslednji element liste od ostatka liste i uništimo ga.

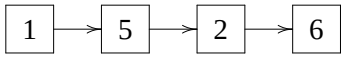
```
r↑.next := nil;
```



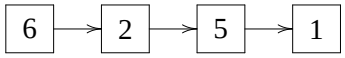

```
dispose(q)
```



Primer. Napisati proceduru koja okreće listu preuređivanjem pokazivača. Na primer, procedura treba od liste



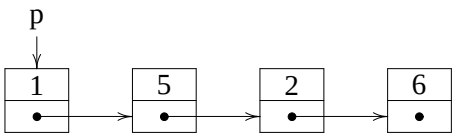
da napravi listu



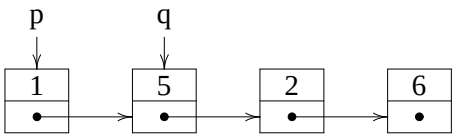

```
procedure Rev(var p : IntList);
var
  q, r : IntList;
begin
  if p <> nil then
    if p↑.next <> nil then begin
      q := p↑.next;
      p↑.next := nil;
      repeat
        r := p;
        p := q;
        q := q↑.next;
        p↑.next := r
      until q = nil;
    end
end;
```

Procedura `Rev` je veoma interesantna, pa ćemo detaljno pogledati kako ona radi. Pre svega, ako je `p = nil`, lista je prazna pa nemamo šta da okrenemo. Slično, ako je `p <> nil` ali je `p↑.next = nil`, lista ima samo jedan element, pa ponovo nema potrebe da se bilo šta okreće. Sada kada znamo da lista koju treba da okrenemo ima bar dva elementa, na primeru ćemo pokazati kako se prevezuju pokazivači da bi se dobila okrenuta lista.

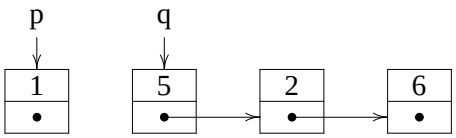
Neka je `p` pokazivač na listu kao u primeru pored →




```
q := p↑.next;
```

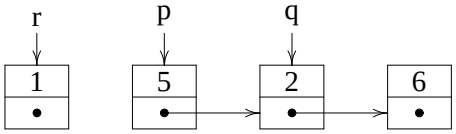



```
p↑.next := nil;
```

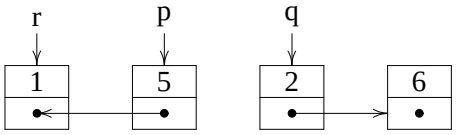


Uđemo u `repeat`-petlju:

```
r := p;
p := q;
q := q↑.next;
```

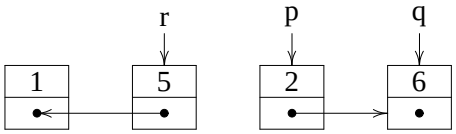



```
p↑.next := r
```

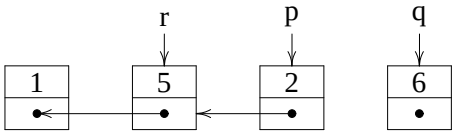


Pošto je `q <> nil`, ponovo se izvršava `repeat`-petlja:

```
r := p;
p := q;
q := q↑.next;
```

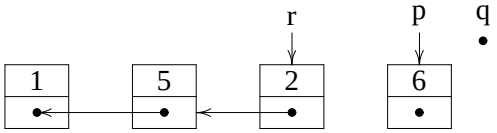



```
p↑.next := r
```

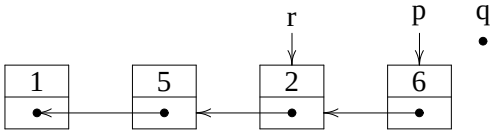


Još jednom se izvršava `repeat`-petlja:

```
r := p;
p := q;
q := q↑.next;
```




```
p↑.next := r
```



Sada je `q = nil` i `repeat`-petlja se završava, a time i procedura.

Primer. Napisati funkciju koja proverava da li su dve liste jednake. Dve liste su jednake ako imaju iste elemente koji se javljaju istim redom.

```
function Eq(p, q : IntList) : Boolean;
var
  ok : Boolean;
begin
  ok := true;
  while ok and (p <> nil) and (q <> nil) do begin
    ok := p↑.n = q↑.n;
    p := p↑.next;
    q := q↑.next
  end;
  Eq := ok and (p = nil) and (q = nil)
end;
```

Zadaci.

6.7. Napisati proceduru

```
procedure InsertLast(var p : IntList; n : integer);
```

koja ubacuje dati element na kraj date liste.

6.8. Napisati funkciju

```
function Factors(n : integer) : IntList;
```

koja za dati ceo broj $\geqslant 2$ vraća spisak prostih delilaca tog broja. Ukoliko je $n < 2$, funkcija vraća `nil`. Lista treba da bude sortirana od manjih ka većim brojevima.

6.9. Napisati funkciju

```
function Copy(p : IntList) : IntList;
```

koja pravi kopiju date liste.

6.10. Napisati proceduru

```
procedure DropFirstN(var p : IntList; n : integer);
```

koja iz liste izbacuje prvih n elemenata.

6.11. Napisati proceduru

```
procedure DropLastN(var p : IntList; n : integer);
```

koja iz liste izbacuje poslednjih n elemenata.

6.12. Napisati proceduru

```
procedure InsertAfter(var p : IntList; q : IntList;
                      n : integer);
```

koja ubacuje element `n` u listu `p` i to neposredno iza elementa na koga pokazuje `q`.

†**6.13.** Napisati proceduru

```
procedure InsertBefore(var p : IntList; q : IntList;
                       n : integer);
```

koja ubacuje element `n` u listu `p` i to neposredno ispred elementa na koga pokazuje `q`.

6.14. Napisati proceduru

```
procedure DropAfter(var p : IntList; q : IntList);
```

koja iz liste `p` izbacuje element koji se nalazi iza elementa na koga pokazuje `q`.

6.15. Napisati proceduru

```
procedure RemoveAll(var p : IntList; n : integer);
```

koja iz liste `p` izbacuje sva pojavljivanja elementa `n`.

6.16. Napisati proceduru

```
procedure InsertSorted(var p : IntList; n : integer);
```

koja ubacuje element `n` u sortiranu listu `p` i to tako da lista ostane sortirana. Lista `p` je sortirana od manjih ka većim brojevima.

6.17. Napisati proceduru

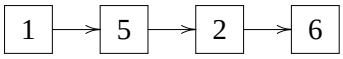
```
procedure ISort(var p : IntList);
```

koja sortira datu listu metodom *Insertion Sort*. Tokom procesa sortiranja nije dozvoljeno praviti nove dinamičke promenljive (nove kućice na heapu), već se sortiranje obavlja prevezivanjem pokazivača između već oformljenih kućica!

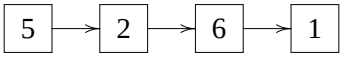
6.18. Napisati proceduru

```
procedure RotL(var p : IntList);
```

koja prvi element neprazne liste premešta na kraj. Na primer, procedura od liste



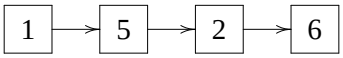
pravi listu



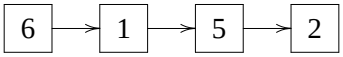
6.19. Napisati proceduru

```
procedure RotR(var p : IntList);
```

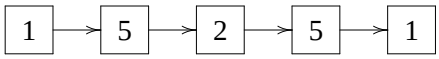
koja poslednji element neprazne liste premešta na početak liste. Na primer, procedura od liste



pravi listu



6.20. Lista celih brojeva je *palindrom* ako je njen prvi element jednak poslednjem, drugi jednak pretposlednjem, i tako dalje. Na primer, sledeća lista je palindrom:



Napisati Pascal funkciju koja proverava da li je data lista palindrom koristeći funkcije `Copy`, `Rev`, `Eq` i `Kill`;

6.21. Napisati funkciju

```
function Append(L1, L2 : IntList) : IntList;
```

koja nadovezuje listu `L2` na listu `L1`. Funkcija treba da napravi kopiju liste `L1` na čiji kraj treba da nadoveže kopiju liste `L2`.

6.22. Napisati funkciju

```
function CopyFirstN(p : IntList; n : integer) : IntList;
```

koja pravi kopiju prvih `n` elemenata liste `p`.

6.23. Napisati funkciju

```
function CopyLastN(p : IntList; n : integer) : IntList;
```

koja pravi kopiju poslednjih `n` elemenata liste `p`.

6.24. Napisati funkciju

```
function Copy2(p : IntList) : IntList;
```

koja pravi listu elemenata liste `p` koji se javljaju na parnim mestima u listi `p`.

6.25. Napisati funkciju

```
function CopyN(p : IntList; n : integer) : IntList;
```

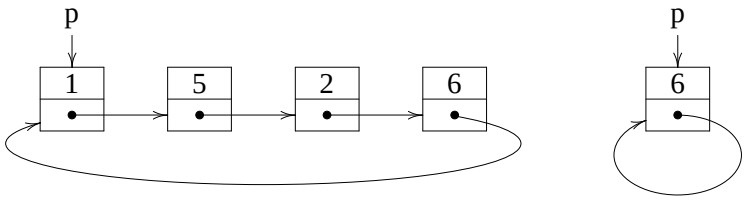
koja pravi listu elemenata liste `p` koji se javljaju na mestima u listi `p` čiji redni broj je deljiv sa `n`.

6.26. Napisati proceduru

```
procedure MergeInto(var L : IntList; p : IntList);
```

čiji argumenti su dve sortirane liste celih brojeva i koja listu `p` "umeša" u listu `L`. Obe ulazne liste kao i rezultat su sortirane od manjih brojeva ka većim brojevima. Tokom procesa nije dozvoljeno praviti nove dinamičke promenljive (nove kućice na heapu), već se sve obavlja prevezivanjem pokazivača između već oformljenih kućica!

6.27. *Kružna lista* je lista kod koje poslednji element liste pokazuje na prvi. Evo nekoliko kružnih listi:



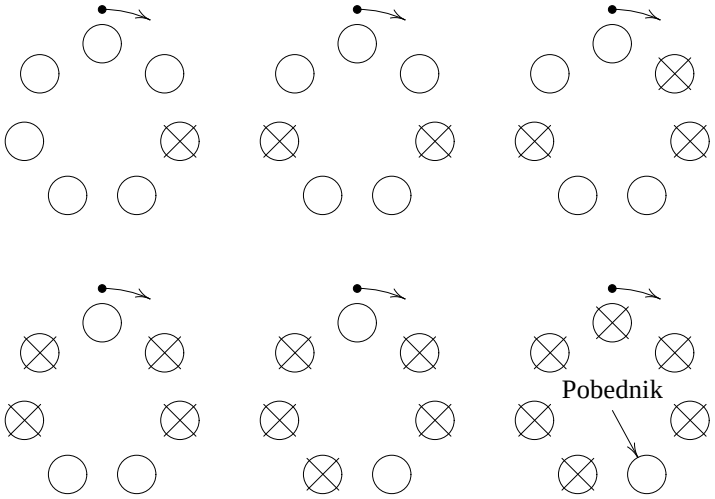
Napisati proceduru

```
procedure MkCirc(p : IntList);
```

koja od obične liste pravi kružnu listu.

6.28. (*Josifov problem*) Nekolikio dece stoji u krugu i igra sledeću igru. Polazimo od jednog od deteta, brojimo po k, gde je k pozitivan ceo broj, i svako k-to dete ispada iz kruga. Pobedilo je ono dete koje ostane poslednje. Na Slici 6.2 je prikazana ova procedura kada u krugu stoji 7 dece i kada se broji po 3. Napisati Pascal program koji od korisnika učitava pozitivne cele brojeve n i k, potom imena n dece i određuje pobednika kada tih n dece stoje u krugu onim redom kojim su uneti, a broji se po k. (Napomena: koristiti kružne liste!)

6.29. *Frekvencijski rečnik* je spisak reči neke tekstualne datoteke, zajedno sa brojem pojavljivanja reči u datoteci. Napisati Pascal program koji za datu tekstualnu datoteku formira njen frekvencijski rečnik. Frekvencijski rečnik upisati u tekstualnu datoteku `frekvrec.txt`. Na Slici 6.3 data je



Slika 6.2: Josifov problem

jedna datoteka i njen frekvencijski rečnik. Pošto ne znamo unapred broj reči u datoteci, reči unositi u sortiranu listu:

```
type
  ListaReci = ↑ElemListe;
  ElemListe = record
    rec : string;
    frekv : integer;
    next : ListaReci
  end;
```

petarpan.txt:

```
All children, except one,
grow up.  They soon know
that they will grow up,
and the way Wendy knew
was this.  One day when
she was two years old
she was playing in a
garden, and she plucked
another flower and ran
with it to her mother.
I suppose she must have
looked rather delightful,
for Mrs. Darling put her
hand to her heart and
cried, "Oh, why can't
you remain like this for
ever!" This was all that
passed between them on the
subject, but henceforth
Wendy knew that she must
grow up.  You always know
after you are two.  Two is
the beginning of the end.
```

frekvrec.txt:

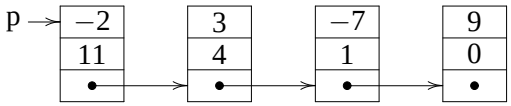
```
a 1              have 1          ran 1
after 1          heart 1         rather 1
all 2            henceforth 1    remain 1
always 1         her 3           she 5
and 4            i 1             soon 1
another 1        in 1            subject 1
are 1            is 1            suppose 1
beginning 1      it 1            that 3
between 1        knew 2          the 4
but 1            know 2          them 1
can't 1          like 1          they 2
children 1       looked 1        this 3
cried 1          mother 1        to 2
darling 1        mrs 1           two 3
day 1            must 2          up 3
delightful 1     of 1            was 4
end 1            oh 1            way 1
ever 1           old 1           wendy 2
except 1         on 1            when 1
flower 1         one 2           why 1
for 2            passed 1        will 1
garden 1         playing 1       with 1
grow 3           plucked 1       years 1
hand 1           put 1           you 3
```

Slika 6.3: Frekvencijski rečnik

6.30. Polinom se može predstaviti kao lista čiji elementi sadrže informacije o ne-nula monomima koji učestvuju u građenju polinoma:

```
type
  Polinom = ↑Monom;
  Monom = record
    koef : real;
    stepen : integer;
    next : Polinom
  end;
```

Na primer, polinom $-2x^{11}+3x^4-7x+9$ se može predstaviti listom ovako:



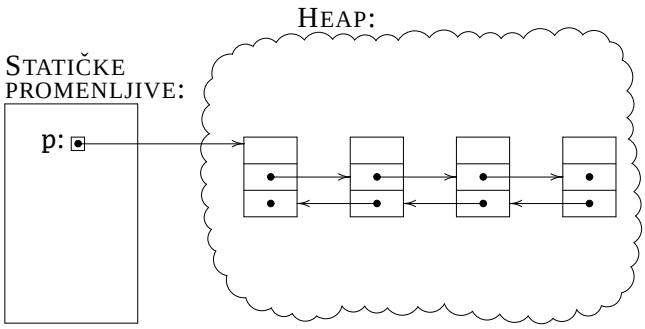
Primetimo još jednom da lista ne sadrži monome čiji koeficijent je 0. Na taj način se štedi na vremenu i prostoru!

Napisati sledeće funkcije koje manipulišu polinomima predstavljenim kao listama:

- `function AddPol(a, b : Polinom) : Polinom;` koja konstruiše novi polinom koji je jednak zbiru polinoma `a` i `b`;
- `function SubPol(a, b : Polinom) : Polinom;` koja konstruiše novi polinom koji je jednak razlici polinoma `a` i `b`;
- `function MulPol(a, b : Polinom) : Polinom;` koja konstruiše novi polinom koji je jednak proizvodu polinoma `a` i `b`;
- `function MulPolReal(a : Polinom; k : real) : Polinom;` koja konstruiše novi polinom koji je jednak proizvodu polinoma `a` i realnog broja `k`;
- `function EvalPol(a : Polinom; x : real) : real;` koja računa vrednost polinoma za datu vrednost nepoznate x;
- `function X2Pol(a : Polinom) : Polinom;` koja konstruiše polinom $\mathtt{a}(x^2)$;
- `function SubstPol(a, b : Polinom) : Polinom;` koja konstruiše polinom $\mathtt{a}(\mathtt{b}(x))$.

6.3 Dvostruko povezane liste

Dvostruko povezana lista je niz kućica na heapu gde svaka kućica pokazuje na onu iza sebe i na onu ispred sebe, kao na sledećoj slici:



Mnoge procedure za rad sa dvostruko povezanim listama su identične procedurama koje obavljaju analogne poslove sa jednostruko povezanim listama. Na primer, određivanje dužine dvostruko povezane liste, ispis dvostruko povezane liste ili uništavanje dvostruko povezane liste. Tako, za dvostruko povezane liste celih brojeva procedura koja ih uništava je *identična* proceduri koja uništava jednostruko povezane liste:

```
type
  IntDList = ↑IntDListElem;
  IntDListElem = record
    n : integer;
    next : IntDList;
    prev : IntDList
  end;
```

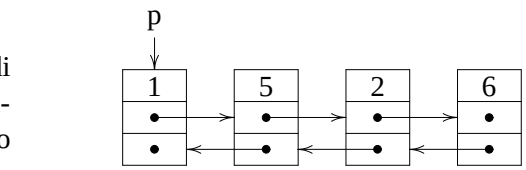
```
procedure Kill(var p : IntDList);
var
  q : IntDList;
begin
  while p <> nil do begin
    q := p;
    p := p↑.next;
    dispose(q)
  end
end;
```

S druge strane, mnoge procedure za rad sa dvostruko povezanim listama su jednostavnije od odgovarajućih procedura na jednostruko povezanim listama. To što u svakoj kućici postoje dva pokazivača znači da ćemo imati malo više knjigovodstva, ali su zato ideje algoritama prirodnije, jer u svakom trenutku možemo da se vratimo na prethodnu kućicu u nizu.

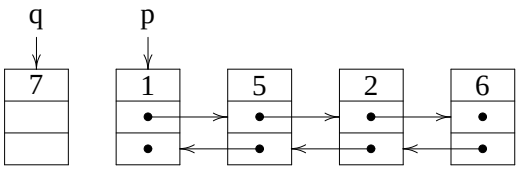
Primer. Napisati proceduru koja ubacuje dati element na početak date dvostruko povezane liste.

```
procedure Insert(var p : IntDList;
                     n : integer);
var
  q: IntDList;
begin
  new(q); q↑.n := n;
  if p = nil then begin
    q↑.next := nil;
    q↑.prev := nil
  end
  else begin
    q↑.next := p;
    q↑.prev := nil;
    p↑.prev := q
  end;
  p := q
end;
```

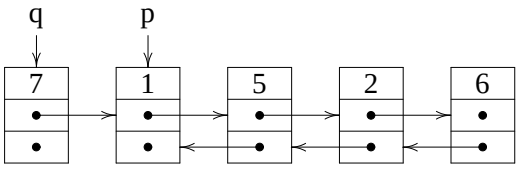
Pogledajmo na primeru kako radi procedura `Insert`. Neka je `p` pokazivač na dvostruko povezanu listu kao na slici pored.



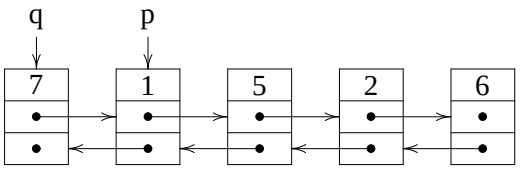

```
new(q);
q↑.n := n;
```



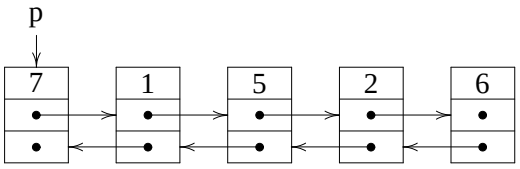

```
q↑.next := p;
q↑.prev := nil;
```




```
p↑.prev := q;
```




```
p := q;
```



Primer. Napisati proceduru koja iz date neprazne dvostruko povezane liste izbacuje njen prvi element.

```
procedure Drop(var p : IntDList);
var
  q : IntDList;
begin
  q := p;
  p := p↑.next;
  p↑.prev := nil;
  dispose(q)
end;
```

Primer. Lista celih brojeva je *palindrom* ako je njen prvi element jednak poslednjem, drugi jednak pretposlednjem, i tako dalje. Napisati Pascal funkciju koja vraća `true` ako je data lista palindrom.

Ovaj problem smo već rešavali za jednostruko povezane liste, a sada ćemo pokazati koliko je rešenje za dvostruko povezane liste jednostavnije. Ideja rešenja koje ćemo sada pokazati je da se pomoćni pokazivač `q` "odvoza na kraj liste", a onda `q` ide od kraja ka početku, dok `p` ide od početka ka kraju liste.

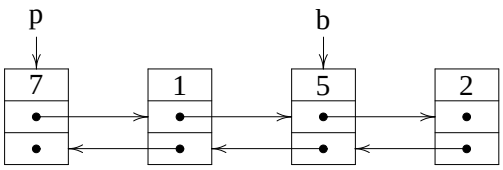
```
function Palind(p: IntDList): Boolean;
var
  q : IntDList;
  ok : Boolean;
begin
  if p = nil then
    Palind := true
  else begin
    q := p;
    while q↑.next <> nil do
      q := q↑.next;
    ok := true;
    while ok and (p <> nil) do begin
      ok := p↑.n = q↑.n;
      p := p↑.next;
      q := q↑.prev
    end;
    Palind := ok
  end
end;
```

Primer. Napisati proceduru koja ubacuje neki element u dvostruko povezanu listu pre elementa na koga pokazuje pomoćni pokazivač.

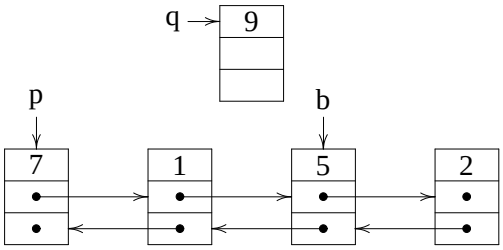
```
procedure InsertBefore(var p: IntDList;
  n: integer; b: IntDList);
var
  q : IntDList;
begin
  if (p = nil) or (b = p) then
    Insert(p, n)
  else begin
    new(q);
    q↑.n := n;
    q↑.next := b;
    q↑.prev := b↑.prev;
    b↑.prev↑.next := q;
    b↑.prev := q
  end
end;
```

Pogledajmo na primeru kako radi procedura `InsertBefore`.

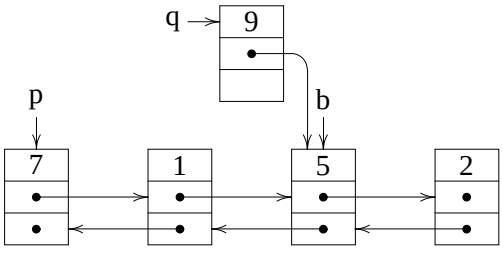
Neka je `p` pokazivač na dvostruko povezanu listu kao na slici pored i neka je `b` pokazivač na element te liste ispred koga treba ubaciti novi element.



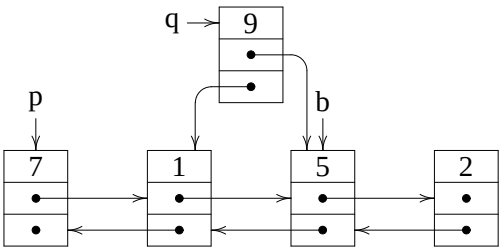

```
new(q);
q↑.n := n;
```



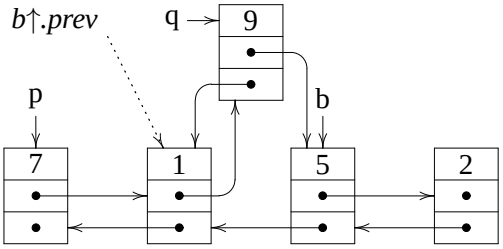

```
q↑.next := b;
```



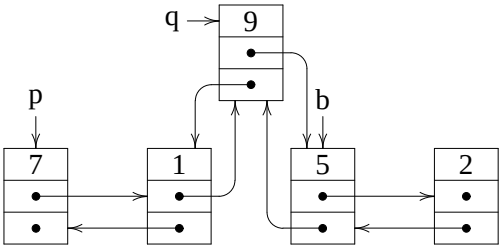
```
q↑.prev := b↑.prev;
```




```
b↑.prev↑.next := q;
```




```
b↑.prev := q;
```



Zadaci.

6.31. Napisati proceduru

```
procedure InsertLast(var p : IntDList; n : integer);
```

koja ubacuje dati element na kraj date liste.

6.32. Napisati proceduru

```
procedure InsertAfter(var p : IntDList; q : IntDList;
                      n : integer);
```

koja ubacuje element `n` u listu `p` i to neposredno iza elementa na koga pokazuje `q`.

6.33. Napisati proceduru

```
procedure DropLast(var p : IntDList);
```

koja iz liste izbacuje njen poslednji element.

6.34. Napisati proceduru

```
procedure DropAfter(var p : IntDList; q : IntList);
```

koja iz liste p izbacuje element koji se nalazi iza elementa na koga pokazuje q.

6.35. Napisati proceduru

```
procedure DropThis(var p : IntDList; q : IntList);
```

koja iz liste p izbacuje element na koga pokazuje q.

6.36. Napisati proceduru

```
procedure RemoveAll(var p : IntDList; n : integer);
```

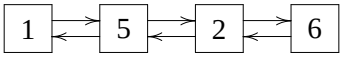
koja iz liste p izbacuje sva pojavljivanja elementa n.

6.37. Napisati proceduru `procedure Rev(var p : IntDList);` koja obrće dvostruko povezanu listu p.

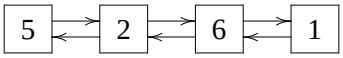
6.38. Napisati proceduru

```
procedure RotL(var p : IntDList);
```

koja prvi element neprazne liste premešta na kraj. Na primer, procedura od liste



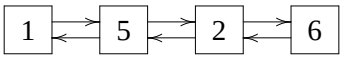
pravi listu



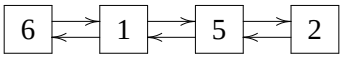
6.39. Napisati proceduru

```
procedure RotR(var p : IntDList);
```

koja poslednji element neprazne liste premešta na početak liste. Na primer, procedura od liste



pravi listu

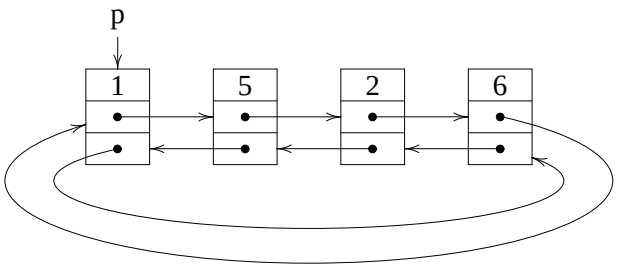


6.40. Napisati funkciju

```
function Append(L1, L2 : IntDList) : IntDList;
```

koja nadovezuje listu L2 na listu L1. Funkcija treba da napravi kopiju liste L1 na čiji kraj treba da nadoveže kopiju liste L2.

6.41. *Kružna dvostruko povezana lista* je dvostruko povezana lista kod koje poslednji element liste pokazuje na prvi, i prvi pokazuje na poslednji. Na primer:

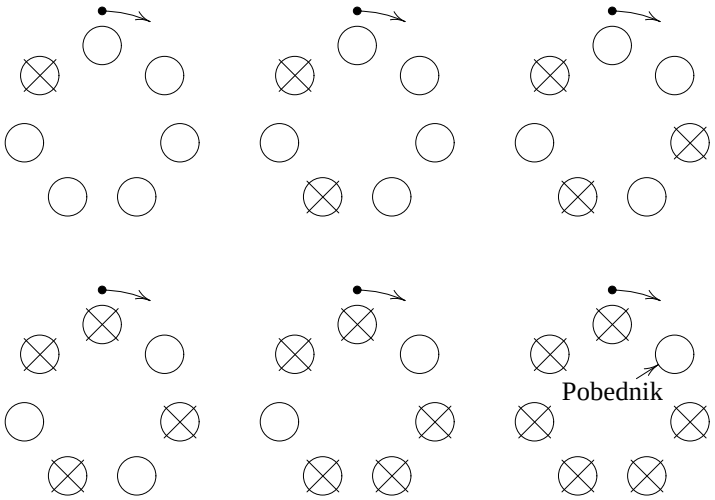


Napisati proceduru

```
procedure MkCirc(p : IntDList);
```

koja od obične dvostruko povezane liste pravi kružnu dvostruko povezanu listu.

6.42. (*Dvosmerni Josifov problem*) Nekolikio dece stoji u krugu i igra sledeću igru. Polazimo od jednog od deteta, brojimo po k, gde je k *proizvoljan nenula* ceo broj, i svako k-to dete ispada iz kruga. Pobedilo je ono dete koje ostane poslednje. Na Slici 6.4 je prikazana ova procedura kada u krugu stoji 7 dece i kada se broji po -2. Napisati Pascal program koji od korisnika učitava pozitivne cele brojeve n i k, potom imena n dece i određuje pobednika kada tih n dece stoje u krugu onim redom kojim su uneti, a broji se po k. (Napomena: koristiti kružne dvostruko povezane liste!)



Slika 6.4: Josifov problem sa negativnim brojanjem

Glava 7

Rekurzija
Podeli pa vladaj

U ovoj glavi uvodimo rekurziju kao jedno izuzetno značajnu programersku tehniku. Kao primer tehnike programiranja poznate pod imenom *podeli pa vladaj* pokazujemo dva superiorna sorta – Quicksort i Mergesort. Nakon Quicksort algoritma pokazujemo kako se njegova osnovna ideja može upotrebiti za efikasno nalaženje k-tog elementa u nizu.

7.1 Rekurzija

Za neku proceduru ili funkciju kažemo da je *rekurzivna* ako se u telu javlja poziv te iste procedure ili funkcije za neke druge, obično "manje" vrednosti parametara. Uopšte, rekurzivne procedure ili funkcije se pišu kada posao za n možemo nekako da odradimo preko istih poslova za $n-1$ ili neke druge brojeve manje od n. Prilikom pisanja rekurzivnih procedura i funkcija moramo imati precizno formulisane odgovore na sledeća pitanja:

- kako se tačno posao za n može odraditi preko tog istog posla, ali za manje vrednosti; i
- kada se izlazi iz rekurzije, tj. kako izgleda taj posao za prvih nekoliko vrednosti broja n koje su prirodno mogu pojaviti.

Kada nam je ovo jasno, rekurzivna procedura (ili funkcija) izgleda ovako:

```
procedure P(n : integer);
begin
    if n = neka od početnih vrednosti za koje je sve trivijalno then
        odradi posao za odgovarajuću vrednost
    else
        svedi posao za n na poslove za vrednosti manje od n
        pozivajući istu ovu proceduru
end;
```

Primer. Napisati funkciju koja rekurzivno računa $n!$.

Ovde prvo treba da shvatimo kako se $n!$ može izračunati preko manjih vrednosti. Primetimo da je

```
function Fakt(n: integer): integer;
begin
  if n = 0 then
    Fakt := 1
  else
    Fakt := n * Fakt(n - 1)
end;
```

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1 = n \cdot (n-1)!$$

Eto načina! Faktorijel broja n se računa tako što se prvo izračuna faktorijel broja $n-1$, pa se on pomnoži sa n. Početni uslov je, naravno, $0! = 1$.

Primer. Napisati funkciju `Fib(n)` koja rekurzivno računa n-ti Fibonaccijev broj.

Posao računanja Fibonaccijevog broja F_n se može svesti na računanje Fibonaccijevih brojeva za manje vrednosti na poznati način:

```
function Fib(n: integer): integer;
begin
  if n = 0 then
    Fib := 0
  else if n = 1 then
    Fib := 1
  else
    Fib := Fib(n - 1) + Fib(n - 2)
end;
```

$$F_n = F_{n-1} + F_{n-2},$$

dok za prve dve vrednosti imamo: $F_0 = 0$ i $F_1 = 1$.

Primer. Napisati funkciju `A(n)` koja računa brojeve A_n rekurzivno zadate na sledeći način: $A_1 = 1$, $A_2 = 2$, $A_n = nA_{n-1} - 3A_{n-2} + 3$.

```
function A(n: integer): integer;
begin
  if n = 1 then
    A := 1
  else if n = 2 then
    A := 2
  else
    A := n*A(n-1) - 3*A(n-2) + 3
end;
```

Primer izvršavanja rekurzivnog potprograma. Pogledajmo kako se izvršava rekurzivan potprogram na primeru računanja faktorijela broja 4.

Krenemo da računamo `Fakt(4)`. Za to nam je potrebna vrednost `Fakt(3)`. Zato funkcija `Fakt` poziva sama sebe, ali ovaj put za $n = 3$.

```
Fakt(4) →
    Fakt := 4 * Fakt(3)
```

Dalje, `Fakt(3)` poziva sebe ponovo za $n = 2\ldots$

```
Fakt(4) →
    Fakt := 4 * Fakt(3)
Fakt(3) →
    Fakt := 3 * Fakt(2)
```

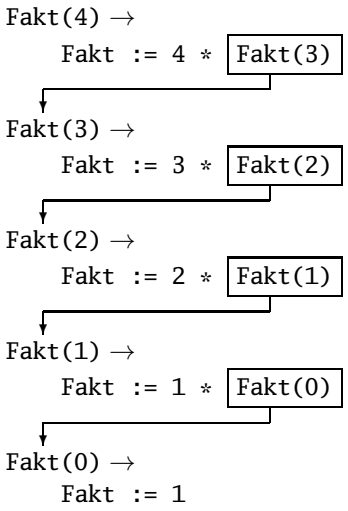
..., `Fakt(2)` poziva sebe za $n = 1\ldots$

```
Fakt(4) →
    Fakt := 4 * Fakt(3)
Fakt(3) →
    Fakt := 3 * Fakt(2)
Fakt(2) →
    Fakt := 2 * Fakt(1)
```

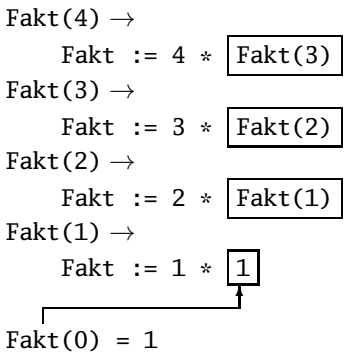
..., `Fakt(1)` poziva sebe za $n = 0\ldots$

```
Fakt(4) →
    Fakt := 4 * Fakt(3)
Fakt(3) →
    Fakt := 3 * Fakt(2)
Fakt(2) →
    Fakt := 2 * Fakt(1)
Fakt(1) →
    Fakt := 1 * Fakt(0)
```

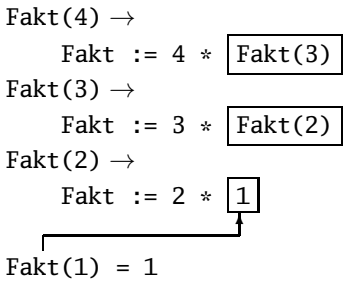
..., a `Fakt(0)` ne poziva više nikoga, jer je $n = 0$ izlaz iz rekurzije. Funkcija sada *zna* da je `Fakt(0) = 1`.



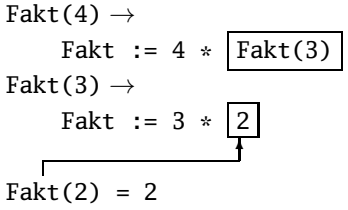
Počinje vraćanje vrednosti. Poziv `Fakt(0)` je završen i ova funkcija vraća odgovarajuću vrednost u izraz iz koga je pozvana.



Lanac *vraćanja iz rekurzivnih poziva* se dalje odmotava...



..., i dalje...



..., i dalje...

```
Fakt(4) →
    Fakt := 4 * 6
Fakt(3) = 6
```

..., i tako dobijamo konačnu vrednost: 24.

```
Fakt(4) = 24
```

Sa upotrebom rekurzije treba biti obazriv. Rekurzivni programi mogu da budu neefikasniji od odgovarajućih nerekurzivnih verzija. Na primer, rekurzvno računanje Fibonaccijevih brojeva je neuporedivo sporije od računanja pomoću "for" ciklusa koga smo videli ranije. Takođe, ako se početni uslovi rekurzije (koji se zovu još i *izlazni kriterijumi*) ne postave kako treba, program veoma lako može da zaglavi. Kada nije neefikasna, upotreba rekurzije je veoma pogodna za rešavanje problema, zato što najčešće dovodi do elegantnih rešenja.

Primer. Napisati proceduru `IspisiBin(n)` koja dati nenegativan ceo broj ispisuje u binarnom sistemu.

Pre svega treba da shvatimo kako se binarni zapis broja n može dobiti na osnovu binarnog zapisa nekog manjeg broja. Pogledajmo kako teče konverzija broja n u osnovu 2 "peške": prvo n podelimo sa dva; ostatak "napišemo iza crte", a količnik dalje konvertujemo u binarnu osnovu. Zato što se dobijeni niz cifara čita od poslednjeg ka prvom, ostatak koga smo prvog zapisali "iza crte" bude poslednja cifra. Eto pravila! Konverzija broja n u binarni zapis ide ovako: prvo se n `div` 2 konvertuje u binarni zapis, pa se na to dopiše n `mod` 2. Izlazni kriterijum je jednostavan: ako je $n \in \{0,1\}$ to je to (samo ga ispišemo).

```
procedure IspisiBin(n : integer);
begin
  if (n = 0) or (n = 1) then
    write(n)
  else
    begin
      IspisiBin(n div 2);
      write(n mod 2)
    end
end;
```

Primer. Napisati Paskal program koji od korisnika učitava prirodne brojeve n i M, potom n brojeva $c_1, \ldots, c_n$ i proverava da li se umetanjem znakova $+$ i $-$ ispred ili između nekih od unetih brojeva može dobiti aritmetički izraz čija vrednost je M. Na primer za $n = 4$, $M = 989$ i brojeve 1, 10, 100, 1000 izraz $-1-10+1000$ ima vrednost 989.

Rešenje.

```
program MiniSlagalica;
const
  MaxN = 50;
var
  n, i : integer;
  M : longint;
  c : array [1 .. MaxN] of integer;
  op : array [0 .. MaxN] of char;
  ok : boolean;

procedure IspisiIzraz;
var
  i : integer;
begin
  for i := 1 to n do
    if op[i] = '0' then
      { nista ne radimo -- '0' znaci da se odgovarajuci broj preskace }
    else if (op[i] = '+') then
      if i = 1 then write(c[i])
      else if c[i] >= 0 then write(' + ', c[i])
      else write(' + (', c[i], ')')
    else { op[i] = '-' }
      if c[i] < 0 then write(' - (', c[i], ')')
      else write(' - ', c[i]);
  writeln
end;

procedure Pronadji(n : integer; M : longint);
begin
  if n = 0 then
    begin
      ok := M = 0;
      if ok then IspisiIzraz
    end
  else begin
    op[n] := '+';
    Pronadji(n - 1, M - c[n]);
    if not ok then begin
```

```
      op[n] := '-';
      Pronadji(n - 1, M + c[n]);
      if not ok then begin
        op[n] := '0';
        Pronadji(n - 1, M);
      end
    end
  end
end;

begin
  write('Koliko ima brojeva -> '); readln(n);
  write('Brojevi -> ');
  for i := 1 to n do read(c[i]);
  readln;
  write('Rezultat -> '); readln(M);
  ok := false;
  Pronadji(n, M);
  if not ok then
    writeln('Ne moze')
end.
```

Zadaci.

7.1. Napisati rekurzivnu Pascal funkciju koja računa niz brojeva b_n zadat ovako: $b_0 = 0$, $b_1 = 2$, $b_2 = 3$, i $b_n = n^2 b_{n-1} + b_{n-3}^2 - 3n$.

7.2. Napisati rekurzivnu Pascal funkciju koja računa niz brojeva d_n zadat ovako: $d_0 = 0$, $d_1 = 5$, i

$$d_n = \begin{cases} d_{n-1} + 2d_{n-2}, & n \text{ parno} \\ 4d_{n-1} - nd_{n-2}, & n \text{ neparno.} \end{cases}$$

7.3. Napisati Pascal program koji računa *NZD* dva pozitivna broja na sledeći način:

$$NZD(m,n) = \begin{cases} m, & m = n \\ NZD(m-n,n), & m > n \\ NZD(n-m,m), & n > m \end{cases}$$

7.4. Napisati rekurzivnu proceduru `IspisiUOsnovi(n, b)` koja nenegativan ceo broj `n` ispisuje u osnovi `b`.

7.5. Za realan broj x i prirodan broj n, broj x^n se može izračunati ovako:

$$x^n = \begin{cases} 1, & n = 0 \\ x, & n = 1 \\ x \cdot (x^k)^2, & n = 2k+1 \\ (x^k)^2, & n = 2k. \end{cases}$$

Koristeći ovo i Pascalovu ugrađenu funkciju `sqr`, napisati Pascal program koji rekurzivno računa x^n.

7.6. Niz brojeva $b_0, b_1, b_2, \ldots, b_n, \ldots$ zadat je ovako:

$$\begin{aligned} &b_0 = -1, \quad b_1 = 1, \\ &b_n = b_p + b_q, \quad \text{za } n \geqslant 2, \end{aligned}$$

gde je p broj nula u binarnom zapisu broja n, a q broj jedinica u binarnom zapisu broja n. Napisati Paskal program koji od korisnika učitava nenegativan broj n tipa `longint` i potom računa i štampa broj b_n.

7.7. Napisati Pascal program koji rekurzivno računa količnik m/n celih brojeva m i n i ispisuje njegovih prvih k decimala.

7.8. Verižni razlomak je razlomak oblika

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots \\ a_{n-1} + \cfrac{1}{a_n}}}}}$$

gde su a_k neki ne-nula realni brojevi.

(a) Napisati Pascal program koji od korisnika učitava n, potom $n+1$ realnih brojeva a_0, a_1, …, a_n, i onda računa odgovarajući verižni razlomak *nerekurzivno* (obična “for” petlja).

(b) Napisati Pascal program koji od korisnika učitava n, potom $n+1$ realnih brojeva a_0, a_1, …, a_n, i onda računa odgovarajući verižni razlomak *rekurzivno*.

7.9. Napisati proceduru koja za dato n ispisuje niz brojeva koji se formira na sledeći način:

$n = 1$: 1
$n = 2$: 1, 2, 1
$n = 3$: 1, 2, 1, 3, 1, 2, 1
$n = 4$: 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1

Pravilo za formiranje niza izgleda ovako:

$$\boxed{\text{niz za } n} = \boxed{\text{niz za } n-1},\quad n,\quad \boxed{\text{niz za } n-1}.$$

7.10. Grayov kod reda n je niz koji se sastoji od svih 01-reči dužine n (kojih ima 2^n) sa osobinom: svake dve susedne reči se razlikuju samo na jednom mestu. Na primer, Grayov kod reda 4 izgleda ovako:

0000	0110	1100	1010
0001	0111	1101	1011
0011	0101	1111	1001
0010	0100	1110	1000

Napisati Pascal program koji od korisnika učitava pozitivan ceo broj n i potom štampa Grayov kod reda n, svaku reč u novom redu.

(Napomena: Naredni element Grayovog koda se formira tako što se u prethodnom elementu komplementira jedan bit. Niz bitova koje treba komplementirati je upravo niz koji je formiran u prethodnom zadatku.)

7.11. Particija broja n je svaki način da se broj n predstavi u obliku zbira prirodnih brojeva. Na primer, sve particije broja 5 su

$$\begin{aligned} 5 &= 1+1+1+1+1 \\ &= 1+1+1+2 \\ &= 1+2+2 \\ &= 1+1+3 \\ &= 2+3 \\ &= 1+4 \\ &= 5 \end{aligned}$$

Napisati Pascal program koji za dati prirodan broj n ispisuje broj particija broja n. Broj particija broja n jednak je $f(n,n)$, gde je

$$f(m,n) = \begin{cases} 1, & m = 1 \text{ ili } n = 1 \\ f(m,m), & m < n \\ f(m,m-1)+1, & m = n \\ f(m,n-1)+f(m-n,n), & m > n. \end{cases}$$

7.12. Napisati Pascal program koji od korisnika učitava prirodne brojeve n i k, potom k različitih prirodnih brojeva $a_1, a_2, \ldots, a_k$, i ispisuje sve načine na koje se dati broj n može predstaviti kao zbir nekih od brojeva $a_1, \ldots, a_k$. Svaki broj a_i se može pojaviti u reprezentaciji proizvoljan broj puta.

7.13. *(Hanojske kule)* Data su tri štapa, A, B, C. Na štapu A je nanizano n diskova (sa rupom u sredini). Svi diskovi su različitog prečnika i na štapu su nanizani po veličini, pri čemu je naveći na dnu, a najmanji na vrhu. Jedan potez u igri se sastoji u tome da se disk koji je na vrhu neke gomile prebaci sa jednog štapa na neki drugi štap. Pri čemu je zabranjeno da se disk većeg prečnika nalazi iznad diska manjeg prečnika. Zadatak igre je da se svi diskovi prebace sa štapa A na štap C koristeći štap B kao pomoćni.

Napisati Pascal program koji od korisnika učitava n – broj diskova na nizu A i ispisuje redosled premeštanja diskova. Na primer, za $n = 3$ treba ispisati sledeće:

Sa A na C	Sa B na A
Sa A na B	Sa B na C
Sa C na B	Sa A na C
Sa A na C	

Koristiti sledeću ideju: n diskova se sa A na C može prebaciti tako što se prebaci $n-1$ disk sa A na B, potom se izvrši potez `Sa A na C`, i onda se $n-1$ diskova prebaci sa B na C.

†**7.14.** Dokazati da ideja opisana u prethodnom zadatku dovodi do rešenja problema Hanojskih kula sa n diskova u $2^n - 1$ poteza.

(Uputstvo: koristiti matematičku indukciju.)

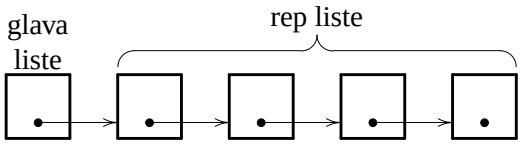
†**7.15.** Konveksni poligon je nekim svoim dijagonalama razbijen na manje konveksne poligone, a pri tome ne postoje dve dijagonale koje imaju zajedničku unutrašnju tačku (drugim rečima, ako se dve dijagonale seku, seku se u temenu poligona). Od svih potpoligona koji u svojoj unutrašnjosti ne sadrže nijednu dijagonalu odrediti onaj koji ima najviše temena.

†**7.16.** Napisati Pascal program koji od korisnika učitava prirodne brojeve n i M, potom n cifara $c_1, \ldots, c_n$ i proverava da li se umetanjem znakova $+$ između cifara može dobiti aritmetički izraz čija vrednost je M. Na primer za $n = 8$, $M = 1000$ i cifre 8 8 8 8 8 8 8 8 (osam osmica), umetanjem znakova $+$ na sledeći način: $888 + 88 + 8 + 8 + 8$ dobija se izraz čija vrednost je 1000.

7.2 Liste i rekurzija

Svaka neprazna lista se na prirodan način može podeliti na dva dela:

- *glava liste*, što je prvi element liste, i
- *rep liste*, što je lista koju čine svi ostali elementi liste.



Tako dobijamo da su liste rekurzivne strukture: lista dužine $l > 0$ jednoznačno je određena svojim prvim elementom (njena glava) i jednom listom dužine $l-1$ (njen rep). Zato ne iznenađuje što se mnogi algoritmi na listama veoma jednostavno implementiraju rekurzivno.

Kao i do sada, uglavnom ćemo raditi sa listama celih brojeva čija definicija je data pored.

```
type
  IntList = ↑IntListElem;
  IntListElem = record
    n : integer;
    next : IntList
  end;
```

Dakle,

☞ *Ako je* `p` *pokazivač na nepraznu listu celih brojeva, onda je* `p↑.n` *glava liste, a* `p↑.next` *pokazivač na njen rep.*

Primer. Napisati funkciju koja rekurzivno određuje dužinu celobrojne liste.

Rešenje:

- Dužina prazne liste je 0;
- Dužina neprazne liste = = 1 + dužina njenog repa.

```
function Len(p : IntList) : integer;
begin
  if p = nil then
    Len := 0
  else
    Len := 1 + Len(p↑.next)
end;
```

Primer. Napisati funkciju koja rekurzivno određuje sumu elemenata celobrojne liste.

Rešenje:

- Suma prazne liste je 0;
- Suma neprazne liste = glava liste + suma njenog repa.

```
function Sum(p : IntList) : integer;
begin
  if p = nil then
    Sum := 0
  else
    Sum := p↑.n + Sum(p↑.next)
end;
```

Primer. Napisati proceduru koja rekurzivno uništava celobrojnu listu i oslobađa prostor na heapu koji je bio rezervisan za njene elemente.

Rešenje:

- Kod prazne liste ne treba ništa da uništavamo;
- Nepraznoj listi rekurzivno uništimo rep, pa onda unižtimo glavu.

```
procedure Kill(var p : IntList);
begin
  if p <> nil then begin
    Kill(p↑.next);
    dispose(p);
    p := nil
  end
end;
```

Rekurzivni programi sa listama su jasniji, elegantniji i jednostavniji. Poređenja radi, na Slici 7.1 pokazane su nerekurzivna i rekurzivna verzija ova tri potprograma.

Zadaci.

7.17. Napisati funkciju

```
function Member(p : IntList; n : integer) : Boolean;
```

koja rekurzivno proverava da li se dati element nalazi u listi celih brojeva.

7.18. Napisati proceduru

```
procedure CountMin(p : IntList; var min, br : integer);
```

koja rekurzivno određuje koliko se puta najmanji element liste celih brojeva `p` pojavljuje u toj listi. Argument `min` treba da vrati najmanji element liste, a argument `br` broj pojavljivanja elementa `min` u listi. Pretpostavlja se da je lista neprazna.

7.19. Napisati proceduru

```
procedure InsertLast(var p : IntList; q : IntList);
```

koja rekurzivno ubacuje kućicu na koju pokazuje `q` na kraj liste `p`.

7.20. Napisati proceduru

```
procedure DropFirstN(var p : IntList; n : integer);
```

koja iz liste rekurzivno izbacuje prvih `n` elemenata.

7.21. Napisati proceduru

```
procedure RemoveAll(var p : IntList; n : integer);
```

koja iz liste `p` rekurzivno izbacuje sva pojavljivanja elementa `n`.

†**7.22.** Napisati proceduru

```
function Len(p: IntList): integer;
var
  n: integer;
begin
  n := 0;
  while p <> nil do begin
    n := n + 1;
    p := p↑.next
  end;
  Len := n
end;
```

```
function Len(p: IntList): integer;
begin
  if p = nil then
    Len := 0
  else
    Len := 1 + Len(p↑.next)
end;
```

```
function Sum(p: IntList): integer;
var
  n: integer;
begin
  n := 0;
  while p <> nil do begin
    n := n + p↑.n;
    p := p↑.next
  end;
  Sum := n
end;
```

```
function Sum(p: IntList): integer;
begin
  if p = nil then
    Sum := 0
  else
    Sum := p↑.n + Sum(p↑.next)
end;
```

```
procedure Kill(var p: IntList);
var
  q: IntList;
begin
  while p <> nil do begin
    q := p;
    p := p↑.next;
    dispose(q)
  end
end;
```

```
procedure Kill(var p: IntList);
begin
  if p <> nil then begin
    Kill(p↑.next);
    dispose(p);
    p := nil
  end
end;
```

Slika 7.1: Nerekurzivna i rekurzivna verzija tri potprograma

```
procedure Rev(var p : IntList);
```

koja rekurzivno obrće listu.

†**7.23.** Napisati funkciju

```
function Copy(p : IntList) : IntList;
```

koja rekurzivno kopira listu.

7.24. Napisati proceduru

```
procedure InsertSorted(var p : IntList; q :  IntList);
```

koja rekurzivno umeće broj kućicu na koju pokazuje `q` u sortiranu listu `p`. Lista `p` nakon umetanja treba da ostane sortirana.

7.25. Napisati proceduru

```
procedure ISort(var p : IntList);
```

koja sortira listu `p` algoritmom *insertion sort*. Tokom procesa sortiranja nije dozvoljeno praviti nove dinamičke promenljive (nove kućice na heapu), već se sortiranje obavlja prevezivanjem pokazivača između već oformljenih kućica!

7.3 Podeli pa vladaj

Divide et impera je bila maksima rimskih vladara kojom su uspevali da pokore i najtvrdokornije neprijatelje. U programiranju se strategija "Podeli pa vladaj" (engl. Divide and conquer) koristi za efikasno rešavanje teških problema i sastoji se u tome da se se komplikovan problem podeli na dva potproblema približno iste veličine koji se onda nezavisno rešavaju, pa se na osnovu tako dobijenih parcijalnih rešenja rekonstruiše konačno rešenje. Mi ćemo strategiju demonstrirati na tri primera: Quicksort algoritam, traženje k-tog po veličini elementa niza (što je modifikacija Quicksort algoritma), i Mergesort algoritam.

7.3.1 Quicksort

Sada ćemo pokazati kako radi jedan superiorni algoritam za sortiranje – Quicksort. Interesantno je da je on u osnovi rekurzivan. Osnovna ideja Quicksort algoritma je ujedno i jedna od osnovnih ideja programiranja uopšte koja, kada može da se primeni, daje veoma dobre rezultate, a zove se *divide and conquer* (podeli pa

vladaj)[1]. Neka nam je dat niz brojeva

$$3, 2, 6, 5, 7, 1, 8, 1, 1, 6, 3, 6, 7.$$

U odnosu na prvi element niza (*pivot*) sve ostale elemente razvrstamo u dve grupe: na one koji su manji od njega ili jednaki sa njim, i na one koji su veći od njega. Dobijamo dva kraća niza:

$$2,\ 1,\ 1,\ 1,\ 3 \qquad \text{i} \qquad 6,\ 5,\ 7,\ 8,\ 6,\ 6,\ 7$$

Ova dva niza sortiramo (rekurzivno) svakog posebno i onda ih samo nadovežemo tako što ispišemo elemente prvog niza, potom pivot – broj 3, i na kraju elemente drugog niza. U prethodnom primeru, nakon sortiranja dobijamo dva niza:

$$1,\ 1,\ 1,\ 2,\ 3 \qquad \text{i} \qquad 5,\ 6,\ 6,\ 6,\ 7,\ 7,\ 8,$$

a nakon spajanja niz

$$1,\ 1,\ 1,\ 2,\ 3,\ \ 3,\ \ 5,\ 6,\ 6,\ 6,\ 7,\ 7,\ 8.$$

Ova ideja se može implementirati rako da se sortiranje obavlja *u istom nizu*. Preuredićemo elemente niza u odnosu na prvi element tako da se ispred njega nalaze oni koji su manji ili jednaki sa njim, a iza njega oni koji su strogo veći od njega. Onda pozovemo proceduru da sortira samo odgovarajuće delove velikog niza.

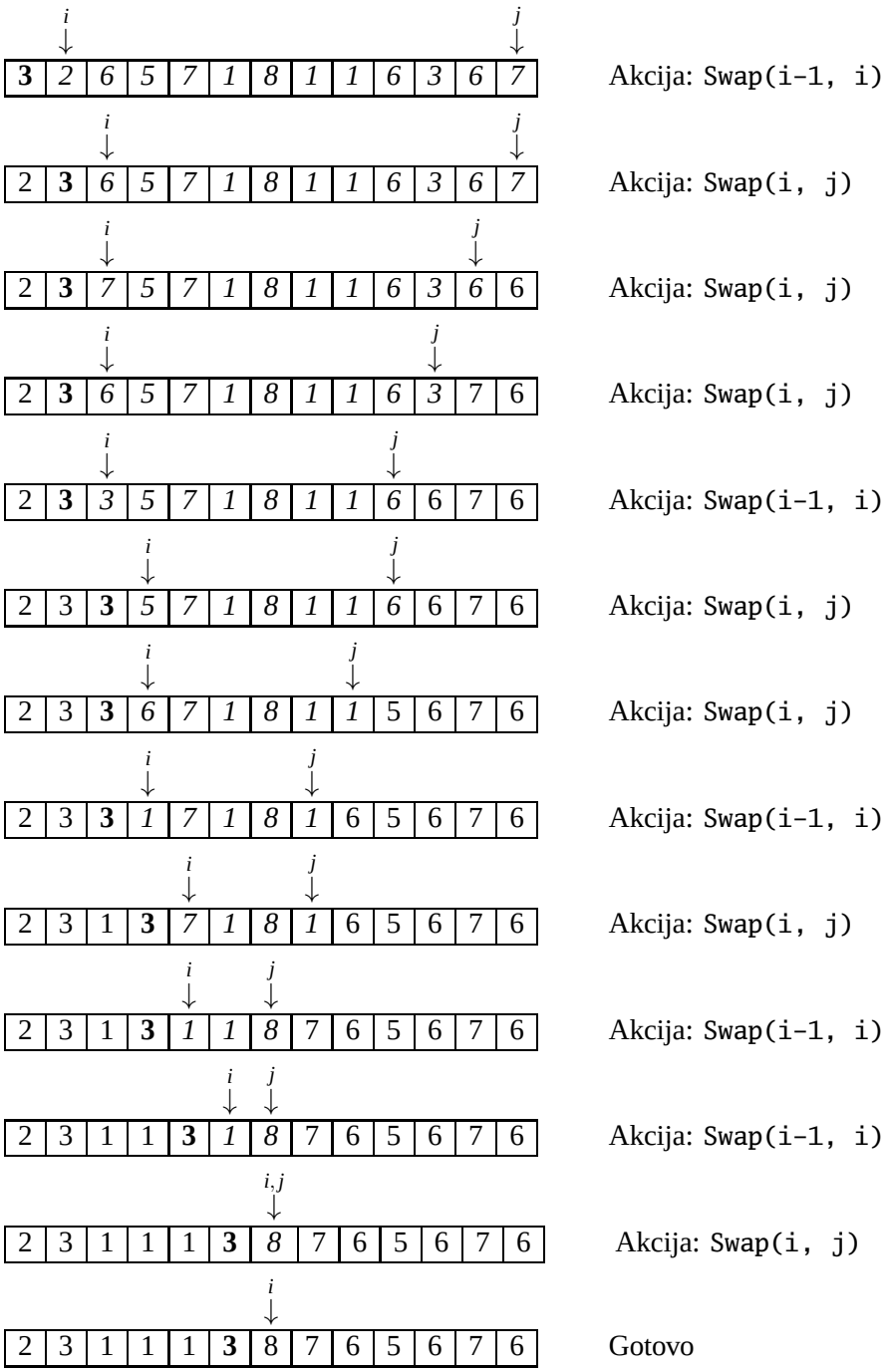
Za pivota uzimamo prvi element niza i gledamo element koji je neposredno desno od njega. Ako je on manji od pivota ili jednak sa njim, samo im zamenimo mesta. Ako je veći od pivota, prebacimo ga na kraj niza. Prebacivanje na kraj niza takođe moramo malo organizovati. Na kraju niza se nalaze elementi koje još nismo poredili sa pivotom. O njima razmišljamo kao o "slobodnim" elementima. Tako prebacivanje na kraj niza zapravo svodimo na razmenu mesta sa najdesnijim slobodnim elementom. Element koji na taj način stigne na kraj niza više nije slobodan zato što on *mora* ostati desno od pivota.

Na ovaj način ne samo da štedimo memoriju, već i vreme: razvrstavanje se svodi na lukavo organizovano razmeštanje elemenata unutar istog niza, a faza spajanja *ne postoji!*

Primer. Pogledajmo na primeru kako funkcioniše razvrstavanje. Pivot je podebljan, a "slobodni" elementi su prikazani iskošeno. Brojač *i* pokazuje na element koga upravo poredimo sa pivotom i razmatramo šta sa njim da radimo, a brojač

[1] latinski: Divide et impera

j na najdesniji slobodan element. `Swap(p, q)` je procedura koja u datom nizu zamenjuje elemente na mestima `p` i `q`.



Program. Neka su nam date sledeće deklaracije:

```
const
  MaxEl = 1000;

type
  Niz = array [1 .. MaxEl] of integer;

var
  a : Niz;
```

Procedura `Swap` koja zamenjuje elemente mestima `i` i `j` globalnog niza `a` je krajnje jednostavna:

```
procedure Swap(i, j : integer);
var
  t : integer;
begin
  t := a[j]; a[j] := a[i]; a[i] := t
end;
```

Srce algoritma je procedura `Split(p, q, k)` koja preuređuje segment `a[p .. q]` niza `a` tako da u novom rasporedu svi elementi iz `a[p .. k-1]` budu manji od pivota `a[p]` iz starog niza, da svi elementi iz `a[k+1 .. q]` budu veći ili jednaki sa pivotom `a[p]` iz starog niza, i da pivot u novom rasporedu bude na poziciji `k`.

```
procedure Split(p, q : integer; var k : integer);
var
  i, j, pivot : integer;
begin
  pivot := a[p];
  i := p + 1; j := q;
  while i <= j do
    if a[i] < pivot then
      begin
        Swap(i-1, i); i := i + 1
      end
    else if a[i] = pivot then i := i + 1
    else { a[i] > pivot }
      begin
        Swap(i, j); j := j - 1
      end;
  k := i - 1
end;
```

Na kraju sledi procedura koja sortira segment `a[p .. q]` niza `a` tako što ga pozivom procedure `Split` preuredi, a potom rekurzivno segmente `a[p .. k-1]` i `a[k+1 .. q]`. Procedura se poziva sa `QSort(1, n)`.

```
procedure QSort(i, j : integer);
var
  k : integer;
begin
  if j - i = 1 then
    begin
      if a[i] > a[j] then Swap(i, j)
    end
  else if j - i > 1 then
    begin
      Split(i, j, k);
      QSort(i, k - 1);
      QSort(k + 1, j)
    end
end;
```

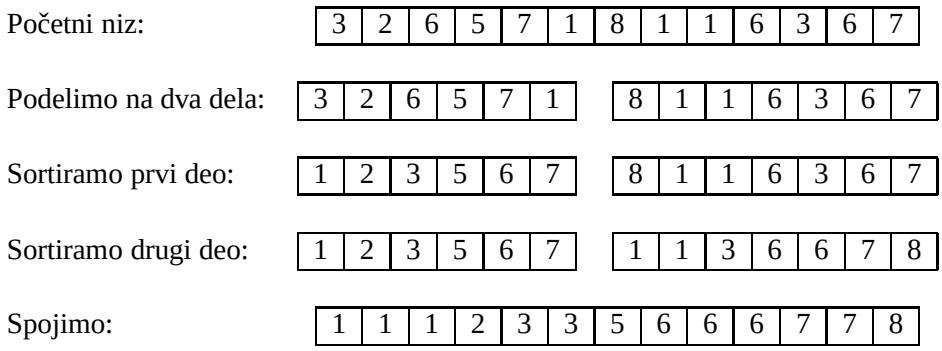
Pažljivom analizom algoritma se primećuje da se drugi rekurzivni poziv `QSort` procedure može eliminisati, i tako dobijamo još efikasniju verziju algoritma:

```
procedure QSort2(i, j : integer);
var
  k : integer;
begin
  while j - i > 1 do begin
    Split(i, j, k);
    QSort(i, k - 1);
    i := k + 1
  end;
  if j - i = 1 then
    begin
      if a[i] > a[j] then Swap(i, j)
    end
end;
```

7.3.2 Mergesort

Pokazaćemo sada drugi superiorni algoritam za sortiranje koji se zove *mergesort* (engl. sortiranje sa spajanjem) i koji predstavlja još jedan dobar primer strategije podeli pa vladaj. U ovom odeljku ćemo pokazati kako se mergesort može upotrebiti da se efikasno sortiraju liste.

Mergesort je takođe rekurzivan sort, a bazira se na veoma jednostavnoj ideji. Niz koga treba sortirati podelimo na dva podniza iste ili približno iste veličine, oba podniza sortiramo rekurzivnim pozivom algoritma, a onda ta dva sortirana dela spojimo (možda je slikovitije reći, slijemo) u jedan. Evo primera koji ilustruje osnovnu ideju algoritma:



Procedura koja implementira mergesort je veoma jednostavna. Ona sortira listu `p` tako što pozivom procedure `Split` podeli njene elemente da dva dela `a` i `b` približno jednakih dužina, potom rekurzivnim pozivom sortira listu `a`, pa listu `b`, i na kraju pozivom procedure `Join` "umeša" sortirane liste `a` i `b` u novu listu.

```
procedure MergeSort(var p : IntList);
var
  a, b : IntList;
begin
  if p <> nil then
    if p↑.next <> nil then begin
      Split(p, a, b);
      MergeSort(a);
      MergeSort(b);
      Join(a, b, p)
    end
end;
```

Procedura `Split` podeli elemente liste `p` da dva dela približno jednakih dužina. Ona radi tako što prvi element liste `p` prebaci na početak lista `a`, sledeći element

liste p prebaci na početak liste b, sledeći na početak liste a, i tako dok se lista p ne isprazni.

```
procedure Split(var p, a, b : IntList);
var
  q : IntList;
  flip : Boolean;
begin
  a := nil; b := nil;
  flip := false;
  while p <> nil do begin
    q := p;
    p := p↑.next;
    flip := not flip;
    if flip then begin
      q↑.next := a;
      a := q
    end
    else begin
      q↑.next := b;
      b := q
    end
  end
end;
```

Procedura Join spaja sortirane liste a i b u novu, takođe sortiranu listu. Za inicijalizaciju liste p procedura koristi činjenicu da ni a ni b nisu prazne liste (što obezbedi procedura MergeSort). Pokazivač p pokazuje na početak nove liste, dok q pokazuje na poslednji element nove liste. Tako se lakše realizuje dodavanje elementa na kraj liste na koju pokazuje p.

```
procedure Join(var a, b, p : IntList);
var
  q : IntList;
begin
  if a↑.n < b↑.n then begin
    p := a;
    q := a;
    a := a↑.next;
    q↑.next := nil
  end
  else begin
    p := b;
    q := b;
    b := b↑.next;
```

```
    q↑.next := nil
  end;

  while (a <> nil) and (b <> nil) do
    if a↑.n < b↑.n then begin
      q↑.next := a;
      q := a;
      a := a↑.next;
      q↑.next := nil
    end
    else begin
      q↑.next := b;
      q := b;
      b := b↑.next;
      q↑.next := nil
    end;

  if a <> nil then
    q↑.next := a
  else
    q↑.next := b
end;
```

7.3.3 Traženje *k*-tog po veličini elementa u nizu

Kada znamo da sortiramo nizove, problem nalaženja elementa niza koji je *k*-ti po veličini se lako rešava – sortiramo niz, pa iz sortiranog niza pročitamo *k*-ti element:

```
  const
    MaxN = 1000;
  type
    Niz = array [1 .. MaxN] of integer;

  procedure Sort(var a : Niz; n : integer);
  {sortira niz a}
  ...

  function FindKth(a : Niz; n, k : integer) : integer;
  {pretpostavljamo da je 1 <= k <= n}
  begin
    Sort(a, n);
    FindKth := a[k]
  end;
```

Interesantno je da se k-ti element niza `a` može naći i brže, bez sortiranja niza, algoritmom koji koristi ideju Quicksorta.

(1) Procedurom `Split(a, 1, n, s)` podelimo niz `a` na dva dela kao u Quicksort algoritmu;

(2) ako je $s = k$, k-ti element po veličini je tačno `a[k]`;

(3) ako je $s > k$, k-ti po veličini element se nalazi u segmentu `a[1...s-1]`, pa nastavimo da tražimo, i to k-ti element u segmentu `a[1...s-1]`;

(4) ako je $s < k$, k-ti po veličini element se nalazi u segmentu `a[s+1...n]`, pa nastavimo da tražimo, ali ovaj put $(k-s)$-ti element u segmentu `a[s+1...n]`.

Evo i implementacije. Procedura `Split(a, p, q, s)` deli segment `a[p...q]` niza `a` na dva dela kao u algoritmu za Quicksort, a `s` je pozicija pivota nakon preuređivanja niza. Procedure `min` i `max` vraćaju manji, odnosno, veći od dva broja.

```
function FindKth(a : Niz; n, k : integer) : integer;
var
  p, q, s : integer;
  found : Boolean;
begin
  p := 1;
  q := n;
  found := false;
  while not found and (q - p > 1) do begin
    Split(a, p, q, s);
    if (p-1) + k = s then
      found := true
    else if (p-1) + k < s then
      q := s - 1
    else begin
      p := s + 1;
      k := k - s
    end
  end;

  if found then
    FindKth := a[s]
  else if k = 1 then
    FindKth := min(a[p], a[q])
  else
    FindKth := max(a[p], a[q])
end;
```

Zadaci.

7.26. Napisati Pascal program koji sortira niz brojeva Quicksort algoritmom, ali u nerastućem redosledu, od najvećeg ka najmanjem.

7.27. Praksa pokazuje da se kratki nizovi efikasnije sortiraju primenom nekih od inferiornih algoritama, dok Quicksort može da pokaže svoju premoć samo na relativno dugačkim nizovima. Napisati Pascal program koji sortira nizove koristeći hibridnu tehniku: segmenti čija dužina je veća od 100 se sortiraju Quicksort algoritmom, dok se na segmente sa ne više od 100 elemenata primenjuje Insertion sort.

7.28. Napisati proceduru

```
procedure QSort(var p : IntList);
```

koja sortira listu metodom *quick sort*. Tokom procesa sortiranja nije dozvoljeno praviti nove dinamičke promenljive (nove kućice na heapu), već se sortiranje obavlja prevezivanjem pokazivača između već oformljenih kućica!

7.29. Napisati proceduru

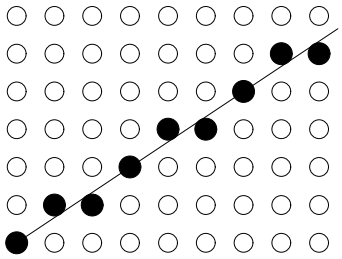
```
procedure MergeSort2(var p : IntList);
```

koja sortira listu metodom *mergesort*, ali od najvećeg ka najmanjem elementu liste. Tokom procesa sortiranja nije dozvoljeno praviti nove dinamičke promenljive (nove kućice na heapu), već se sortiranje obavlja prevezivanjem pokazivača između već oformljenih kućica!

Glava 8

Elementi računarske grafike

Računarska grafika je granična oblast računarstva i geometrije koja se bavi vernim prikazivanjem geometrijskih objekata na ekranu. Na slici ispod je pokazan najbolji mogući način da se u konačnom svetu kakav je ekran računara predstavi deo prave $y = \frac{2}{3}x$.



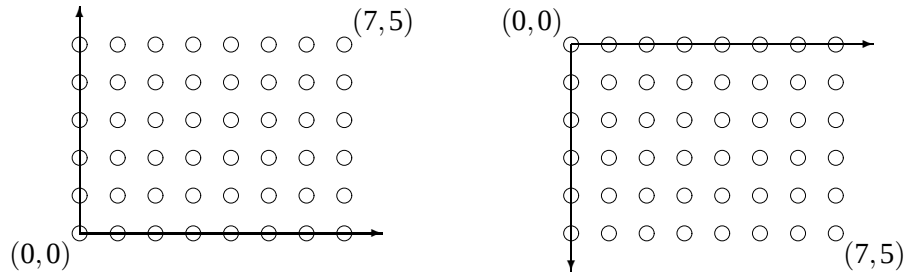
Računarska geometrija je deo matematike koji se bavi algoritamskim rešavanjem nekih geometrijskih problema. Na primer, sledeći problem pripada domenu računarske geometrije: za dati konačan skup tačaka S odrediti podskup C skupa S sa što manjim brojem elemenata takav da se sve tačke skupa S nalaze u unutrašnjosti ili na rubu poligona čija temena su tačke iz C.

Iako su ove dve discipline često prepletene, lako ih je razlikovati: algoritmi računarske grafike crtaju, dok algoritmi računarske geometrije konstruišu nove objekte i daju odgovore na pitanja.

Ekran računara je pravougaona matrica tačaka[1]. Dimenzije ove matrice određuju *rezoluciju* ekrana, odnosno, njegovu moć razlučivanja. Na primer, 800×600

[1]U žargonu se umesto "tačka" kaže "piksel"; piksel, od engleskog *pixel*, je veštački stvorena reč. Pre njene pojave, zvanično ime jedne tačke ekrana je bilo *picture element*. Inženjeri, koji vole skraćenice, reč *picture* standardno skraćuju na *pic*, reč *element* na *el*, što sa malo saksonskog genitiva od fraze *picture element* napravi *pic's el*. Uz brisanje razmaka dobijamo *picsel*, a nakon pogrešnog spelovanja, *pixel*.

ili 1280×1024. Svaki piksel je određen svojim koordinatama, parom nenegativnih celih brojeva. Pri tome je koordinatni početak u jednom temenu odgovarajuće matrice, dok su koordinatne ose postavljene ili uobičajeno ili izvrnuto:



uobičajeno izvrnuto

8.1 Osnovne grafičke rutine

Svaki grafički sistem je zasnovan na nekoliko osnovnih operacija kao što su inicijalizacija grafičkog režima, izlazak iz grafičkog režima, osvetljavanje jednog piksela i slično. Ove operacije zavise od platforme (platforma = hardver + operativni sistem + programski jezik) i najčešće se implementiraju u mašinskom jeziku. Zato osnovne grafičke rutine nisu propisane standardom, već je njihov dizajn ostavljen implementatorima konkretnog sistema.

☞ *U ovoj glavi ćemo koristiti FreePascal verziju 2.4. Napominjemo još jednom da druge implementacije Pascala mogu pratiti sasvim drugačiju filozofiju.*

FreePascal verzija 2.4 poznaje obilje grafičkih rutina koje se nalaze u modulu `Graph`, a od kojih ćemo mi pomenuti svega nekoliko. Da bi grafičke rutine iz ovog modula mogle da se koriste, moramo prvo Pascal prevodiocu reći da da "uveze" modul `Graph` upotrebom deklaracije `uses` koja se navodi odmah nakon deklaracije `program`:

```
program PrimerRadaSaGrafikom;
uses Graph;
{ sada mozemo da koristimo graficke rutine }
...
```

Evo sada i spiska osnovnih grafičkih rutina koje ćemo najčešće koristiti. Za inicijalizaciju grafičkog režima koristićemo sledeće rutine:

`procedure DetectGraph(var driver, mode: integer);`
detektuje grafičke parametre sistema i vrati ih kroz celobrojne promenljive

`driver` i `mode`; ove vrednosti koristimo kao ulazne parametre procedure `InitGraph`.

`procedure InitGraph(var driver, mode: integer; path: string);`
: inicijalizuje grafički režim rada sa parametrima `driver` i `mode` koje smo dobili pozivom procedure `DetectGraph`; argument `path` se koristi kada želimo da koristimo napredne načine da inicijalizujemo grafički režim rada, u kom slučaju moramo da obezbedimo svoj drajver; u ovom kursu nećemo želeti da koristimo napredne načine inicijalizacije, tako da ćemo na mestu ovog argumenta uvek prosleđivati prazan string.

`function GraphResult: integer;`
: vraća broj koji govori o tome da li je grafička naredba koja prethodi pozivu funkcije `GraphResult` uspešno izvršena; na primer, do greške može doći ako pokušamo da inicijalizujemo grafički režim parametrima koje hardver ne podržava.

`const GrOk = 0;`
: je konstanta koja se koristi u saradnji sa funkcijom `GraphResult` i označava da je sve u redu, odnosno, da je grafička naredba koja prethodi pozivu funkcije `GraphResult` uspešno izvršena.

`procedure CloseGraph;`
: napušta grafički režim.

Tipičan program u ovom kursu koji se bavi crtanjem će imati sledeću strukturu:

```
program TipicanGrafProgram;
uses Graph;
var
  gd, gm : integer;
  ...
begin
  DetectGraph(gd, gm);
  InitGraph(gd, gm, '');
  if GraphResult <> GrOk then
    writeln('Graph Init error')
  else begin
    { neko crtanje }
    ...
    CloseGraph;
  end
end.
```

Za crtanje osnovnih grafčkih objekata koristimo sledeće naredbe:

`procedure SetColor(color: integer);`
postavlja boju kojom se crta (boja mastila); promenljiva `color` treba da ima vrednost iz skupa $\{0,1,\ldots,255\}$.

`function GetColor: integer;`
vraća boju kojom se crta (boja mastila).

`procedure SetBkColor(color: integer);`
postavlja boju pozadine po kojoj se crta (boja papira); promenljiva `color` treba da ima vrednost iz skupa $\{0,1,\ldots,255\}$.

`function GetBkColor: integer;`
vraća boju pozadine po kojoj se crta (boja papira).

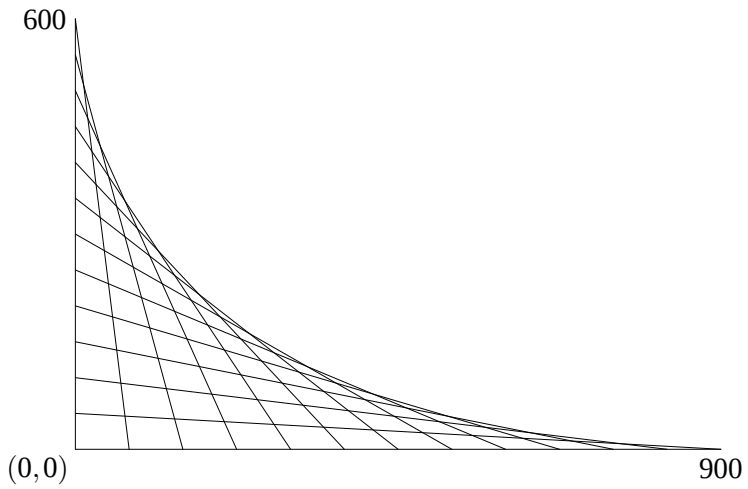
`procedure PutPixel(x, y, color: integer);`
crta piksel na koordinatama (x,y) bojom `color`; promenljiva `color` treba da ima vrednost iz skupa $\{0,1,\ldots,255\}$.

`procedure Line(x1, y1, x2, y2: integer);`
crta duž čije krajnje tačke su (x_1,y_1) i (x_2,y_2) bojom koja je postavljena naredbom `SetColor`.

`procedure Circle(x, y, r: integer);`
crta krug sa centrom u tački (x,y) i poluprečnika `r` bojom koja je postavljena naredbom `SetColor`.

Primer. Napisati Pascal program koji crta apstrakciju na Sl. 8.1.

```
program Apstrakcija;
uses graph;
var
  gd, gm, i, p, q : integer;
begin
  DetectGraph(gd, gm);
  InitGraph(gd, gm, '');
  if GraphResult <> GrOk then
    writeln('Graph Init error')
  else begin
    { prvo crtamo vertikalnu i horizontalnu liniju }
    Line(0, 0, 900, 0);
    Line(0, 0, 0, 600);
```



Slika 8.1: Apstrakcija

```
    { sada crtamo kose linije }
    p := 600;
    q := 75;
    for i := 1 to 12 do begin
      Line(0, p, q, 0);
      p := p - 50;
      q := q + 75
    end;
    CloseGraph
  end
end.
```

Zadaci.

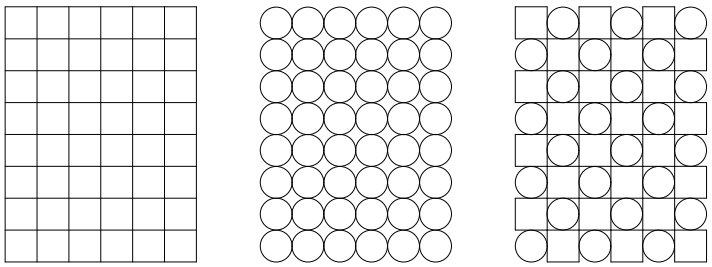
8.1. Napisati Pascal proceduru koja utvrđuje da li data tačka pripada simetrali date duži (duž je data svojim krajnjim tačkama).

8.2. Napisati Pascal proceduru `Rectangle(x0, y0, x1, y1: integer)` koja crta pravougaonik čije donje levo teme ima koordinate (x_0, y_0), a gornje desno koordinate (x_1, y_1).

8.3. Napisati Pascal program koji od korisnika učitava prirodne brojeve m i n i potom crta

(a) "rešetku kvadrata" formata $m \times n$ kao na Sl. 8.2 (a). (Na slici je prikazana rešetka formata 8×6.)

(b) "rešetku krugova" formata $m \times n$ kao na Sl. 8.2 (b). (Na slici je prika-



(a) (b) (c)

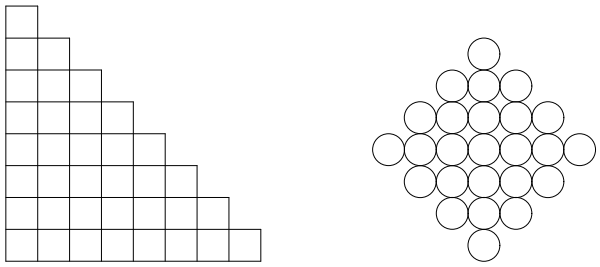
Slika 8.2: Tri rešetke

zana rešetka formata 8×6.)

(c) "rešetku kvadrata i krugova" formata $m \times n$ kao na Sl. 8.2 (c). (Na slici je prikazana rešetka formata 8×6.)

8.4. Napisati Pascal program koji od korisnika učitava prirodan broj n i potom crta "stepenice" sa n stepenika kao na Sl. 8.3 (a). (Na slici su prikazane stepenice sa 8 stepenika.)

8.5. Napisati Pascal program koji od korisnika učitava neparan prirodan broj n i potom crta "dijamant" reda n kao na Sl. 8.3 (b). (Na slici je prikazan dijamant reda 7.)



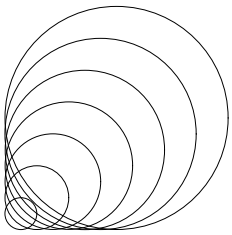
(a) (b)

Slika 8.3: Stepenice i dijamant

8.6. Napisati Pascal proceduru `DrawPolygon(P : Polygon)` koja crta poligon. Poligon je opisan sledećim tipom:

```
const
  MaxN = 100;
type
  Point = record
    x, y : integer
  end;
  Polygon = record
    N : integer; { broj temena }
    A : array [1 .. maxN] of Point { spisak temena }
  end;
```

8.7. Napisati Pascal program koji od korisnika učitava prirodan broj n i potom crta n krugova kao na slici pored, pri čemu i-ti krug ima centar u tački $(10i, 10i)$ i poluprečnik $10i$.



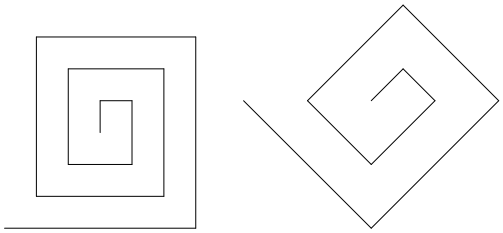
8.8. Napisati Pascal proceduru `SqSpiral(cx, cy, n, k : integer)` koja crta prvih `n` zavoja pravougaone spirale. Početak spirale je tačka `(cx, cy)`, a `k` je dužina jednog koraka.

Pravougaona spirala sa korakom k se crta ovako. Krenemo od početne tačke i idemo k piksela na sever, k piksela na istok, $2k$ piksela na jug i $2k$ piksela na zapad. To je prvi zavoj. Potom $3k$ piksela na sever, $3k$ piksela na istok, $4k$ piksela na jug i $4k$ piksela na zapad. To je drugi zavoj. Za treći zavoj idemo $5k$ piksela na sever, $5k$ piksela na istok, $6k$ piksela na jug i $6k$ piksela na zapad. I tako dalje, Sl. 8.4 (a).

8.9. Napisati Pascal proceduru `SkewSqSpiral(cx, cy, n, k : integer)` koja crta prvih `n` zavoja iskošene pravougaone spirale. Kao i u prethodnom zadatku, početak spirale je tačka `(cx, cy)`, a `k` je dužina jednog koraka.

Iskošena pravougaona spirala sa korakom k se crta ovako. Krenemo od početne tačke i idemo k piksela na severo-istok, k piksela na jugo-istok, $2k$ piksela na jugo-zapad, $2k$ piksela na severo-zapad, i tako dalje, Sl. 8.4 (b).

8.10. Napisati Pascal program koji od korisnika učitava n tačaka, a potom crta tih n tačaka, kao i najmanji pravougaonik čije dve strane su paralelne koordinatnim osama, a koji u svojoj unutrašnjosti sadrži date tačke.



Slika 8.4: (a) Tri zavoja pravougaone spirale; (b) Dva zavoja iskošene pravougaone spirale.

8.2 Prava

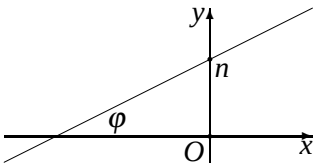
Kao što je tačka u ravni određena svojim koordinatama, tako je prava određena svojom *jednačinom*. Pravoj pripadaju tačno one tačke čije koordinate zadovoljavaju njenu jednačinu. Ako je prava normalna na x-osu i seče je u tački c, jednačina prave ima oblik

$$x = c.$$

Ako prava nije nromalna na x-osu, njena jednačina ima oblik

$$y = kx + n.$$

Broj n se zove *otsečak na y-osi*, zato što prava prolazi kroz tačku $(0,n)$, tj. preseca y-osu u tački n. Broj k se zove *nagib* ili *koeficijent pravca* prave. On je jednak tangensu ugla koga prava zaklapa sa pozitivnim smerom x-ose: $k = \operatorname{tg}\varphi$.



Podsetimo se da se jednačina prave kroz tačke $A(p_1,q_1)$ i $B(p_2,q_2)$ može lako dobiti na sledeći način: ako je $p_1 = p_2$, jednačina prave AB je data sa

$$x = p_1;$$

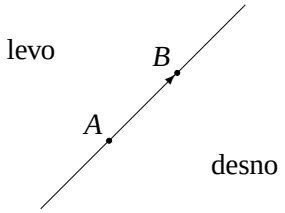
ako je $p_1 \neq p_2$, jednačina prave AB je data sa

$$y = q_1 + \frac{q_2 - q_1}{p_2 - p_1}(x - p1).$$

Primer. Odredimo jednačinu prave $y = kx + n$ koja sadrži tačke $(2,1)$ i $(4,4)$. Kako tačka $(2,1)$ pripada pravoj, ona mora zadovoljavati odgovarajuću jednačinu, pa je $1 = 2k + n$. Isto važi i za drugu tačku: $4 = 4k + n$. Tako smo dobili sistem dve

jednačine sa nepoznatim k i n, čijim rešavanjem nalazimo $k = \frac{3}{2}$, $n = -2$. Tražena jednačina je $y = \frac{3}{2}x - 2$.

Ako pravu l određenu tačkama $A(p_1, q_1)$ i $B(p_2, q_2)$ orijentišemo tako da bude istog smera kao vektor $\overrightarrow{AB}$, onda možemo govoriti o "levoj" i "desnoj" poluravni s obzirom na pravu l. Formulu za jednačinu prave kroz tačke A i B možemo dovesti u sledeći oblik:



$$(q_2 - q_1)x - (p_2 - p_1)y + q_1p_2 - p_1q_2 = 0.$$

Za tačku $M(x_0, y_0)$ stavimo

$$\lambda_M = (q_2 - q_1)x_0 - (p_2 - p_1)y_0 + q_1p_2 - p_1q_2.$$

Jasno, $\lambda_M = 0$ ukoliko tačka M leži na pravoj l. Ako tačka M ne leži na pravoj l ova vrednost nije nula. Interesantno je da *znak* broja λ_M određuje da li se tačka M nalazi u levoj ili desnoj poluravni s obzirom na pravu l:

- ako je $\lambda_M = 0$, tačka M leži na pravoj l;
- ako je $\lambda_M > 0$, tačka M leži u desnoj poluravni;
- ako je $\lambda_M < 0$, tačka M leži u levoj poluravni.

Na osnovu toga možemo lako napraviti funkciju koja za date tri tačke $A(p_1, q_1)$, $B(p_2, q_2)$, $M(x_0, y_0)$ vraća 0, 1 ili -1 u zavisnosti od toga da li tačka M leži na pravoj AB ili se nalazi u desnoj, odnosno, levoj poluravni s obzirom na orijentisanu pravu $\overrightarrow{AB}$:

```
function LeftRight(p1, q1, p2, q2, x0, y0 : integer) : integer;
var
  lambda : longint;
begin
  lambda := (q2 - q1)*x0 - (p2 - p1)*y0 + q1*p2 - p1*q2;
  if lambda > 0 then
    LeftRight := 1
  else if lambda < 0 then
    LeftRight := -1
  else
    LeftRight := 0
end;
```

8.3 Bresenhamov inkrementalni algoritam za crtanje duži

Duž kao geometrijski objekt nema debljinu i sastoji se od beskonačno mnogo tačaka. S druge strane, raster ekrana se sastoji od konačno mnogo piksela koji su dvodimenzionalni objekti, recimo kružići. Algoritam za crtanje duži određuje kolekciju piksela koji leže na ili u blizini date duži.

U ovom odeljku ćemo predstaviti jedno čuveno rešenje ovog problema: Bresenhamov inkrementalni algoritam za crtanje duži[2]. Reč "inkrementalni" znači da se u svakom koraku algoritma na osnovu koordinata piksela koji je upravo osvetljen određuju koordinate narednog piksela koga treba osvetliti. Bresenhamov algoritam je značajan bar iz dva razloga:

- ne koristi realnu aritmetiku čime se dobija na brzini, i
- daje optimalno rešenje u sledećem smislu: ne postoji kolekcija piksela koja ima manje srednje kvadratno odstupanje od idealne duži.

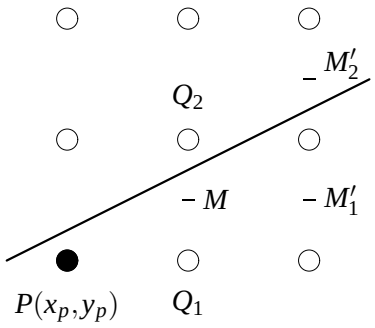
Neka su $A(x_0,y_0)$ i $B(x_1,y_1)$ krajnje tačke duži koju treba nacrtati. Pretpostavićemo da je $x_0 < x_1$ i da koeficijent pravca prave AB pripada intervalu $[0,1]$. Ako uvedemo sledeće oznake: $\Delta_y = y_1 - y_0$ i $\Delta_x = x_1 - x_0$, jednačina prave AB ima oblik $y = kx + n$ za $k = \Delta_y/\Delta_x$ i neko n. Množenjem sa Δ_x i sređivanjem dobijamo:

$$\Delta_y x - \Delta_x y + n\Delta_x = 0.$$

Iz tehničkih razloga (jedne nestašne dvojčice koja se kasnije pojavi), pomnožićemo ovu jednačinu sa 2. Tako, uz oznaku $c = 2n\Delta_x$ dobijamo

$$2\Delta_y x - 2\Delta_x y + c = 0.$$

Pretpostavimo da smo u prethodnom koraku osvetlili piksel $P(x_p,y_p)$. U sledećem koraku ćemo osvetliti ili piksel $Q_1(x_p+1,y_p)$ ili piksel $Q_2(x_p+1,y_p+1)$ prema sledećem kriterijumu. Posmatrajmo tačku $M(x_p+1,y_p+\frac{1}{2})$ koja je središte duži $[Q_1Q_2]$. Ako se ona nalazi ispod prave AB, osvetlićemo piksel Q_1; u suprotnom ćemo osvetliti piksel Q_2. Da li se tačka M nalazi ispod ili iznad prave AB utvrđujemo na način koga smo upravo opisali:



[2] Bresenham, J. E., *Algorithm for Computer Control of a Digital Plotter*, IBM Systems Journal, 4(1) 1965, str. 25–30

neka je $\lambda_M = 2\Delta_y(x_p+1) - 2\Delta_x(y_p+\frac{1}{2}) + c$; ako je $\lambda_M > 0$, tačka M se nalazi ispod prave AB, u suprotnom se nalazi na pravoj AB ili iznad nje.

Prilikom crtanja duži krećemo od tačke sa x-koordinatom x_0 i ponavljamo postupak dok ne dođemo do tačke sa x-koordinatom x_1. Osnovni algoritam, zato, izgleda ovako:

```
x := x0;
y := y0;
PutPixel(x, y, c);
while  x < x1  do begin
    if  λ_{M(x+1,y+1/2)} ⩽ 0  then
        x := x+1
    else begin
        x := x+1;
        y := y+1
    end;
    PutPixel(x, y, c);
end
```

Izraz λ_M takođe može da se računa inkrementalno, čime se algoritam značajno ubrzava. Neka je $\lambda_M = 2\Delta_y(x_p+1) - 2\Delta_x(y_p+\frac{1}{2}) + c$. Ako smo odabrali piksel Q_1, sledeća tačka na osnovu čijeg položaja donosimo odluku je $M_1'(x_p+2, y_p+\frac{1}{2})$, i trebaće nam vrednost izraza $\lambda_{M_1'} = 2\Delta_y(x_p+2) - 2\Delta_x(y_p+\frac{1}{2}) + c$. Primetimo da je $\lambda_{M_1'} = \lambda_M + 2\Delta_y$.

Ako smo odabrali piksel Q_2, sledeća tačka na osnovu čijeg položaja donosimo odluku je $M_2'(x_p+2, y_p+\frac{3}{2})$, i trebaće nam vrednost izraza $\lambda_{M_2'} = 2\Delta_y(x_p+2) - 2\Delta_x(y_p+\frac{3}{2}) + c$. Primetimo da je $\lambda_{M_2'} = \lambda_M + 2(\Delta_y - \Delta_x)$. Dakle, nema potrebe da se izrazi $\lambda_{M_1'}$ i $\lambda_{M_2'}$ računaju od početka; u prvom slučaju je dovoljno na λ_M dodati $2\Delta_y$, a u drugom slučaju $2(\Delta_y - \Delta_x)$.

Potražimo sada početnu vrednost za λ_M koja se dobija za tačku M_0 čije koordinate su $(x_0+1, y_0+\frac{1}{2})$:

$$\begin{aligned}\lambda_{M_0} &= 2\Delta_y(x_0+1) - 2\Delta_x(y_0+\tfrac{1}{2}) + c \\ &= (2\Delta_y x_0 - 2\Delta_x y_0 + c) + 2\Delta_y - \Delta_x \\ &= \lambda_A + 2\Delta_y - \Delta_x.\end{aligned}$$

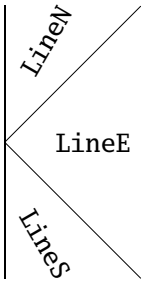
Zato što tačka A pripada pravoj AB imamo $\lambda_A = 0$, i time dobijamo početnu vrednost

$$\lambda_{M_0} = 2\Delta_y - \Delta_x.$$

Konačna verzija algoritma izgleda ovako:

```
const
  MyColor = 30;

procedure Line0(x0, y0, x1, y1 : integer);
var
  lambda, x, y, dx, dy, incr1, incr2 : integer;
begin
  dx := x1 - x0;
  dy := y1 - y0;
  lambda := 2 * dy - dx;
  incr1 := 2 * dy;
  incr2 := 2 * (dy - dx);

  x := x0; y := y0;
  PutPixel(x, y, MyColor);
  while x < x1 do begin
    if lambda <= 0 then begin
      x := x + 1;
      lambda := lambda + incr1
    end
    else begin
      x := x + 1;
      y := y + 1;
      lambda := lambda + incr2
    end;
    PutPixel(x, y, MyColor);
  end
end;
```

Napominjemo još jednom da procedura `Line0` radi korektno samo za $x_0 < x_1$ i samo za one duži čiji koeficijent pravca pripada intervalu $[0,1]$.



Kompletan algoritam za crtanje linije se sastoji od četiri procedure. Tri vode računa o posebnim slučajevima: `LineE` (E – istok) koja crta duži čiji koeficijent pravca pripada intervalu $[-1,1]$, `LineN` (N – sever) koja crta duži čiji koeficijent pravca je veći od 1 i `LineS` (S – jug) koja crta duži čiji koeficijent pravca je manji od -1, dok četvrta, glavna procedura, `DrawLine` utvrđuje o kom slučaju se radi i poziva jednu od ove tri procedure. Procedura `LineE` radi korektno za $x_0 < x_1$ i za duži čiji koeficijent pravca pripada intervalu $[-1,1]$. Ona se dobija jednostavnom modifikacijom procedure `Line0`: promenljiva `stepY` određuje da li će duž biti iscrtavana ka gore (`stepY = 1`) ili ka dole (`stepY = -1`).

```
procedure LineE(x0, y0, x1, y1 : integer);
var
  lambda, x, y, dx, dy, incr1, incr2, stepY : integer;
begin
  if y1 < y0 then stepY := -1
  else stepY := 1;
  dx := x1 - x0; dy := abs(y1 - y0);
  lambda := 2 * dy - dx;
  incr1 := 2 * dy; incr2 := 2 * (dy - dx);

  x := x0; y := y0;
  PutPixel(x, y, MyColor);
  while x < x1 do begin
    x := x + 1;
    if lambda <= 0 then
      lambda := lambda + incr1
    else begin
      lambda := lambda + incr2;
      y := y + stepY
    end;
    PutPixel(x, y, MyColor)
  end
end;
```

Procedura LineN radi korektno za $y_0 < y_1$ i za duži čiji koeficijent pravca je veći od 1. I ona se dobija jednostavnom modifikacijom procedure Line0: samo zamenimo y-koordinatu sa x-koordinatom.

```
procedure LineN(x0, y0, x1, y1 : integer);
var
  lambda, x, y, dx, dy, incr1, incr2 : integer;
begin
  dx := x1 - x0; dy := y1 - y0;
  lambda := 2 * dx - dy;
  incr1 := 2 * dx; incr2 := 2 * (dx - dy);

  x := x0; y := y0;
  PutPixel(x, y, MyColor);
  while y < y1 do begin
    y := y + 1;
    if lambda <= 0 then
      lambda := lambda + incr1
    else begin
      lambda := lambda + incr2;
      x := x + 1
```

```
    end;
    PutPixel(x, y, MyColor)
  end
end;
```

Procedura LineS radi korektno za $y_0 > y_1$ i za duži čiji koeficijent pravca je manji od -1. Ona se dobija jednostavnom modifikacijom procedure LineN.

```
procedure LineS(x0, y0, x1, y1 : integer);
var
  lambda, x, y, dx, dy, incr1, incr2 : integer;
begin
  dx := x1 - x0; dy := abs(y1 - y0);
  lambda := 2 * dx - dy;
  incr1 := 2 * dx; incr2 := 2 * (dx - dy);

  x := x0; y := y0;
  PutPixel(x, y, MyColor);
  while y > y1 do begin
    y := y - 1;
    if lambda <= 0 then
      lambda := lambda + incr1
    else begin
      lambda := lambda + incr2;
      x := x + 1
    end;
    PutPixel(x, y, MyColor)
  end
end;
```

Glavna procedura:

```
procedure DrawLine(x0, y0, x1, y1 : integer);
var
  t : integer;
begin
  if x0 > x1 then begin { zameni tacke }
    t := x0; x0 := x1; x1 := t;
    t := y0; y0 := y1; y1 := t
  end;
  if abs(y1 - y0) <= x1 - x0 then LineE(x0, y0, x1, y1)
  else if y1 > y0 then LineN(x0, y0, x1, y1)
  else LineS(x0, y0, x1, y1)
end;
```

Zadaci.

8.11. Opisati prave čiji koeficijent pravca je: (a) -1; (b) 1; (c) 0.

8.12. Tačke $O(0,0)$, $A(6,8)$, $B(6,4)$, $C(3,0)$, $D(2,2)$, $E(-2,-8)$, $F(2,-2)$, $G(0,5)$, $H(4,2)$, razvrstati u dve grupe prema tome sa koje strane prave $y = \frac{3}{2}x - 3$ se nalaze.

8.13. Napisati Pascal proceduru koja proverava da li data tačka pripada unutrašnjosti datog trougla.

8.14. Napisati Pascal proceduru `ThickLine(x0, y0, x1, y1, w: integer)` koja crta duž sa krajnjim tačkama `(x0, y0)` i `(x1, y1)`, ali tako da debljina duži bude `w` piksela.

Ako je nagib duži između -1 i 1, zadebljanje crtati kao vertikalni niz od `w` piksela, od kojih je polovina iznad, a polovina ispod osnovne linije. Ako je nagib duži veći od 1 ili manji od -1, zadebljanje crtati kao horizontalni niz od `w` piksela, od kojih je polovina levo, a polovina desno od osnovne linije.

8.15. Napisati Pascal proceduru `StyledLine(x0, y0, x1, y1 : integer; st : LineStyle)` koja crta duž sa krajnjim tačkama `(x0, y0)` i `(x1, y1)`, pri čemu je stil (crta-crta, crta-tačka-crta i sl.) opisan promenljivom `st` tipa `LineStyle` koji je definisan ovako:

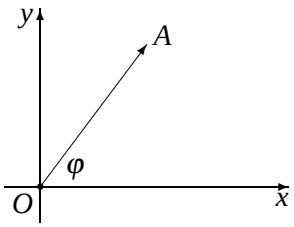
```
type
  LineStyle = array [0 .. 15] of Boolean;
```

Uvedite pomoćnu promenljivu `k` koja broji nacrtane piksele (odnosno, piksele koje bi procedura `Line0` nacrtala). Proceduru `PutPixel(x, y, c)` pozvati samo ako je `st[k mod 16] = true`.

8.16. Napisati Pascal proceduru `ThickStyledLine(x0, y0, x1, y1, w : integer; st : LineStyle)` koja crta podebljanu duž sa naznačenim stilom.

8.4 Polarne koordinate i grafika sa kornjačom

Polarne koordinate predstavljaju drugi način predstavljanja tačaka u ravni. Svaka tačka A ravni je jednoznačno određena svojim rastojanjem od koordinatnog početka i uglom koga sa pozitivnim smerom x-ose zaklapa poluprava OA. Ako je r rastojanje tačke A od koordinatnog početka, a φ ugao koga poluprava OA zaklapa sa pozitivnim smerom x-ose, onda za uređeni par (r, φ) kažemo da predstavlja *polarne koordinate* tačke A.

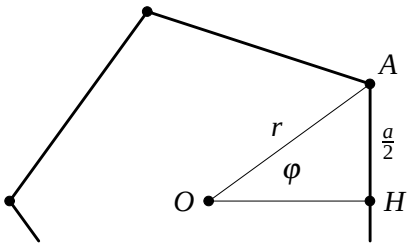


Preračun iz polarnih u "obične" koordinate (koje se zovu još i pravougaone koordinate) je jednostavan: ako tačka A ima pravougaone koordinate (x,y) onda je $x = r\cos\varphi$, $y = r\sin\varphi$. Preračun iz pravougaonih koordinata u polarne je nešto složeniji. Jasno je da je $r = \sqrt{x^2+y^2}$, dok kod izračunavanja ugla φ dolazi do komplikacija.

- Ako je $x = y = 0$, onda se uzima da je $\varphi = 0$.
- Ako je $x = 0$ i $y \neq 0$, onda je $\varphi = \operatorname{sgn} y \cdot \frac{\pi}{2}$.
- Ako je $x > 0$, onda je $\varphi = \operatorname{arctg}\frac{y}{x}$.
- Ako je $x < 0$ i $y = 0$, onda je $\varphi = -\pi$.
- Ako je $x < 0$ i $y \neq 0$, onda je $\varphi = \operatorname{sgn} y \cdot \pi + \operatorname{arctg}\frac{y}{x}$.

Primer. Nacrtati pravilan n-ugao ako je data dužina a njegove strane.

Rešenje. Da bismo nacrtali pravilan n-ugao stranice a trebaće nam polarne koordinate. Pretpostavimo da je centar pravilnog n-ugla u koordinatnom početku O, neka je A jedno njegovo teme, i neka je H podnožje normale iz O na jednu od stranica n-ugla koja sadrži teme A, Sl. 8.5.



Slika 8.5: Pravilan n-ugao

Neka je $\varphi = \angle HOA = \dfrac{1}{2}\cdot\dfrac{2\pi}{n} = \dfrac{\pi}{n}$ i neka je $r = OA$. Kako je $AH = a/2$, lako se vidi da je

$$\frac{a/2}{r} = \sin\varphi = \sin\frac{\pi}{n}$$

odakle dobijamo

$$r = \frac{a}{2\sin\frac{\pi}{n}}.$$

Dakle, polarne koordinate temena pravilnog n-ugla su

$$(r,0), \quad \left(r, \frac{2\pi}{n}\right), \quad \left(r, 2\cdot\frac{2\pi}{n}\right), \ldots \left(r, (n-1)\cdot\frac{2\pi}{n}\right),$$

pa procedura koja crta pravilan n-ugao izgleda ovako:

```
procedure DrawRegPoly(a, n : integer);
var
  phi, r : real;
  x0, y0, x1, y1, i : integer;
begin
  r := a / (2 * sin(pi / n));
  x0 := round(r);
  y0 := 0;
  phi := 0;
  for i := 1 to n do begin
    phi := phi + 2 * pi / n;
    x1 := round(r * cos(phi));
    y1 := round(r * sin(phi));
    Line(500 + x0, 350 + y0, 500 + x1, 350 + y1);
    x0 := x1;
    y0 := y1
  end
end;
```

Grafika sa kornjačom (engl. *turtle graphics*) je razvijena na MIT-u radi približavanja programiranja deci. Zamišljamo da na ekranu imamo kornjaču koja nosi olovku. Olovka može biti podignuta ili spuštena. Ako je olovka spuštena, kornjača crta dok se kreće; ako je olovka podignuta, kornjača se pomera, ali ne crta. Stanje kornjače je određeno njenom pozicijom na ekranu, smerom u kome gleda i informacijom o položaju olovke:

```
var
  Turtle : record
    x, y    : integer; { polozaj kornjace           }
    theta   : real;    { smer u kome kornjaca gleda }
    penDown : Boolean  { polozaj olovke             }
  end;
```

Kornjača razume sledeće naredbe:

`TurtleSet(x, y : integer; phi : real)`
: Postavi kornjaču na koordinate `(x, y)`, postavi `theta` na `phi`, a `penDown` na `true`.

`TurtleMove(k : integer)`
: Kornjača se pomera za k jedinica u smeru u kome gleda i pri tome crta duž ako je olovka spuštena. Da bi se odredio novi položaj kornjače potrebno je malo trigonometrije. Neka se pre naredbe `TurtleMove` kornjača nalazila na poziciji (x_0, y_0) i neka je θ smer u kome kornjača gleda. Novi položaj kornjače (x_1, y_1) se računa ovako:

$$x_1 = x_0 + k\cos\theta,$$
$$y_1 = y_0 + k\sin\theta.$$

Ako je olovka spušetna, nacrta se duž sa krajnjim tačkama (x_0, y_0) i (x_1, y_1), i kornjača pređe na novu poziciju. U suprotnom, kornjača samo promeni položaj.

`TurtleTurn(phi : real)`
: Kornjača se okrene za ugao `phi` tako što se polje `theta` poveća za `phi`.

`TurtlePenDown`
: Kornjača spusti olovku (polje `penDown` se postavi na `true`).

`TurtlePenUp`
: Kornjača podigne olovku (polje `penDown` se postavi na `false`).

Evo sada i njihove implementacije:

```
procedure TurtleSet(x, y : integer; phi : real);
begin
  Turtle.x := x;
  Turtle.y := y;
  Turtle.theta := phi;
  Turtle.penDown := true
end;

procedure TurtleTurn(phi : real);
begin
  Turtle.theta := Turtle.theta + phi
end;
```

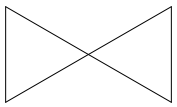
```
procedure TurtlePenDown;
begin
  Turtle.penDown := true
end;

procedure TurtlePenUp;
begin
  Turtle.penDown := false
end;

procedure TurtleMove(k : integer);
var
  x1, y1 : integer;
begin
  x1 := Turtle.x + round(k * cos(Turtle.theta));
  y1 := Turtle.y + round(k * sin(Turtle.theta));
  if Turtle.penDown then Line(Turtle.x, Turtle.y, x1, y1);
  Turtle.x := x1;
  Turtle.y := y1
end;
```

Primer. Nacrtati pravilan n-ugao ako je data dužina a njegove strane koristeći grafiku sa kornjačom.

Rešenje. Da bismo nacrtali pravilan n-ugao stranice a upotrebom grafike sa kornjačom treba prvo da se prisetimo da je zbir spoljašnjih uglova pravilnog n-ugla jednak 2π. Algoritam se sada svodi na to da kornjača n puta iscrta duž dužine a i okrene se za ugao od $2\pi/n$:

```
procedure TurtleRegPoly(a, n : integer);
var
  i : integer;
begin
  for i := 1 to n do begin
    TurtleMove(a);
    TurtleTurn(2 * pi / n)
  end
end;
```

Primer. Nacrtati leptir mašnu koristeći grafiku sa kornjačom. Leptir mašna se sastoji od dva jednakostranična trougla koji se dodiruju, kako je to pokazano na slici.

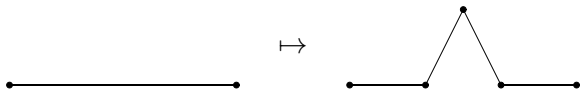


Rešenje.

```
procedure LeptirMasna(a : integer);
begin
  TurtleMove(a);
  TurtleTurn(2*pi/3);
  TurtleMove(2 * a);
  TurtleTurn(-2*pi/3);
  TurtleMove(a);
  TurtleTurn(-2*pi/3);
  TurtleMove(2 * a);
  TurtleTurn(2*pi/3)
end;
```

Primer. *Kochova kriva* reda n je poligonalna linija koja se dobija ovako:

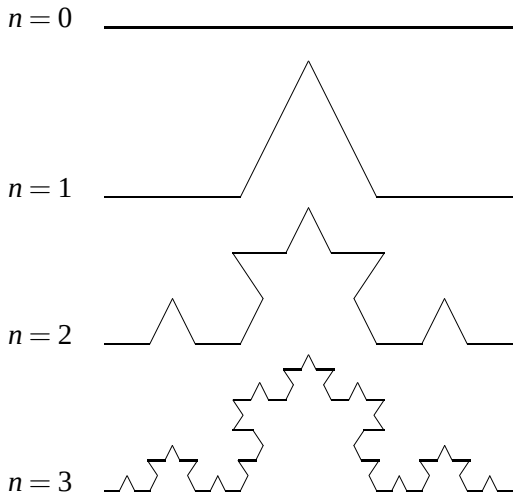
- Kochova kriva reda 0 je duž;
- Ako je konstruisana Kochova kriva reda $n-1$, Kochova kriva reda n se dobija tako što se nad svakom duži Kochove krive reda $n-1$ izvrši sledeća operacija:



  Dakle, nad srednjom trećinom duži konstruišemo jednakostranični trougao i izbacimo onu njegovu stranicu koja leži na duži.

Nekoliko Kochovih krivih je dato na Sl. 8.6. Napisati Pascal proceduru koja za dato n crta Kochovu krivu reda n koristeći grafiku sa kornjačom.
Rešenje.

```
procedure Koch(a, n : integer);
begin
  if n = 0 then
    TurtleMove(a)
  else begin
    Koch(a, n-1);
    TurtleTurn(pi/3);
    Koch(a, n-1);
    TurtleTurn(-2 * pi/3);
    Koch(a, n-1);
```



Slika 8.6: Kochova kriva

```
    TurtleTurn(pi/3);
    Koch(a, n-1)
  end
end;
```

Zadaci.

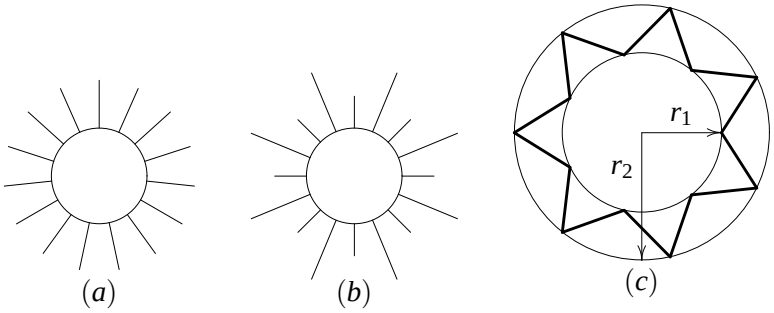
8.17. Odrediti polarne koordinate sledećih tačaka: $(3,0)$, $(1,1)$, $(0,6)$, $(-\frac{1}{2},-\frac{\sqrt{3}}{2})$, $(-\sqrt{2},-\sqrt{2})$.

8.18. Odrediti pravougaone koordinate tačaka čije polarne koordinate su: $(0,0)$, $(1,\frac{\pi}{4})$, $(3,\frac{2\pi}{3})$, $(1,-\pi)$, $(4,-\frac{3\pi}{4})$.

8.19. Odrediti koordinate temena pravilnog petougla čiji centar je u koordinatnom početku i čije jedno teme je na x-osi, ako se zna da je poluprečik opisanog kruga oko tog petougla jednak 3.

8.20. Napisati Pascal procedure `PolarToRect(r, phi : real; var x, y : real)` i `RectToPolar(x, y : real; var r, phi : real)` koje realizuju prelazak sa jednog sistema označavanja tačaka na drugi. (Funkcija arctg se u Pascalu označava sa `atan`.)

8.21. Napisati Pascal proceduru `Sunce(x, y, r, n, z : integer)` koja crta sunce kao na Sl. 8.7 (a) gde parametri imaju sledeće značenje: `(x, y)` su

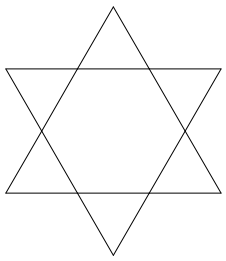


Slika 8.7: Dva sunca i zupčanik

koordinate centra sunca, `r` je poluprečnik tela sunca, `n` je broj zrakova ($n \geqslant 3$), a `z` je dužina zrakova.

8.22. Napisati Pascal proceduru `VeseloSunce(x, y, r, n, z1, z2 : integer)` koja crta sunce kao na Sl. 8.7 (b) gde parametri imaju sledeće značenje: `(x, y)` su koordinate centra sunca, `r` je poluprečnik tela sunca, `n` je broj zrakova (n je paran broj i $n \geqslant 6$), a `z1` i `z2` su dužine zrakova.

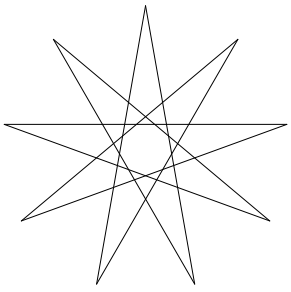
8.23. Napisati Pascal proceduru `Zupcanik(x, y, r1, r2, n : integer)` koja crta zupcanik kao na Sl. 8.7 (c) gde parametri imaju sledeće značenje: `(x, y)` su koordinate centra zupcanika, `r1` i `r2` su unutrašnji i spoljašnji poluprečnik zupčanika, a `n` je broj zubaca ($n \geqslant 5$).



8.24. Napisati Pascal program koji crta Davidovu zvezdu (videti sliku pored)

(a) koristeći grafiku sa kornjačom;

(b) koristeći samo naredbu `Line`.



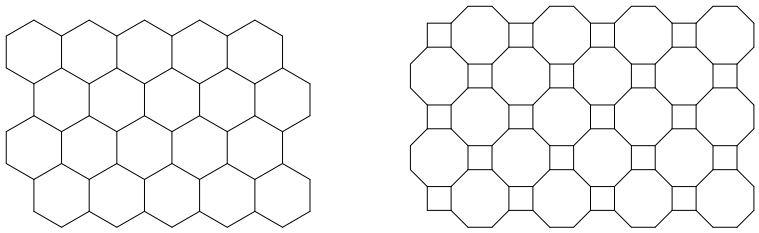
8.25. Za neparno $n \geqslant 5$ nacrtati pravilnu zvezdu n-kraku ako je dato jedno njeno teme i dužina kraka

(a) koristeći grafiku sa kornjačom;

(b) koristeći samo naredbu `Line`.

(Uputstvo: Zbir uglova u temenima zvezde n-krake je π. Na slici pored nacrtana je zvezda 9-kraka.)

8.26. Napisati Pascal program koji koristeći grafiku sa kornjačom ilustruje



(a) (b)

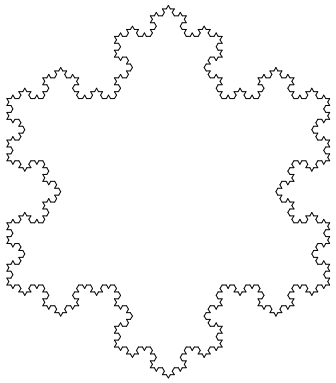
Slika 8.8: Zadatak 8.26

(a) popločavanje ravni pravilnim šestouglovima, Sl. 8.8 (a).

(b) popločavanje ravni kvadratima i pravilnim osmouglovima, Sl. 8.8 (b).

†**8.27.** Napisati Pascal proceduru koja za dato n crta Kochovu krivu reda n koristeći samo naredbu `Line`.

8.28. Napisati Pascal program koji crta Kochovu pahuljicu reda n, Sl. 8.9. Kochova pahuljica reda n se dobija tako što se svaka strana jednakostraničnog trougla zameni Kochovom krivom reda n.

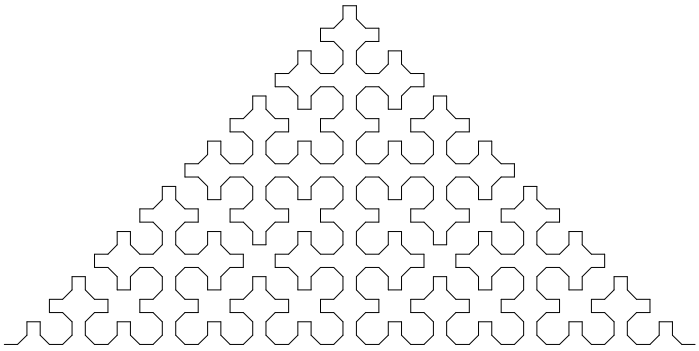


Slika 8.9: Kochova pahuljica

8.29. *Kriva Sierpinskog reda n sa parametrom a,* Sl.8.10, je kriva koja je rekurzivno definisana ovako:

- ako je $n = 0$ to je samo duž dužine a;
- u ostalim slučajevima radimo sledeće:
  - okreni se u mestu za ugao $(-1)^n \frac{\pi}{4}$;
  - nacrtaj krivu Sierpinskog reda $n-1$ sa parametrom a;
  - okreni se u mestu za ugao $(-1)^{n+1} \frac{\pi}{4}$;
  - idi napred a koraka;
  - okreni se u mestu za ugao $(-1)^{n+1} \frac{\pi}{4}$;
  - nacrtaj krivu Sierpinskog reda $n-1$ sa parametrom a;
  - okreni se u mestu za ugao $(-1)^n \frac{\pi}{4}$;

Napisati Pascal proceduru koja crta krivu Sierpinskog reda n sa parametrom a.



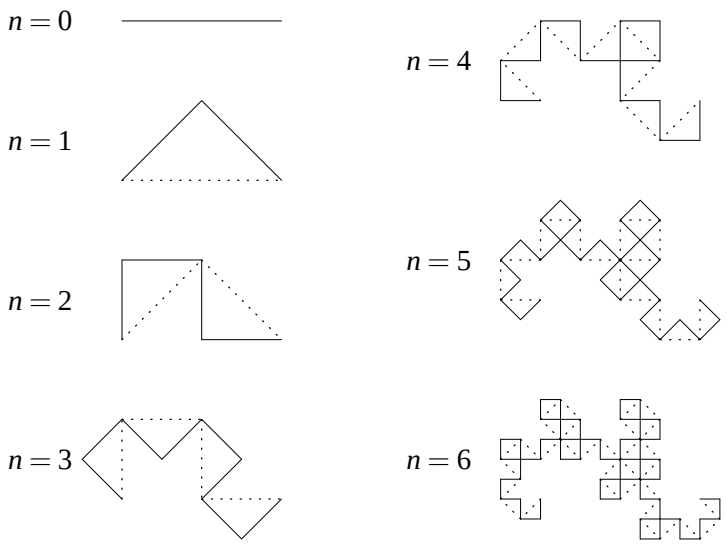
Slika 8.10: Kriva Sierpinskog

8.30. *Heighwayov zmaj reda n sa parametrom a* za prvih nekoliko vrednosti broja n prikazan je na Sl.8.11. Napisati Pascal proceduru koja crta Heighwayovog zmaja reda n sa parametrom a.

8.5 Krug

Krug sa centrom $C(c_0, c_1)$ i poluprečnikom r je skup tačaka $X(x,y)$ u ravni koje su na rastojanju r od tačke C: $d(X,C) = r$. Tako dobijamo da koordinate tačke X moraju zadovoljavati sledeće:

$$\sqrt{(x-c_0)^2 + (y-c_1)^2} = r,$$



Slika 8.11: Heighwayov zmaj

odnosno, nakon kvadriranja (iz čisto estetskih razloga),

$$(x-c_0)^2+(y-c_1)^2=r^2.$$

Ova jednačina se zove *jednačina kruga* zato što neka tačka pripada krugu ako i samo ako njene koordinate zadovoljavaju jednačinu. Za ovakvu jednačinu kažemo da je *u implicitnom obliku* zato što nije "rešena po y (ili x)".

Unutrašnjost kruga sa centrom C i poluprečnikom r je skup tačaka X takvih da je $d(C,X)<r$, a *spoljašnjost istog kruga* je skup tačaka X za koje je $d(C,X)>r$. Odatle se lako dobija da tačka pripada unutrašnjosti kruga ako njene koordinate zadovoljavaju nejednačinu $(x-c_0)^2+(y-c_1)^2<r^2$, a spoljašnjosti ako zadovoljavaju nejednačinu $(x-c_0)^2+(y-c_1)^2>r^2$. Prema tome, ako za neku tačku $M(x_0,y_0)$ stavimo

$$\lambda_M=(x_0-c_0)^2+(y_0-c_1)^2-r^2,$$

tada je $\lambda_M=0$ ukoliko tačka M leži na krugu. Ako tačka M ne leži na krugu ova vrednost nije nula. Kao i u slučaju duži, *znak* broja λ_M određuje da li se tačka M nalazi u unutrašnjosti ili spoljašnjosti kruga. Da rezimiramo:

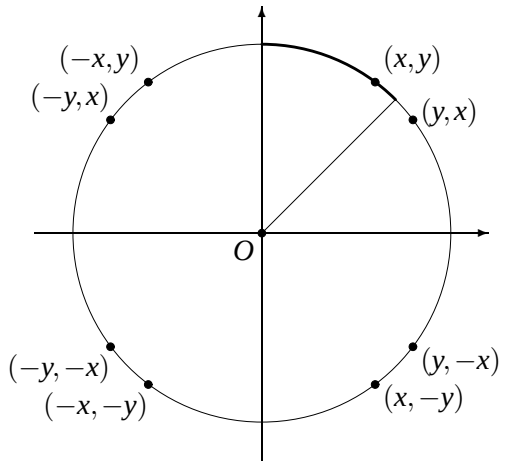
- ako je $\lambda_M=0$, tačka M leži na krugu;
- ako je $\lambda_M>0$, tačka M leži u spoljašnjosti kruga;
- ako je $\lambda_M<0$, tačka M leži u unutrašnjosti kruga.

8.6 Bresenhamov inkrementalni algoritam za crtanje kruga

Krug kao geometrijski objekt nema debljinu i sastoji se od beskonačno mnogo tačaka. Kao i kod crtanja duži, problem crtanja kruga se sastoji u tome da se odabere takva kolekcija piksela koja što vernije reprezentuje taj geometrijski objekt na rasteru ekrana. Koristeći istu ideju kao kod algoritma za crtanje linije, Bresenham je predložio jedan efikasan algoritam, koga ćemo sada predstaviti[3]. Kao i ranije, dobićemo algoritam koji koristi isključivo celobrojnu aritmetiku, što ga čini izuzetno povoljnim za implementaciju u hardveru.

Da bismo pojednostavili računanje, daćemo algoritam samo u slučaju kruga čiji centar je u koordinatnom početku. Jednostavnom modifikacijom dobijamo algoritam koji crta krug sa proizvoljnim centrom, što ostavljamo za vežbu.

Zbog sveopšte simetrije kruga, dovoljno je da račun izvedemo samo za jedan njegov deo. Posmatraćemo kružni luk između y-ose i prave $y = x$. Kada odredimo piksel (x,y) iz tog isečka, preostalih sedam simetričnih isečaka popunjavamo tačkama (y,x), $(-x,y)$, $(y,-x)$, $(x,-y)$, $(-y,x)$, $(-x,-y)$, $(-y,-x)$, kako je to pokazano na Sl. 8.12.



Slika 8.12: Osam tačaka kruga simetričnih datoj tački

Ovim se dobija na brzini zato što izvodimo osam puta manje računskih operacija, a procedura koja realizuje osvetljavanje osam simetričnih tačaka je veoma jednostavna:

[3] Bresenham, J. E., *A Linear Algorithm for Incremental Digital Display of Circular Arcs*, Communications of the ACM, 20(2) 1977, 100–106

```
procedure Put8Pixels(x, y : integer);
begin
    PutPixel( x,  y, MyColor); PutPixel( y,  x, MyColor);
    PutPixel( y, -x, MyColor); PutPixel(-x,  y, MyColor);
    PutPixel( x, -y, MyColor); PutPixel(-y,  x, MyColor);
    PutPixel(-x, -y, MyColor); PutPixel(-y, -x, MyColor)
end;
```

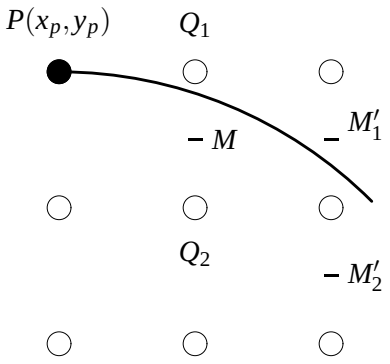
Jednačina kruga poluprečnika r sa centrom u koordinatnom početku ima oblik $x^2+y^2=r^2$ ili, što je ekvivalentno,

$$4x^2+4y^2-4r^2=0.$$

(Jednačinu smo pomnožili sa 4 iz tehničkih razloga.) Za tačku $A(p,q)$, neka je $\lambda_A := 4p^2+4q^2-4r^2$ izraz koji se dobija kada koordinate tačke A uvrstimo u izraz na desnoj strani. Jasno je da broj λ_A ima sledeću osobinu:

- ako je $\lambda_A < 0$, tačka A pripada unutrašnjosti kruga;
- ako je $\lambda_A = 0$, tačka A pripada krugu;
- ako je $\lambda_A > 0$, tačka A pripada spoljašnjosti kruga.

Pretpostavimo da smo u prethodnom koraku osvetlili piksel $P(x_p,y_p)$. U sledećem koraku ćemo osvetliti ili piksel $Q_1(x_p+1,y_p)$ ili piksel $Q_2(x_p+1,y_p-1)$ prema sledećem kriterijumu. Posmatrajmo tačku $M(x_p+1,y_p-\frac{1}{2})$ koja je središte duži $[Q_1Q_2]$. Ako se ona nalazi u unutrašnjosti kruga, osvetlićemo piksel Q_1; u suprotnom ćemo osvetliti piksel Q_2. Da li se tačka M nalazi u unutrašnjosti kruga ili ne utvrđujemo na osnovu znaka broja λ_M.



Polazeći od tačke sa koordinatama $(0,r)$ algoritam koji crta krug sada ima sledeći oblik:

```
x := 0;
y := r;
Put8Pixels(x, y);
while  y > x  do begin
    if  λ_{M(x+1,y-1/2)} ⩽ 0  then
        x := x+1
    else begin
        x := x+1;
        y := y-1
    end;
    Put8Pixels(x, y)
end
```

Izraz za λ_M se može računati inkrementalno. Prva odluka se donosi na osnovu vrednosti u tački $M_0(1, r-\frac{1}{2})$:

$$\lambda_{M_0} = 4 \cdot 1^2 + 4(r-\frac{1}{2})^2 - 4r^2 = 5 - 4r.$$

Ako smo osvetlili piksel Q_1 sledeća odluka se donosi na osnovu vrednosti izraza λ u tački $M_1'(x_p+2, y_p-\frac{1}{2})$. Ako smo osvetlili piksel Q_2, odluka se donosi na osnovu vrednosti izraza λ u tački $M_2'(x_p+2, y_p-\frac{3}{2})$. Kako je

$$\delta_1 := \lambda_{M_1'} - \lambda_M = 8x_p + 12$$
$$\delta_2 := \lambda_{M_2'} - \lambda_M = 8(x_p - y_p) + 20$$

inkrementalna verzija algoritma izgleda ovako:

```
procedure Circle0(R : integer);
var
  x, y, lambda : integer;
begin
  x := 0; y := R;
  lambda := 5 - 4*R;
  Put8Pixels(x, y);

  while y > x do begin
    if lambda <= 0 then begin
        lambda := lambda + 8*x + 12;
        x := x + 1
    end
    else begin
        lambda := lambda + 8*x - 8*y + 20;
```

```
        x := x + 1;
        y := y - 1
    end;
    Put8Pixels(x, y)
  end
end;
```

Zadaci.

8.31. Ispitati položaj tačke $(1,-3)$ prema krugu

(a) $x^2+y^2=1$;

(b) $x^2+y^2-8x-4y-14=0$;

(c) $x^2+y^2-8x+4y-8=0$.

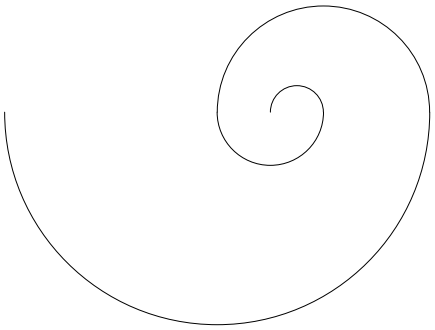
8.32. Odrediti jednačinu kruga čiji centar je u tački $(5,-2)$, a koji prolazi kroz tačku $(-1,5)$.

8.33. Naći jednačinu kruga čiji prečnik je duž $[AB]$, gde je $A(5,-1)$ i $B(-3,7)$.

8.34. Naći tačke preseka kruga $x^2+y^2-4x+4y+4=0$ sa koordinatnim osama.

8.35. Napisati Pascal proceduru koja crta krug poluprečnika `R` sa centrom u tački `(x0, y0)`. (Napomena: dovoljno je samo malo promeniti proceduru `Put8Pixels`.)

8.36. Napisati Pascal program koji crta gornji polukrug poluprečnika `R` sa centrom u tački `(x0, y0)`. (Napomena: dovoljno je samo malo promeniti proceduru `Put8Pixels`.)



Slika 8.13: Prva četiri zavoja spirale iz Zadatka 8.37

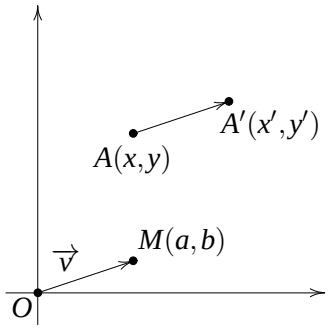
8.37. Napisati Pascal program koji crta prvih n zavoja spirale koja se sastoji od polukrugova, Sl. 8.13. Prvi zavoj spirale je gornji polukrug poluprečnika

r. Drugi zavoj je donji polukrug poluprečnika $2r$ koji ima centar u krajnjoj levoj tački prvog zavoja. Treći zavoj je gornji polukrug poluprečnika $4r$ koja ima centar u desnoj krajnjoj tački drugog zavoja, itd.

8.38. Napisati Pascal proceduru koja crta krug debljine w piksela. (Uputstvo: jedan način da se ovo uradi se sastoji u tome da se više puta pozove procedura `Circle0` sa poluprečnicima iz intervala $[r-\frac{w}{2}, r+\frac{w}{2}]$; drugi način se sastoji u tome da se umesto procedure `PutPixel(x, y, MyColor)` pozove procedura `Square(x, y, w)` koja crta popunjen kvadrat sa centrom u tački (x,y) i poluprečnikom w.)

8.7 Transformacije u ravni

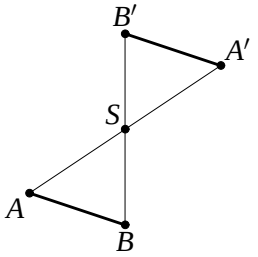
Sada ćemo opisati nekoliko standardnih ravanskih transformacija koje imaju veliku primenu u računarskoj grafici.



Translacija za vektor $\overrightarrow{v}$ je preslikavanje ravni na sebe koje proizvoljnu tačku A preslikava na tačku A' sa osobinom $\overrightarrow{AA'} = \overrightarrow{v}$.

To znači da se slika tačke A dobija tako što se tačka A "pomeri" za vektor $\overrightarrow{v}$. Da bismo dobili analitički izraz za translaciju, uzmimo da je $\overrightarrow{v} = \overrightarrow{OM}$ gde tačka M ima koordinate (a,b). Ako tačka A ima koordinate (x,y), a tačka A' koordinate (x',y'), tada se lako vidi da je

$$x' = x + a$$
$$y' = y + b.$$



Centralna simetrija u odnosu na tačku S, je preslikavanje kojim se tačka A preslikava na tačku A' tako da je tačka S središte duži $[AA']$.

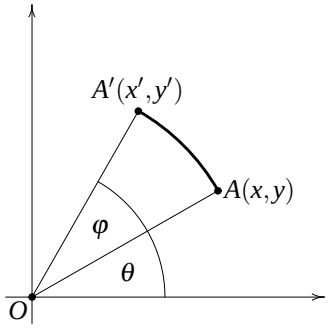
Neka je $S(p,q)$ centar simetrije i neka tačka A ima koordiate (x,y), a tačka A' koordinate (x',y'). Prema definiciji, S je središte duži $[AA']$, pa je $\frac{x+x'}{2} = p$ i $\frac{y+y'}{2} = q$. Odatle dobijamo

$$x' = 2p - x$$
$$y' = 2q - y.$$

Specijalno, simetrija u odnosu na koordinatni početak ima sledeći analitički izraz: $x' = -x$, $y' = -y$.

Rotacija oko tačke S za orijentisani ugao φ je preslikavanje kojim se tačka A preslikava na tačku A' koja ima sledeće osobine: $[SA] \cong [SA']$ i orijentisani uglovi

$\angle ASA'$ i φ su jednaki.



Prvo ćemo izvesti analitički izraz za rotaciju oko koordinatnog početka. Neka je $A(x,y)$ proizvoljna tačka i neka je $A'(x',y')$ slika tačke A dobijena rotacijom oko koordinatnog početka za ugao φ. Tada je $\angle AOA' = \varphi$. Neka je θ ugao koga sa pozitivnim smerom x-ose zaklapa vektor $\overrightarrow{OA}$. Ako dužinu duži $[OA]$ označimo sa r, onda je (polarne koordinate tačke):

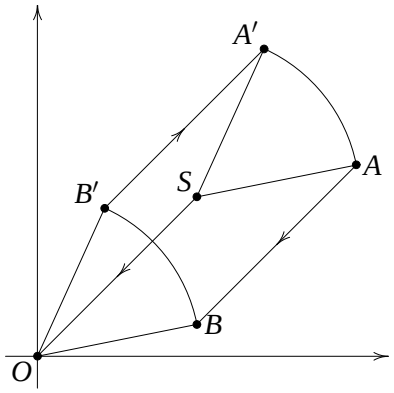
$$x = r\cos\theta \qquad x' = r\cos(\theta+\varphi)$$
$$y = r\sin\theta \qquad y' = r\sin(\theta+\varphi)$$

Kada raspišemo izraze za x' i y' primenom adicionih formula za kosinus zbira, odnosno, sinus zbira, dobijamo:

$$x' = r\cos\theta\cos\varphi - r\sin\theta\sin\varphi$$
$$y' = r\sin\theta\cos\varphi + r\cos\theta\sin\varphi.$$

Uvrštavanjem izraza za x i y, na kraju dobijamo:

$$x' = x\cos\varphi - y\sin\varphi$$
$$y' = x\sin\varphi + y\cos\varphi,$$



Da bismo odredili analitički izraz rotacije oko proizvoljne tačke, prvo ćemo primetiti da se rotacija oko proizvoljne tačke može predstaviti kao niz koji se sastoji od translacije, rotacije oko koordinatnog početka i još jedne translacije. Dakle, kada želimo da zarotiramo tačku A oko tačke S za ugao φ, možemo to učiniti tako što ćemo prvo sve translirati za vektor $\overrightarrow{SO}$, čime se tačka S preslikava u tačku O, a tačka A u neku tačku B. Potom tačku B rotiramo za ugao φ oko koordinatnog početka i dobijamo tačku B'. Na kraju tačku B' transliramo za vektor $\overrightarrow{OS}$ i tako dobijamo tačku A'. Sada se lako dobija da rotacija oko tačke $S(p,q)$ za ugao φ ima sledeći oblik:

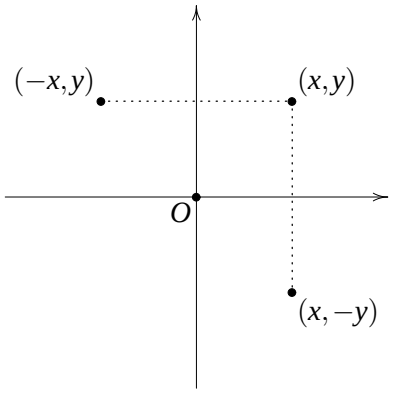
$$x' = (x-p)\cos\varphi - (y-q)\sin\varphi + p$$
$$y' = (x-p)\sin\varphi + (y-q)\cos\varphi + q.$$

Osna simetrija u odnosu na pravu s je preslikavanje kojim se tačka A preslikava na tačku A' tako da je prava s simetrala duži $[AA']$. Analitički izraz za osnu simetriju u odnosu na osu Ox je veoma jednostavan:

$$x' = x$$
$$y' = -y,$$

kao za simetriju u odnosu na osu Oy:

$$x' = -x$$
$$y' = -y.$$



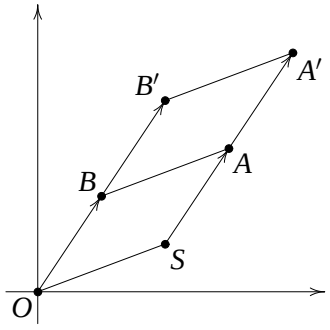
U opštem slučaju, analitički izraz za osnu simetriju je relativno komplikovan i time se u ovom kursu nećemo baviti.

Homotetija sa centrom S i koeficijentom $\alpha \neq 0$ preslikava tačku A na tačku A' tako da je onda je $\overrightarrow{SA'} = \alpha \cdot \overrightarrow{SA}$.

Homotetija sa centrom u koordinatnom početku i koeficijentom α ima veoma jednostavan analitički oblik. Ako A i A' imaju koordinate, redom, (x,y) i (x',y'), onda je

$$x' = \alpha \cdot x$$
$$y' = \alpha \cdot y.$$

Kada želimo da tačku A preslikamo homotetijom sa centrom u tački $S(p,q)$ i sa koeficijentom α, možemo to učiniti tako što ćemo prvo sve translirati za vektor $\overrightarrow{SO}$, čime se tačka S preslikava u tačku O, a tačka A u neku tačku B. Potom tačku B preslikamo homotetijom sa centrom O i koeficijentom α i dobijamo tačku B'. Na kraju tačku B' transliramo za vektor $\overrightarrow{OS}$ i tako dobijamo tačku A'. Odatle dobijamo da homotetija sa centrom $S(p,q)$ i koeficijentom α ima sledeći oblik:



$$x' = \alpha \cdot (x - p) + p$$
$$y' = \alpha \cdot (y - q) + q.$$

Zadaci.

8.39. Napisati Pascal program koji od korisnika učitava tačke A, B i S i potom crta paralelogram $ABCD$ čije dijagonale se seku u S.

8.40. Napisati Pascal program koji od korisnika učitava tačke P, Q i R i potom crta trougao ABC tako da je P središte duži $[BC]$, Q središte duži $[AC]$, a R središte duži $[AB]$.

†**8.41.** Napisati Pascal program koji od korisnika učitava n tačaka i pravu l, a potom crta te objekte, kao i najmanji pravougaonik čije dve strane su paralelne pravoj l, a koji u svojoj unutrašnjosti sadrži date tačke.

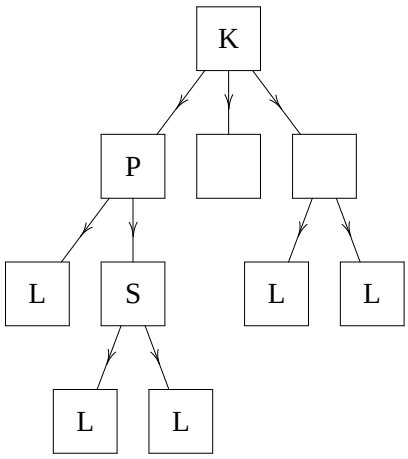
8.42. Napisati pascal program koji od korisnika učitava tačku A i prave b i c, a potom crta te objekte, kao i tačke $B \in b$ i $C \in c$ sa osobinom da je trougao ABC jednakostraničan.

8.43. Napisati Pascal program koji od korisnika učitava tačke A i B i crta kvadrat $ABCD$.

8.44. Napisati Pascal program koji od korisnika učitava tačke A i B crta trougao ABC kod koga je $\angle A = \pi/3$ i $\angle B = \pi/4$.

Glava 9

Stabla



Stabla predstavljaju razgranate dinamičke strukture: to su strukture kod kojih jedna kućica može da pokazuje na više kućica. Pri tome se insistira da struktura bude još i *aciklična*, što znači da kućice ne smeju da obrazuju cikluse, i da svaka kućica ima tačno jednog *prethodnika*. Na slici pored prikazano je jedno stablo. Kućica P je *prethodnik* kućice S, a kućica S je *sledbenik* kućice P.

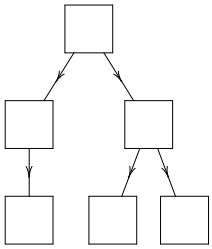
Svako stablo ima tačno jednu kućicu koja nema prethodnika. To je *koren* stabla (na slici pored koren je označen sa K). Stablo ima najmanje jednu kućicu koja nema sledbenika. Kućice koje nemaju sledbenike se zovu *listovi* stabla (na slici pored listovi su označeni sa L).

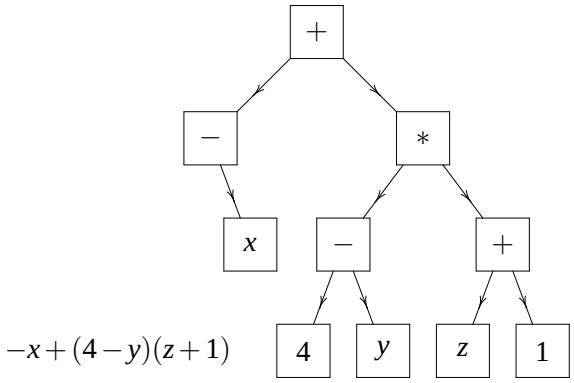
U ovoj glavi ćemo se baviti jednom specijalnom klasom stabala koja se zovu *binarna stabla*. Videćemo kako se formiraju i obilaze binarna stabla, i kako se binarna stabla mogu koristiti za reprezentaciju algebarskih izraza. Nakon toga ćemo uvesti pojam uređenog binarnog stabla i videćemo nekoliko standardnih algoritama nad takvim stablima.

Algoritmi za manipulisanje binarnim stablima su uglavnom rekurzivni, tako da će ova glava pružiti još jednu lepu priliku da učvrstimo naše razumevanje rekurzije.

9.1 Binarna stabla

Binarno stablo je stablo kod koga svaka kućica ima *najviše dva* sledbenika. Stablo koje je prikazano na početku glave nije binarno zato što njegov koren ima tri sledbenika. Stablo na slici pored jeste binarno, jer svaki kućica ima 0, 1 ili 2 sledbenika. Binarna stabla su prirodni modeli algebarskih izraza, kako je to pokazano na Sl. 9.1. Više o ovoj vezi ćemo videti u Zadacima 9.17 i 9.18.

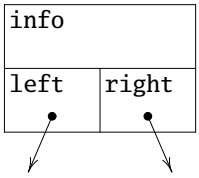




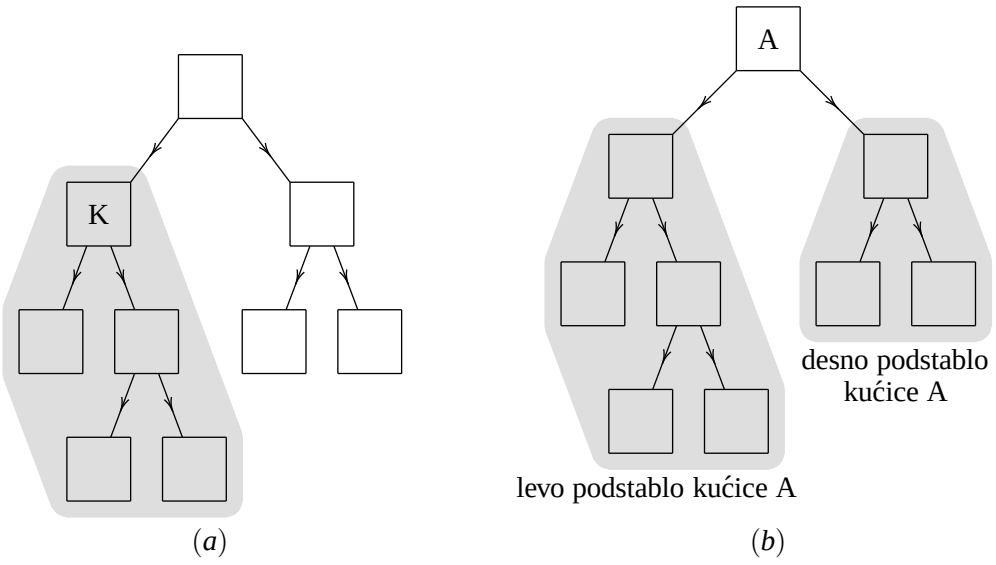
Slika 9.1: Algebarski izraz i njemu pridruženo binarno stablo

Binarno stablo se u programskom jeziku Pascal predstavlja kao pokazivačka struktura kod koje svaka kućica ima dva pokazivača: jedan pokazuje na *levog* naslednika, a drugi na *desnog*. Deklaracija izgleda ovako:

```
type
  BinTree = ↑Elem;
  Elem = record
    info : ⟨neki tip⟩;
    left, right : BinTree
  end;
```



U binarnom stablu svaka kućica je koren nekog *podstabla* tog stabla. Na primer, na Sl. 9.2 (a) kućica označena sa K je koren podstabla koga čine osenčene kućice. Tako dobijamo da svaka kućica binarnog stabla pokazuje na dva podstabla: na njeno *levo podstablo* i na njeno *desno podstablo*, Sl. 9.2 (b). Naravno, jedno ili oba podstabla svake kućice mogu biti prazna stabla. Ovakva struktura predstavlja osnovu svih algoritama za manipulisanje binarnim stablima.



Slika 9.2: Levo i desno podstablo jedne kućice

Primer. Napisati proceduru `KillTree` koja uništava binarno stablo i tako oslobađa prostor na heapu koji je bio rezervisan za njegove elemente. Procedura rekurzivnim pozivom uništi prvo levo, pa desno podstablo, a na kraju i koren stabla.

```
procedure KillTree(var t : BinTree);
begin
  if t <> nil then begin
    KillTree(t↑.left);
    KillTree(t↑.right);
    dispose(t)
  end
end;
```

Primer. Napisati funkciju `Count` koja određuje broj kućica u binarnom stablu. Koristiti činjenicu da prazno stablo nema nijednu kućicu, dok se za stabla koja nisu prazna broj kućica određuje kao 1 (za koren) + broj kućica u levom podstablu + broj kućica u desnom podstablu.

```
function Count(t : BinTree) : integer;
begin
  if t = nil then
    Count := 0
  else
    Count := 1 + Count(t↑.left)
               + Count(t↑.right)
end;
```

Problem formiranja binarnog stabla je znatno složeniji od primera koje smo do sada videli. Pre svega, potrebno je da opišemo reprezentaciju binarnog stabla nizom karaktera, recimo, i da pokažemo kako se taj niz karaktera može konvertovati u odgovarajuću pokazivačku strukturu na heapu.

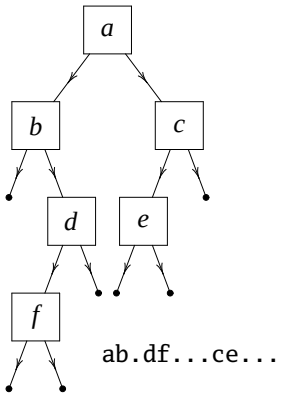
Zbog jednostavnosti, koristićemo deklaraciju binarnog stabla kod koga polje `info` ima tip `char`, kako je to pokazano pored.

```
type
  BinTree = ↑Elem;
  Elem = record
    info : char;
    left, right : BinTree
  end;
```

Simbolom "`.`" (tačka) označićemo prazno binarno stablo. Neprazna stabla ćemo predstaviti nizom karaktera koji se formira ovako:

$$\langle ZapisStabla\rangle \equiv$$
$$\equiv \langle info\rangle\langle ZapisLPodstabla\rangle\langle ZapisDPodstabla\rangle$$

gde je $\langle info\rangle$ karakter koji se nalazi u polju `info` korena stabla. Pretpostavljamo, naravno, da nijedna kućica u stablu ne sadrži simbol "`.`". Primer stabla i njegove reprezentacije je dat na slici pored, a proces konstrukcije te reprezentacije opisan je korak po korak na Sl. 9.3.

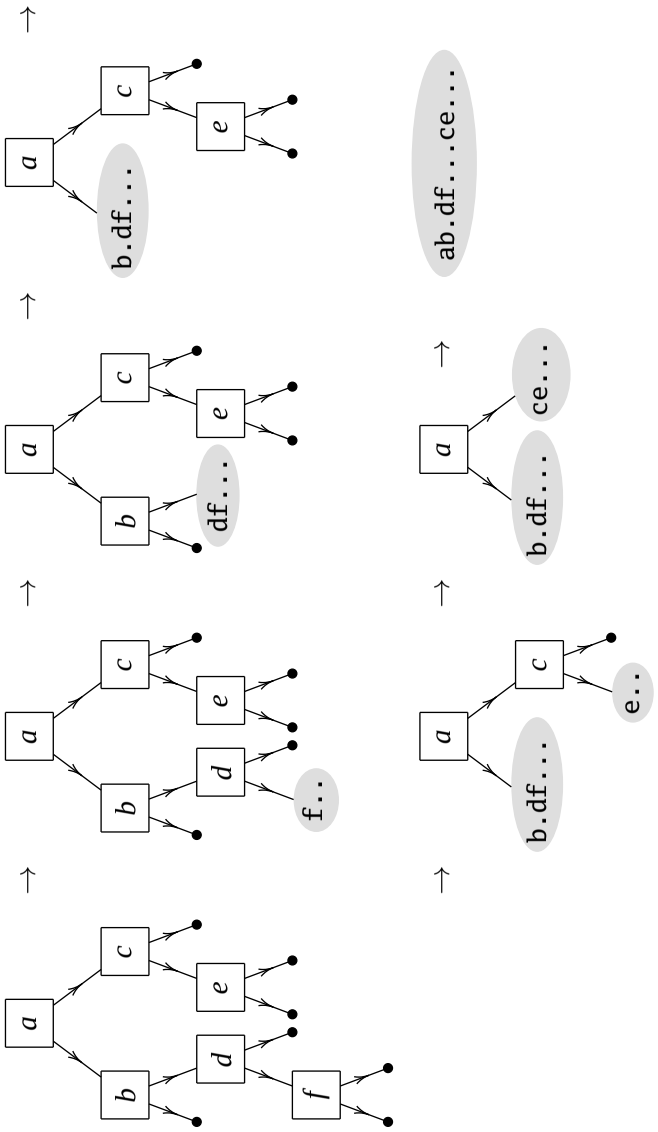


Procedura `ReadTree` učitava niz karaktera iz ulaznog reda koji predstavlja reprezentaciju binarnog stabla i na heapu formira odgovarajuće stablo.

Da bismo ispisali elemente binarnog stabla treba da ispišemo sadržaj polja `info` u korenu stabla, da ispišemo rekurzivno levo podstablo, i da ispišemo rekurzivno desno podstablo. Naravno, kada naiđemo na prazno stablo, ispisujemo "`.`" (tačka). Procedura `WriteTree` koja ispisuje binarno stablo je jednostavna i data je pored.

Naravno, string koga ispisuje procedura `WriteTree` nije ništa drugo do reprezentacija binarnog stabla koju smo opisali na početku priče o učitavanju binarnog stabla.

```
procedure ReadTree(var t : BinTree);
var
  c : char;
begin
  read(c);
  if c = '.' then t := nil
  else begin
    new(t);
    t↑.info := c;
    ReadTree(t↑.left);
    ReadTree(t↑.right)
  end
end;


procedure WriteTree(t : BinTree);
begin
  if t = nil then write('.')
  else begin
    write(t↑.info);
    WriteTree(t↑.left);
    WriteTree(t↑.right)
  end
end;
```



Slika 9.3: Proces konstrukcije reprezentacije binarnog stabla

Zadaci.

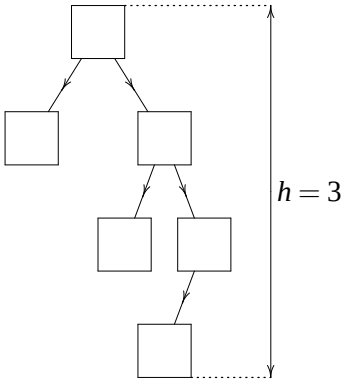
9.1. *Celobrojno stablo* je stablo čiji elementi su tipa `integer`. Napisati funkciju

```
function SumTree(t : IntTree) : integer;
```

koja određuje sumu svih elemenata celobrojnog stabla, pri čemu se uzima da je suma elemenata praznog stabla jednaka 0. Tip `IntTree` kojim se opisuju celobrojna stabla dat je sa:

```
type
  IntTree = ↑Elem;
  Elem = record
    info : integer;
    left, right : IntTree
  end;
```

9.2. *Visina stabla* je dužina najdužeg mogućeg puta koji vodi od korena stabla do nekog od listova stabla (Sl. 9.4).



Slika 9.4: Visina stabla

Napisati funkciju

```
function Height(t : BinTree) : integer;
```

koja određuje visinu stabla `t`. Visina stabla se računa na sledeći način:

- visina praznog stabla je -1,
- u ostalim slučajevima visina h_T stabla T se računa ovako:

$$h_T = 1 + \max(h_L, h_R)$$

gde je h_L visina levog, a h_R visina desnog podstabla koje odgovara korenu stabla T.

Napomena: za visinu praznog stabla uzimamo -1 kako bismo dobili da visina stabla koje ima samo koren (i kome su, stoga, i levo i desno podstablo prazni) bude 0.

9.3. Napisati funkciju

```
function Member(t : BinTree; c : char) : Boolean;
```

koja proverava da li se karakter `c` pojavljuje u stablu `t`.

9.4. Napisati funkciju

```
function CountMember(t : BinTree; c : char) : integer;
```

koja utvrđuje koliko se puta karakter `c` pojavljuje u stablu `t`.

9.5. Napisati funkciju

```
function CountLeaves(t : BinTree) : integer;
```

koja utvrđuje koliko listova ima stablo `t`. List u stablu je kućica koja nema ni levo ni desno podstablo. Prazno stablo nema listove.

9.6. *Obilazak stabla* predstavlja sistematski način da posetimo svaku kućicu u stablu. *Preorder* obilazak stabla se sastoji u tome da

- ☞ obiđemo koren stabla,
- rekurzivno obiđemo levo podstablo, i
- rekurzivno obiđemo desno podstablo.

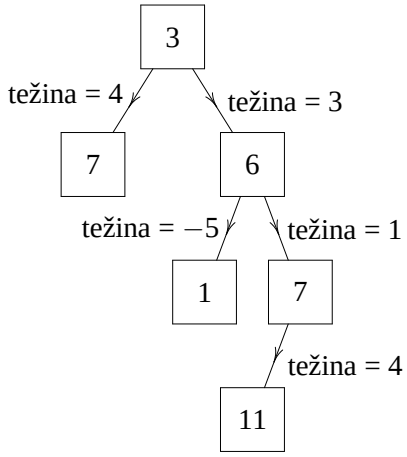
Ovaj način obilaska stabla se zove "preorder" zato što koren stabla obiđemo *pre* nego što obiđemo levo i desno podstablo (lat. pre = pre, ispred; lat. ordo = poredak). *Inorder* obilazak stabla se sastoji u tome da

- rekurzivno obiđemo levo podstablo,
- ☞ obiđemo koren stabla, i
- rekurzivno obiđemo desno podstablo.

Ovaj način obilaska stabla se zove "inorder" zato što koren stabla obiđemo *između* obilazaka levog i desnog podstabla (lat. in = u, između). *Postorder* obilazak stabla se sastoji u tome da

- rekurzivno obiđemo levo podstablo,
- rekurzivno obiđemo desno podstablo, i
- ☞ obiđemo koren stabla.

Ovaj način obilaska stabla se zove "postorder" zato što koren stabla obiđemo *nakon* obilazaka levog i desnog podstabla (lat. post = posle).



Slika 9.5: Celobrojno stablo i težine njegovih grana

Na primer, procedura `WriteTree` ispisuje elemente stabla koristeći preorder obilazak, dok procedura `ReadTree` učitava reprezentaciju stabla koja je formirana prema redosledu kućica u preorder obilasku stabla (prvo se navodi vrednost polja `info` korena stabla, a onda se navodi struktura levog podstabla, pa struktura desnog podstabla).

(a) Napisati proceduru

```
procedure WriteTreeInord(t : BinTree);
```

koja ispisuje elemente stabla koristeći inorder obilazak stabla.

(b) Napisati proceduru

```
procedure WriteTreePostord(t : BinTree);
```

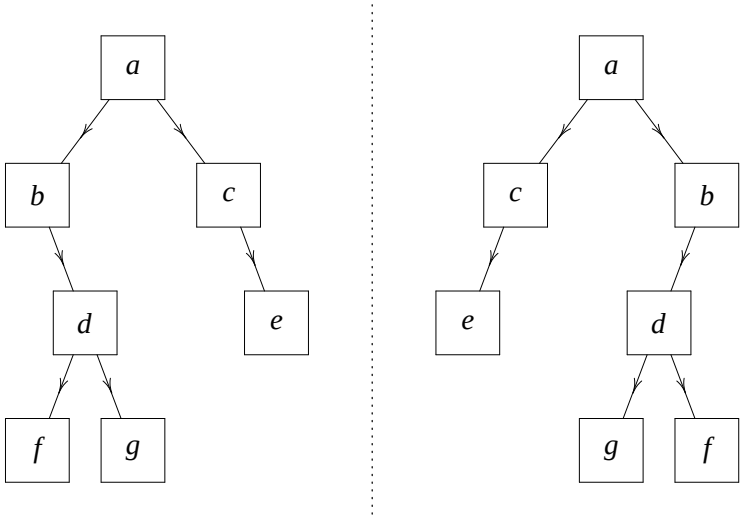
koja ispisuje elemente stabla koristeći postorder obilazak stabla.

(c) Za razliku od preorder i postorder zapisa stabla, inorder zapis stabla ne određuje stablo jednoznačno (zato nam kod pisanja aritmetičkih izraza trebaju zagrade!). Dati primer dva različita stabla koja imaju isti inorder zapis.

9.7. U celobrojnom stablu se vrednost polja `info` ponekad zove i *težina* odgovarajuće kućice. Napisati funkciju

```
function HeaviestLeaf(t : IntTree) : integer;
```

koja vraća težinu najtežeg lista u stablu `t`. Pretpostavlja se da je `t <> nil`.



Slika 9.6: Binarno stablo i njemu simetrično stablo

9.8. Ako je u celobrojnom stablu kućica A prethodnik kućice B, onda se broj

$$W_{\mathtt{AB}} = \mathtt{B.info} - \mathtt{A.info}$$

zove *težina grane* AB (primer je dat na Sl. 9.5). Napisati funkciju

```
function HeaviestBranch(t : IntTree) : integer;
```

koja vraća težinu najteže grane u stablu `t`. Ako stablo ima manje od dve kućice, kao rezultat rada funkcije vratiti `-maxint`.

9.9. Napisati funkciju

```
function Mirror(t : BinTree) : BinTree;
```

koja konstruiše novo stablo simetrično stablu `t`. Dva stabla su *simetrična* ako se jedno od njih može dobiti osnom simetrijom onog drugog. Primer stabla i njemu simetričnog stabla je dat na Sl. 9.5.

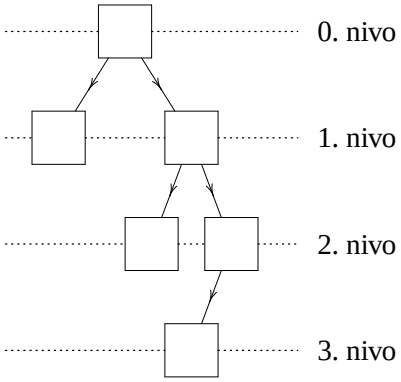
†**9.10.** Napisati funkciju

```
function AllLeaves(t : IntTree) : IntList;
```

koja vraća listu svih listova u stablu `t`.

9.11. *Nivo* određene kućice u stablu je dužina puta koga treba preći od korena stabla do te kućice (primer je dat na Sl. 9.7). Napisati funkciju

```
function LevelOf(t : BinTree; el : BinTree) : integer;
```



Slika 9.7: Nivoi u stablu

koja u stablu `t` utvrđuje nivo kućice na koju pokazuje `el`.

†**9.12.** Napisati funkciju

```
function TheLevel(t : IntTree; L : integer) : IntList;
```

koja vraća listu svih elemenata stabla `t` koji se nalaze na nivou `L`.

9.13. Napisati funkciju

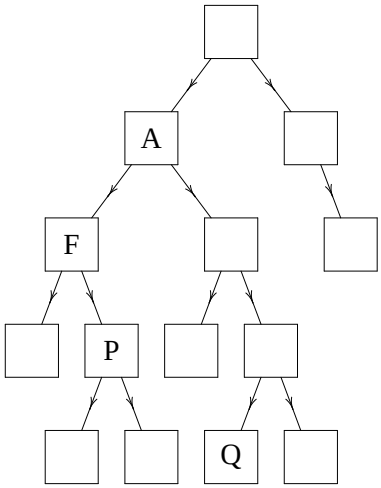
```
function Father(t : BinTree; el : BinTree) : BinTree;
```

koja vraća pokazivač na prethodnika elementa `el` u stablu `t`. Podsetimo se da koren stabla nema prethodnika, pa ukoliko `el` pokazuje na koren stabla, funkcija vraća `nil`. Na primer, na Sl. 9.8, kućica F je prethodnik (*father*) kućice P.

9.14. Napisati funkciju

```
function Ancestor(t: BinTree; p, q: BinTree): BinTree;
```

koja vraća pokazivač na prvog zajedničkog pretka elementa `p` i `q` u stablu `t`. Na primer, na Sl. 9.8, kućica A je prvi zajednički predak (*ancestor*) za kućice P i Q.

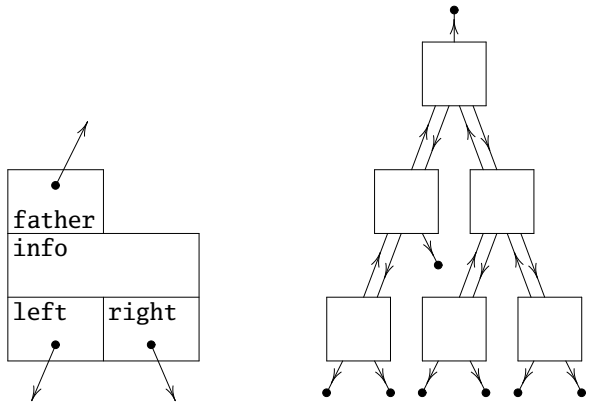


Slika 9.8: Kućica F je prethodnik (*father*) kućice P, a kućica A je prvi zajednički predak (*ancestor*) za kućice P i Q

9.15. Kao što smo videli u prethodna dva zadatka, nije lako vratiti se do prethodnika neke kućice u binarnom stablu. Zato se često posmatraju binarna stabla kod kojih svaka kućica osim pokazivača na levo i desno podstablo ima i pokazivač na svog prethodnika. Takvo stablo se zove *dvostruko povezano stablo* i predstavlja analogon dvostruko povezane liste, a u programskom jeziku Pascal se može opisati ovako:

```
type
  DBinTree = ↑Elem;
  Elem = record
    info : ⟨neki tip⟩;
    left, right, father : DBinTree
  end;
```

Ilustracija i jedan primer dati su na sledećoj slici:



(a) Napisati funkciju

```
function LevelOf(t : DBinTree; el : DBinTree) : integer;
```

koja u dvostruko povezanom stablu `t` utvrđuje nivo kućice `el`.

(b) Napisati funkciju

```
function Mirror(t : DBinTree) : DBinTree;
```

koja konstruiše novo dvostruko povezano stablo koje je simetrično dvostruko povezanom stablu `t`.

(c) Napisati funkciju

```
function Ancestor(t: DBinTree; p, q: DBinTree): DBinTree;
```

koja vraća pokazivač na prvog zajedničkog pretka elementa `p` i `q` u dvostruko povezanom stablu `t`.

9.16. *Ternarno stablo* je stablo u kome kućica može da ima najviše tri sledbenika. U programskom jeziku Pascal ternarno stablo se može opisati na sledeći način:

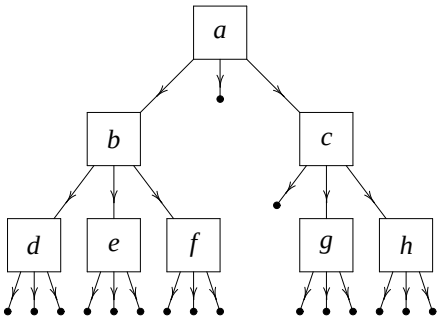
```
type
  TTree = ↑Elem;
  Elem = record
    info : char;
    left, mid, right : TTree
  end;
```

Preorder obilazak ternarnog stabla realizuje se tako što se prvo obiđe koren stabla, a potom se obiđu levo, srednje i desno podstablo. Postorder obilazak ternarnog stabla realizuje se tako što se prvo obiđu levo, srednje i desno podstablo tog stabla, a potom se obiđe koren stabla.

Napisati Pascal program koji iz ulazne datoteke `PREORD3.TXT` učitava niz karaktera dobijenih preorder obilaskom nekog ternarnog stabla i potom u datoteku `POSTORD3.TXT` upisuje niz karaktera koji se dobijaju postorder obilaskom tog stabla. Na primer,

PREORD3.TXT
`abd...e...f....c.g...h...`
POSTORD3.TXT
`...d...e...fb.....g...hca`

Ternarno stablo koje odgovara ovom primeru dato je na sledećoj slici:



†**9.17.** Algebarski izrazi se mogu na prirodan način predstaviti binarnim stablom (videti Sl. 9.1). Napisati Pascal funkciju

```
function ReadExpr() : ExprTree;
```

koja od korisnika učitava korektan algebarski izraz i formira na heapu stablo koje predstavlja taj izraz. Tip `ExprTree` je dat sa

```
type
  ExprTree = ↑Elem;
  Elem = record
    kind : (num, vbl, op);
    ch : char;
    int : integer;
    left, right : ExprTree
  end;
```

Polje `kind` daje informaciju o tome koju vrstu podataka čuva kućica. Ako je `kind = num` kućica predstavlja ceo broj, a polje `int` sadrži vrednost tog broja. Ako je `kind = vbl` kućica predstavlja promenljivu, a polje `ch` sadrži ime promenljive. Konačno, ako je `kind = op` kućica predstavlja neku operaciju, a polje `ch` sadrži oznaku te operacije. Na Sl. 9.9 je dat primer pokazivačke strukture koja odgovara algebarskom izrazu sa Sl. 9.1.

Pravila koja opisuju celobrojne aritmetičke izraze se mogu zapisati ovako:

⟨*Cifra*⟩ ≡ “0” | “1” | “2” | . . . | “9”

⟨*OpMnoženja*⟩ ≡ “∗” | “/” | “%”

⟨*OpSabiranja*⟩ ≡ “+” | “−”

⟨*Broj*⟩ ≡ ⟨*Cifra*⟩ { ⟨*Cifra*⟩ }

⟨*Promenljiva*⟩ ≡ “a” | “b” | “c” | . . . | “z”

⟨*Činilac*⟩ ≡ ⟨*Broj*⟩ | ⟨*Promenljiva*⟩ | “(” ⟨*Izraz*⟩ “)”

⟨*Sabirak*⟩ ≡ ⟨*Činilac*⟩ { ⟨*OpMnoženja*⟩ ⟨*Činilac*⟩ }

⟨*Izraz*⟩ ≡ [“+” | “−”] ⟨*Sabirak*⟩ { ⟨*OpSabiranja*⟩ ⟨*Sabirak*⟩ }

Vertikalna crta u gornjim zapisima označava alternativu, jednu od ponuđenih mogućnosti. Vitičaste zagrade { i } označavaju da se izraz u zagradi može pojaviti proizvoljno mnogo puta, ali se i ne mora pojaviti. Uglaste zagrade [i] označavaju da je izraz u zagradi opcioni – može se pojaviti, a ne mora (ova konvencija za zapisivanje formalnih (tj. veštačkih) jezika se zove EBNF, Extended Backus-Naur Formalism).

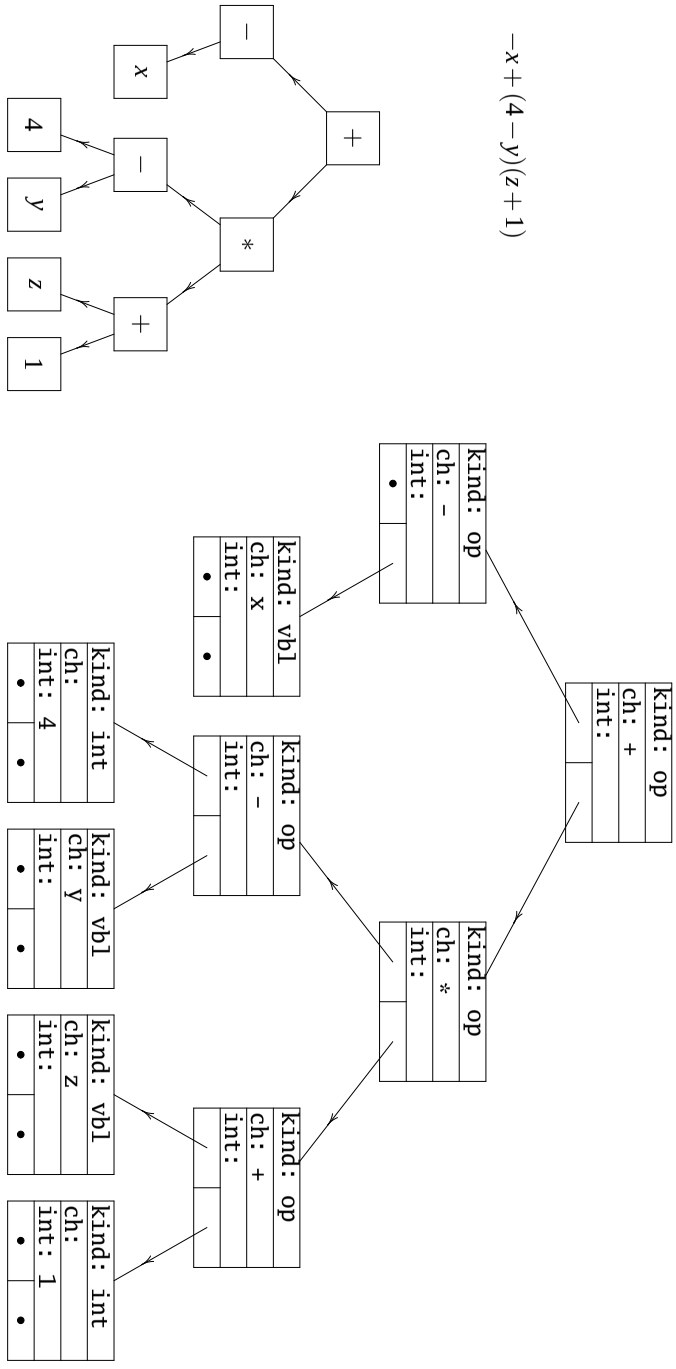
Navedena pravila, dakle, znače sledeće:

- cifra je neki od karaktera 0, . . . , 9;
- operator množenja je neki od karaktera $*$ (množenje), / (količnik pri celobrojnom deljenju) i % (ostatak pri celobrojnom deljenju);
- operator sabiranja je jedan od karaktera $+$ (sabiranje) ili $-$ (oduzimanje);
- broj je proizvoljan niz cifara koji ima bar jednu cifru;
- promenljiva je jedno od slova *a*, *b*, . . . , *z*;
- činilac je broj, promenljiva ili izraz okružen zagradama;
- sabirak je činilac ili niz činilaca razdvojenih operatorom množenja;
- izraz može imati opcioni predznak $+$ ili $-$, nakon čega sledi sabirak, ili niz sabiraka razdvojenih operatorima sabiranja.

Listovi stabla će biti kućice koje predstavljaju cele brojeve ili promenljive. Ostale kućice u stablu predstavljaju operacije. Kućica koja predstavlja binarnu operaciju ima dva podstabla: levo podstablo koje odgovara levom argumentu operacije, i desno podstablo koje odgovara desnom argumentu operacije. Kućica koja predstavlja unarni minus (operacija promene znaka) ima samo desno podstablo: levo podstablo je prazno, a desno odgovara argumentu operacije.

Jedna od osnovnih tehnika analize ovako zadatih izraza se zove *tehnika rekurzivnog supsta* (engl. recursive descent). Prema toj tehnici svakom pravilu dodelimo jednu proceduru ili funkciju koja prihvata karaktere ili poziva druge procedure/funkcije onim redom kojim su oni navedeni u odgovarajućem pravilu. U našem slučaju svakom pravilu ćemo dodeliti jednu funkciju koja vraća pokazivač na stablo izraza koga je funkcije pročitala iz ulazne linije i prepoznala.

Pošto ćemo u nekim situacijama morati da donosimo odluke o tome koje pravilo treba primeniti (tj. koju funkciju pozvati) kako bi se nastavilo sa analiziranjem izraza, standardna preporuka je da uvek treba gledati jedan karakter unapred. Promenljiva koja sadrži sledeći karakter koga treba obraditi zove se *lookahead* (što bi se sa engleskog prevelo otprilike kao “viri unapred”).



Slika 9.9: Reprezentacija celobrojnog algebarskog izraza

†**9.18.** Napisati Pascal proceduru

```
procedure EvalExpr(t : ExprTree; var res : integer;
                                 var ok : Boolean);
```

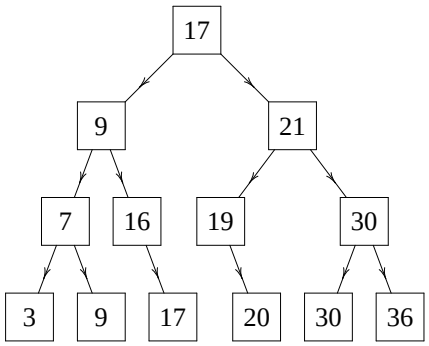
koja računa vrednost algebarskog izraza predstavljenog stablom `t` (za definiciju tipa `ExprTree` i konvencije koje su usvojene pri građenju stabla izraza videti Zadatak 9.17). Ako procedura uspe da izračuna vrednost izraza, argument `res` će sadržati tu vrednost, a argument `ok` će imati vrednost `true`. Ukoliko, međutim, procedura ne može da izračuna vrednost izraza (recimo, usled deljenja nulom), vrednost argumenta `res` neće biti definisana, a argument `ok` će imati vrednost `false`. Vrednost praznog stabla je 0.

Ako izraz sadrži promenljive, vrednosti promenljivih treba učitati od korisnika. Ako se jedna promenljiva javlja više puta u izrazu, korisnik treba njenu vrednost da zada tačno jednom.

9.2 Uređena binarna stabla

Da se podsetimo, celobrojno binarno stablo je binarno stablo kod koga polje `info` svake kućice ima tip `integer` i takva stabla opisujemo tipom `IntTree` (videti Zadatak 9.1). Za celobrojno binarno stablo kažemo da je *uređeno* ili da je *binarno stablo pretraživanja* (engl. binary search tree) ako za svaku kućicu `A` stabla važi sledeće:

- za svaku kućicu `B` levog podstabla koje odgovara kućici `A` imamo `B.info` $\leqslant$ `A.info`, i
- za svaku kućicu `C` desnog podstabla koje odgovara kućici `A` imamo `A.info` $<$ `C.info`.



Drugim rečima, svi brojevi u levom podstablu su manji ili jednaki od broja u korenu stabla, a broj u korenu stabla je strogo manji od svih brojeva u desnom podstablu, i to važi za svaku kućicu stabla. Jedan primer je dat na slici pored.

Uređena binarna stabla odgovaraju sortiranim listama. Međutim, rad sa uređenim binarnim stablima je neuporedivo efikasniji zato što su brojevi u uređenom binarnom stablu lukavo raspoređeni.

Primer. Napisati funkciju `Member` koja utvrđuje da li se dati broj nalazi u uređenom binarnom stablu. Koristiti činjenicu da ako broj koga tražimo nije u korenu, onda treba pretražili samo jedno podstablo!

```
function Member(t : IntTree; n : integer) : Boolean;
begin
  if t = nil then
    Member := false
  else if t↑.info = n then
    Member := true
  else if n < t↑.info then
    Member := Member(t↑.left, n)
  else
    Member := Member(t↑.right, n)
end;
```

Primer. Napisati proceduru `Insert` koja uređenom binarnom stablu dodaje novi element, ali tako da dobijeno stablo i dalje ostane uređeno.

```
procedure Insert(var t : IntTree; n : integer);
begin
  if t = nil then begin
    new(t);
    t↑.info := n;
    t↑.left := nil;
    t↑.right := nil
  end
  else if n <= t↑.info then
    Insert(t↑.left, n)
  else
    Insert(t↑.right, n)
end;
```

Sada je jasno da se proces formiranja uređenog binarnog stabla svodi na niz poziva procedure `Insert`: krenemo od praznog stabla, učitavamo brojeve od korisnika i `Insert`-ujemo ih u stablo jedan po jedan.

Primer. Napisati funkciju `Min` koja određuje najmanji element nepraznog uređenog stabla. Koristiti činjenicu da se najmanji element u uređenom stablu nalazi u "donjem levom uglu" stabla, a do njega se dolazi tako što se samo "spustimo" niz pokazivače na levo podstablo.

```
function Min(t : IntTree) : integer;
{ pretpostavljamo da je t <> nil }
begin
  while t↑.left <> nil do
    t := t↑.left;
  Min := t↑.info
end;
```

Zadaci.

9.19. Napisati Pascal funkciju

```
function Max(t : IntTree) : integer;
```

koja određuje najveći element nepraznog uređenog stabla.

9.20. Napisati Pascal program koji sortira niz celih brojeva tako što redom učitava brojeve od korisnika, formira uređeno binarno stablo pozivom procedure `Insert`, a potom ispiše elemente tog stabla koristeći inorder obilazak.

9.21. Napisati Pascal funkciju

```
function IsOrdered(t : IntTree) : Boolean;
```

koja proverava da li je dato celobrojno stablo uređeno.

†**9.22.** Napisati Pascal proceduru

```
procedure Delete(var t : IntTree; el : IntTree);
```

koja iz uređenog binarnog stabla `t` izbacuje kućicu na koju pokazuje `el`, ali tako da nakon izbacivanja i dalje imamo uređeno binarno stablo.
(Napomena: broj iz kućice na koju pokazuje `el` treba zameniti najvećim brojem u desnom podstablu koje odgovara kućici `el`.)

9.23. Efikasnost algoritama za rad sa uređenim stablom zasniva se na tome da kućice budu "ravnomerno" raspoređene unutar stabla, odnosno, da stablo bude balansirano.

Neka je sa $n_L(A)$, odnosno $n_R(A)$, označen broj kućica u levom, odnosno desnom, podstablu koje odgovara kućici A. Za binarno stablo kažemo da je *balansirano* ako za svaku kućicu A u stablu važi

$$|n_L(A) - n_R(A)| \leqslant 1.$$

Drugim rečima, za svaku kućicu A u stablu, broj kućica u levom i desnom podstablu (koji odgovaraju kućici A) se razlikuju najviše za 1. Napisati Pascal funkciju

```
function IsBalanced(t : IntTree) : Boolean;
```

koja proverava da li je dato celobrojno stablo balansirano.

†**9.24.** Standardna strategija kojom se balansira binarno stablo (ako mu je balansiranje potrebno) sastoji se iz dve faze: prvo se od kućica binarnog stabla formira soritrana lista, a onda se od sortirane liste formira balansirano binarno stablo.

(a) Napisati Pascal proceduru

```
procedure TreeToVine(var t : IntTree);
```

koja od uređenog bianrnog stabla pravi sortiranu listu (engl. vine = vinova loza). Procedura radi tako što rekurzivnim pozivom levo podstablo konvertuje u sortiranu listu, desno podstablo konvertuje u sortiranu listu, i na kraju samo nadoveže "levu" listu, koren stabla i "desnu" listu. Liste formirati upotrebom pokazivača `right`.

(b) Napisati Pascal proceduru

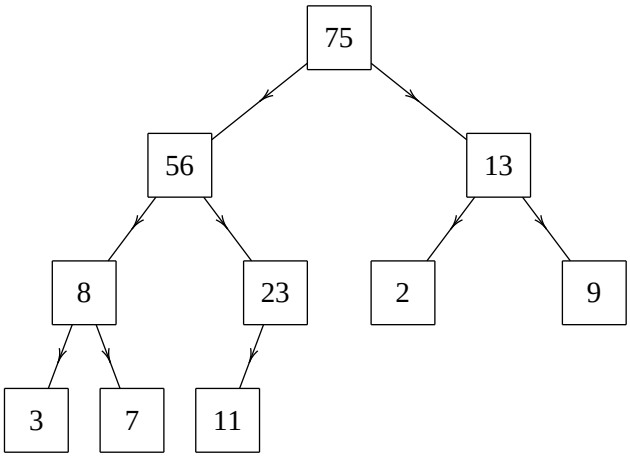
```
procedure VineToTree(var t : IntTree);
```

koja od sortirane liste ("vine") pravi uređeno binarno stablo. Procedura radi tako što sortiranu listu "prelomi" na pola, pa onda od "leve" polovine rekurzivnim pozivom napravi levo podstablo, a od "desne" polovine desno podstablo. Kućica "na sredini liste" će biti koren stabla.

9.3 Heap sort

Heap (engl. gomila, hrpa) je celobrojno binarno stablo kod koga važi sledeće:

- za svaki čvor `A` i za svakog njegovog potomka `B` je `B.info` $\leqslant$ `A.info`;
- visine listova ovog stabla se razlikuju najviše za 1, i pri tome se listovi sa većom visinom nalaze "levlje" od listova sa manjom visinom (drugim rečima, *heap* se popunjava nivo po nivo, svaki nivo se popunjava redom, sleva nadesno, i ne otvara se novi nivo dok se tekući nivo ne popuni do kraja).

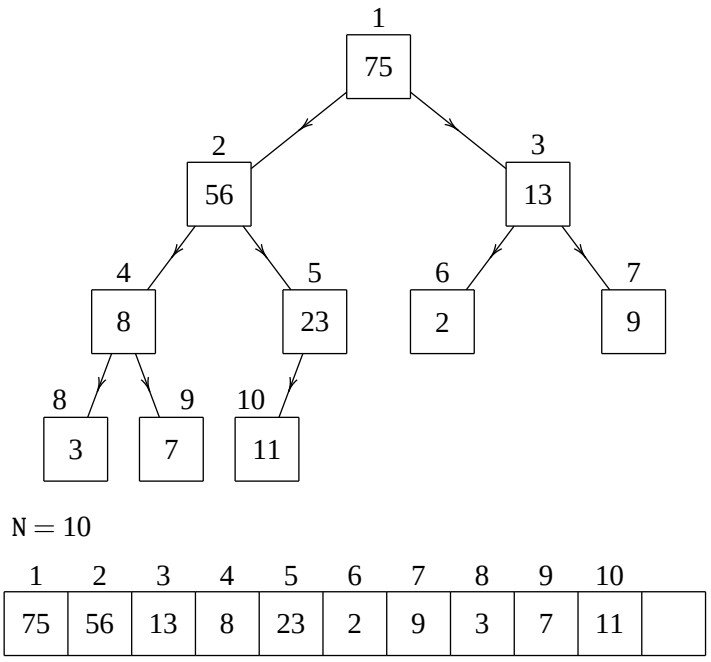


(Ovakav *heap* se još zove i *max-heap* jer se u korenu svakog podstabla nalazi maksimum tog podstabla; u slučaju da za svakog potomka `B` proizvoljnog čvora `A` važi `B.info` $\geqslant$ `A.info` dobijamo *heap* koji se zove *min-heap*. U ovom odeljku ćemo posmatrati samo *max-heap*ove.)

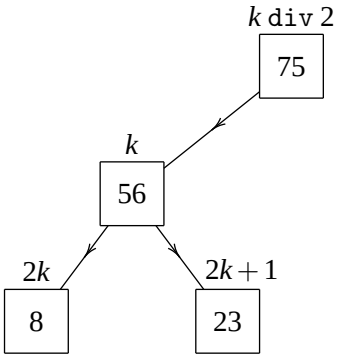
Drugi zahtev u definiciji *heap*a (da se *heap* popunjava redom, nivo po nivo) omogućuje da se *heap* implementira ne kao pokazivačka struktura već veoma jednostavo pomoću niza: *heap* smestimo u niz redom, nivo po nivo. Na primer, *heap* sa gornje slike ćemo predstaviti kao niz

```
const
  MaxN = 1000;
type
  IntArray = array[1 .. MaxN] of integer;
var
  A : IntArray;
  N : integer;
```

(pri čemu je `N` broj popunjenih članova niza `A`) ovako:



Struktura stabla može veoma lako rekonstruisati: otac čvora k je k `div` 2, njegov levi sledbenik je $2k$, a desni $2k+1$:

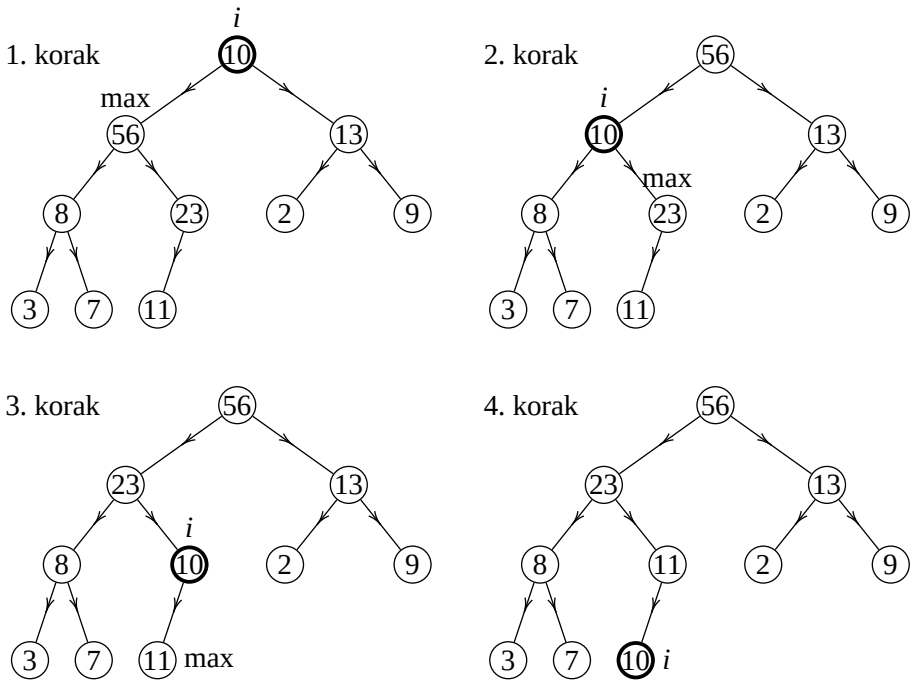


Pri tome, čvor k je u stablu ako i samo ako je $k \leqslant N$. Tako, na primer, levi sledbenik čvora 5 je u stablu ($2 \cdot 5 \leqslant 10$), dok desni nije ($2 \cdot 5 + 1 > 10$).

Osnovni metod za manipulisanje *heap*om je procedura `Heapify` koja ima zadatak da popravi stanje u *heap*u koje je narušeno u čvoru i. Pretpostavimo da vrednost u čvoru *i nije* maksimalna vrednost podstabla čiji je to koren, dok su njegovo levo i desno podstablo korektno formirani *heap*ovi. Stanje popravljamo propagacijom poremećaja naniže:

- posmatramo čvorove i, $2i$ i $2i+1$ (dakle, čvor u kome je poremećaj i negovog levog i desnog sina) i utvrdimo na kom od ta tri mesta se nalazi najveći broj;
- zamenimo sadržaj čvora i sa najvećom vrednosšću koju smo utvrdili u prethodnom koraku i tako poremećaj gurnemo jedan nivo dublje;
- nastavljamo sa ovim procesom sve dok ne dođemo do korektne situacije.

Jedan primer je pokazan na Sl. 9.10. U prvom koraku najveći od brojeva 10, 56, 13 je 56. Razmenimo 10 i 56, i nastavljamo razmatranjem korena levog podstabla. U drugom koraku najveći od brojeva 10, 8, 23 je 23. Razmenimo 10 i 23, i nastavljamo razmatranjem korena desnog podstabla. U trećem koraku najveći od brojeva 10, 11 je 11 (ovde desno podstablo ne postoji). Razmenimo 10 i 11, i nastavljamo razmatranjem korena levog podstabla. U četvrtom koraku konstatujemo da je ovo podstablo list, pa nema potrebe dalje bilo šta korigovati.



Slika 9.10: Primer rada procedure `Heapify`

```
procedure Heapify(var A : IntArray; N : integer; i : integer);
var
  max : integer;
  correct : boolean;
begin
  correct := false;
  while not correct do begin
    { od indeksa i, 2 * i i 2 * i + 1 odaberemo onaj koji }
    { indeksira najveću od vrednosti A[i], A[2 * i] i A[2 * i + 1] }
    max := i;
    if (2 * i <= N) and (A[2 * i] > A[i]) then max := 2 * i;
    if (2 * i + 1 <= N) and (A[2 * i + 1] > A[max]) then max := 2 * i + 1;
    if max = i then
      { postigli smo korektnu situaciju, algoritam se zaustavlja }
      correct := true
    else
      begin
        { maksimalnu vrednost postavljamo u koren }
        { i nastavljamo sa propagacijom naniže }
        Swap(A, i, max);
        i := max
      end
  end
end;
```

Procedura `MakeHeap` od proizvoljnog niza `A[1 .. N]` pravi *heap*. Primetimo da je za sve $k > N \text{ div } 2$ čvor `A[k]` zapravo koren trivijalnog stabla (jer je i $2k > N$ i $2k+1 > N$, što znači da `A[k]` u strukturi nema ni levog ni desnog sina) i zato za čvorove `A[N div 2 + 1 .. N]` nema potrebe raditi ništa: to su trivijalni *heap*ovi koji se sastoje samo od jednog čvora. Za ostale elemente niza može lako da se desi da je struktura *heap*a narušena, što se ispravlja pozivom procedure `Heapify`.

```
procedure MakeHeap(var A : IntArray; N : integer);
var
  i : integer;
begin
  for i := N div 2 downto 1 do Heapify(A, N, i)
end;
```

Konačno, procedura `HeapSort` sortira niz `A` na sledeći način. Prvo od niza napravi *heap* pozivom procedure `MakeHeap`. Tako smo postigli da se na poziciji `A[1]` nalazi najveći element niza. Njega razmenimo sa poslednjim elementom niza (što je donji desni list u stablu, i o kome ne znamo skoro ništa) i smanjimo `N` za jedan. Time smo postigli dve stvari:

- najveći element niza je na kraju niza i *heap* ga više ne vidi, pa algoritmi za manipulisanje *heapom* ne mogu da utiču na njega, i
- na poziciji `A[1]` se sada nalazi element za koga ne možemo da garantujemo da je najveći – to je poremećaj koga popravljamo pozivom procedure `Heapify`.

```
procedure HeapSort(var A : IntArray; N : integer);
begin
  MakeHeap(A, N);
  while N >= 2 do begin
    Swap(A, 1, N);
    dec(N);
    Heapify(A, N, 1);
  end
end;
```

Glava 10

Uvod u OOP

U ovoj glavi počinjemo *Sagu o objektno-orijentisanom programiranju* čiji cilj je se predstavi niz ideja koji vodi od samih početaka, pa sve do do objektno-orijentisanog programiranja (OOP), o kome ćemo govoriti u narednoj glavi.

Veliki deo *Sage* će biti posvećen pojmu *apstraktnog tip podataka* i *apstraktnom* programiranju uopšte. OOP je najapstraktniji od svih apstraktnih pogleda na programiranje. Zato je besmisleno pričati o OOP bez usvojenih osnovnih ideja apstraktnog programiranja.

Treba uočiti da ovaj pregled ni približno neće biti hronološki, jer (verovali ili ne), prvi jezik sa elementima OOP je nastao u Švedskoj i zvao se Simula-67. U to vreme nije doživeo neku veliku popularnost, ali je poslužio kao odskočna daska za Smalltalk-80 (Xerox, Palo Alto 1980), od koga je sve počelo.

No, pođimo redom (dakle, ne-redom, ne-hronološki).

10.1 Tip podataka, struktura podataka

Šta je to tip podataka, a šta struktura? U čemu je razlika između ta dva pojma?

> *Pre nego što odgovori na pitanja budu isporučeni, jedna napomena: ovde nećete naći* stroge *i* formalne *definicije pojmova. Samo ideje. Za one* **prave** *matematičke definicije je potrebno dati prilično neprijatan uvod u nekoliko komplikovanih teorija. O tome drugi put...*

Tip podataka je određen (konačnim) skupom vrednosti i nekim paketom operacija i relacija nad elementima tog skupa. Na primer,

```
({-32768, ..., -1, 0, 1, ..., 32767}, +, -, * div, mod, =, <)
```

je tip podataka u narodu poznatiji kao celobrojni tip (ili `integer`).

Struktura podataka je konglomerat podataka. Ona se formira od drugih (jednostavnijih) struktura i od tipova. Jedine operacije koje se nad strukturama mogu obavljati su operacije selekcije (i, kao posledica toga, pretraživanja). Posao operacija selekcije je da iz strukture izdvoje jedan jednostavniji podatak (jednu komponentu strukture), ili da nam omoguće pristup njoj.

Ovde treba razlikovati strukturu podataka od metoda strukturiranja podataka. Recimo, `array` je metod strukturiranja, a nakon deklaracije

```
var v: array [1 .. 100] of real;
```

promenljiva `v` postaje struktura. Svaki metod strukturiranja ima karakteristične operacije selekcije. Recimo, `array` kao operaciju selekcije ima indeksiranje, a `record` kao operaciju selekcije ima odabir polja sloga. Na primer, u Pascalu postoje dva eksplicitna metoda strukturiranja (`array` i `record`) i jedan implicitni (preko pokazivača).

Razlika između tipa i strukture je u tome što je nad elementima tipa moguće vršiti nekakve operacije i smeštati ih u nekakve relacije, dok je nad strukturom moguće obavljati samo selekciju komponente.

Tipovi se razlikuju po svom nosaču (tj. skupu osnovnih vrednosti) i operacijama koje se mogu obaviti nad elementima nosav ca, a strukture se razlikuju po operacijama selekcije, načinu smeštanja u memoriju (i, kao posledica toga, načinu pretraživanja strukture).

10.2 Dilema

I tek što nam se sjasnilo šta je to tip, a šta struktura, počinjemo da gledamo po programskim jezicima i uviđamo da granica između ova dva pojma nije uopšte jasna.

Na primer, u standardnom Pascalu su stringovi implementirani kao pakovani nizovi znakova, i tako predstavljaju strukturu. S druge strane, u raznim konkretnim implementacijama Pascala (uključujući FreePascal i TurboPascal) su stringovi implementirani kao osnovni tip podataka (jer se nad stringovima u FreePascalu, recimo, mogu obavljati razne operacije i stringovi se mogu stavljati u razne relacije: nadovezivanje, uzimanje potstringa, poređenje stringova...)

Prema ovome se čini da je stvar ostavljena onome ko dizjanira jezik. U jednom jeziku se nešto može javiti kao tip, a u drugom jeziku to isto kao struktura. I čini se da smo na taj način pomirili antagonizme. Ali nismo!

Šta se dešava kada u standardnom Pascalu napišemo paket procedura koje rade sa stringovima? Recimo, ovako:

```
type STR = packed array [1 .. 100] of char;
procedure Concat(s1, s2: STR; var res: STR); ...
procedure Substr(s: STR; from, to: integer; var res: STR); ...
function  Compare(s1, s2: STR): integer; ...
```

Nije li tako STR postao tip? Imamo skup vrednosti (svi nizovi karaktera dužine 100) i operacije na elementima tog skupa (`Concat`, `Substr`, `Compare`). Sve što treba da nešto bude tip?

Jeste. STR je tako postao tip. Dakle, STR je, po sebi, struktura koja je u vrednim rukama programera postala tip.

Naravoučenije: svaku strukturu možemo pretvoriti u tip ako joj damo ime i snabdemo je operacijama (koje rade nešto korisno, ali to nije obavezno).

10.3 Zašto je važno praviti nove tipove?

Zbog toga što je mnogo lakše raditi kada se apstrahuju nebitni detalji. (To su oni detalji koji nisu bitni u *tom* trenutku; svaki detalj je bitan, pre ili kasnije!)

Za uspešan rad u velikim timovima i na velikim projektima bitno je pisati apstraktan kōd. Zamislite projekt u kome je potrebno raditi nešto sa matricama. Najbolje je napraviti tip `MATRIX` (dakle, nekoj strukturi dati ime `MATRIX` i onda je snabdeti operacijama za rad sa matricama, recimo, sabiranje, množenje, invertovanje, ...), pa kada dođe do rešavanja konkretnog problema, samo pozivati gotove (i testirane) procedure.

Nije dobra politika pisati svaku operaciju iz početka i to baš tamo gde vam je zatrebala. To ne samo da uvećava kōd, nego i povećava mogućnost da dođe do greške. Osim toga, cilj modernih programskih jezika je u tome da tekst programa što vernije odslikava strukturu problema. Ako je potrebno u jednom trenutku uraditi neku vratolomiju sa matricama, idealno bi bilo da se može napisati nešto ovako:

```
C := Inv(A) * Transpose(B) + Det(Y) * Adj(X)
```

Nešto nepoželjnije, ali još uvek prihvatljivo je ovakvo parče kōda:

```
Inv(A, A1);
Transpose(B, B1);
MatMul(A1, B1, P);

Adj(X, X1);
ScalMul(Det(Y), X1, Q);

MatAdd(P, Q, C);
```

No, najružniji, najlošiji i najneprihvatljiviji način je onaj koji počinje ovako:

```
for i := 1 to n do
    for j := 1 to n do
        { rutina koja invertuje matricu A u A1 };
for i := 1 to n do
    for j := 1 to n do
        B1[i,j] := B[j, i];
for i := 1 to n do
    for j := 1 to n do
        { rutina koja mnozi A1 i B1 };
{ itd }
```

O ovom poslednjem ne treba ni pričati. Ono drugo može u svakom programskom novijem programskom jeziku, a ono prvo samo u nekim jezicima. Između ostalih, i u OO jezicima. Jasno je da je to izražajna moć koju želimo!

Ukoliko nemate mogućnost da koristite OO jezik, koristite bar ideje. Radite kao što je demonstrirano u drugom primeru. Tako će vaši programi biti razumljiviji, lakše će se testirati, i programiranje će teći mnogo prirodnije.

10.4 Apstraktno programiranje

Opšta metodologija programiranja koja se može okarakterisati kao *dobra* metodologija se ukratko može opisati ovako: treba dizajnirati odgovarajuće strukture podataka, napisati procedure za manipulaciju tim strukturama podataka, i kada dođe do ključnog trenutka, samo se pozivaju gotove procedure koje rade posao.

Tako dolazimo do jednog višeg nivoa apstrakcije u programima. U trenutku kada je potrebno primeniti neke operacije na nekim strukturama, ne interesuje nas *kako* procedure rade, već *šta* rade.

Primer. Pretpostavimo da imamo tip `MATRIX` i paket procedura koje barataju sa matricama. Kada kažemo:

```
var a, b: MATRIX;
begin
    NewMatrix(a); NewMatrix(b);
    ReadMatrix(a);
    Transpose(a, b);
    WriteMatrix(b);
    DisposeMatrix(a); DisposeMatrix(b)
end;
```

mi želimo da posao bude obavljen! Uopšte nas ne interesuje da li su matrice implementirane statički, recimo ovako:

```
type MATRIX = array [1 .. MaxX, 1 .. MaxY] of real;
```

ili su implementirane preko pokazivačkih struktura što može biti jako zgodno kada se radi sa retkim matricama:

```
type MATRIX = ↑MatrixEntry;
     MatrixEntry = record
             i, j       : integer;
             entry      : real;
             right, down: MATRIX
         end;
```

Kako su moguće razne implementacije koje rade isti posao, u trenutku upotrebe procedura nije bitna unutrašnja struktura već *ponašanje* procedura (koje je dato u specifikaciji paketa).

Nije bitna implementacija, već manifestacije (tj. osobine i ponašanje) operacija koje tip podataka podržava!

10.5 Apstraktni tip podataka

Videli kako struktura podataka snabdevena odgovarajućim operacijama postaje tip podataka.

Apstraktni tip podataka (Abstract Data Type, ADT) je tip podataka čiju implementaciju ne znamo (primer: ne znamo da li su matrice predstavljene kao `array` ili preko pokazivača), ali znamo kako se ponašaju operacije nad vrednostima tog tipa, tako da pišemo program koristeći samo osobine operacija.

Jasno je zašto se to zove tip podataka (zato što je to upravo tip podataka). Dodat mu je atribut “apstraktni” zato što korisnik ne poznaje konkretnu implementaciju. Korisniku su poznata samo apstraktna svojstva operacija nad tim tipom (drugim rečima, poznato je šta koja operacija radi, ali ne i kako to radi). Dakle, apstrahovana je jedna dimenzija problema: implementacija tipa.

Primer 1. Stek sa celobrojnim elementima kao apstraktni tip podataka.

Specifikacija:

Tip: `IntStack`

`Open(s)`	inicijalizuje stek `s`; mora se pozvati pre prve upotrebe promenljive `s`
`Close(s)`	deinicijalizuje stek `s`; mora se pozvati kada `s` više nije potreban i više se neće koristiti
`Push(s, n, ok)`	na stek `s` stavi broj `n` i vrati `ok = true`; ako to nije moguće, vrati `ok = false`
`Pop(s, n, ok)`	sa steka `s` skine broj `n` i vrati `ok = true`; ako to nije moguće, vrati `ok = false`
`Full(s)`	vrati `true` ako na `s` više nema mesta
`Empty(s)`	vrati `true` ako je `s` prazan

Implementacija:

```
type IntStack = ↑StackEntry;
     StackEntry = record
         entry: integer;
         link : IntStack
     end;

procedure Open(var s: IntStack);
begin
    s := nil
end;

procedure Close(var s: IntStack);
var
    t: IntStack;
begin
    while s <> nil do begin
        t := s;
        s := s↑.link;
        dispose(t)
    end;
    s := nil
end;
```

```
procedure Push(var s : IntStack; n: integer; var OK: Boolean);
var
    t: IntStack;
begin
    new(t);
    if t = nil then
        OK := false
    else begin
        t↑.entry := n;
        t↑.link := s;
        s := t;
        OK := true
    end
end;

procedure Pop(var s : IntStack; var n: integer; var OK: Boolean);
var
    t: IntStack;
begin
    if s = nil then
        OK := false
    else begin
        t := s;
        s := s↑.link;
        n := t↑.entry;
        OK := true;
        dispose(t)
    end
end;

function Full(s: IntStack): Boolean;
begin
    Full := false
    {
      ovde moze da se navede i poziv neke sistemske rutine koja
      ce proveriti da li ima dovoljno mesta za alokaciju
     }
end;

function Empty(s: IntStack): Boolean;
begin
    Empty := s = nil
end;
```

Primer 2. Stek sa celobrojnim elementima kao apstraktni tip podataka.

Specifikacija:

Tip: `IntStack`

`Open(s)`	inicijalizuje stek `s`; mora se pozvati pre prve upotrebe promenljive `s`
`Close(s)`	deinicijalizuje stek `s`; mora se pozvati kada `s` više nije potreban i više se neće koristiti
`Push(s, n, ok)`	na stek `s` stavi broj `n` i vrati `ok = true`; ako to nije moguće, vrati `ok = false`
`Pop(s, n, ok)`	sa steka `s` skine broj `n` i vrati `ok = true`; ako to nije moguće, vrati `ok = false`
`Full(s)`	vrati `true` ako na `s` više nema mesta
`Empty(s)`	vrati `true` ako je `s` prazan

Implementacija:

```
const MaxStack = 100;
type  IntStack = record
          top  : integer;
          entry: array [1 .. MaxStack] of integer
      end;

procedure Open(var s: IntStack);
begin
    s.top := 0
end;

procedure Close(var s: IntStack);
begin
    s.top := 0
end;

procedure Push(var s : IntStack; n: integer; var OK: Boolean);
begin
    if s.top = MaxStack then
        OK := false
    else begin
        s.top := s.top + 1;
        s.entry[s.top] := n;
        OK := true
    end
end;
```

```
procedure Pop(var s : IntStack; var n: integer; var OK: Boolean);
begin
    if s.top = 0 then
        OK := false
    else begin
        n := s.entry[s.top];
        s.top := s.top - 1
    end
end;

function Full(s: IntStack): Boolean;
begin
    Full := s.top = MaxStack
end;

function Empty(s: IntStack): Boolean;
begin
    Empty := s.top = 0
end;
```

Koja je razlika između Primera 1 i Primera 2? Samo u implementaciji tipa! Korisnik ne vidi nikakvu razliku! Ako ima pred sobom specifikaciju

Tip: `IntStack`

`Open(s)`	inicijalizuje stek `s`; mora se pozvati pre prve upotrebe promenljive `s`
`Close(s)`	deinicijalizuje stek `s`; mora se pozvati kada `s` više nije potreban i više se neće koristiti
`Push(s, n, ok)`	na stek `s` stavi broj `n` i vrati `ok = true`; ako to nije moguće, vrati `ok = false`
`Pop(s, n, ok)`	sa steka `s` skine broj `n` i vrati `ok = true`; ako to nije moguće, vrati `ok = false`
`Full(s)`	vrati `true` ako na `s` više nema mesta
`Empty(s)`	vrati `true` ako je `s` prazan

korisniku je nebitno šta se dešava u pozadini. Njegov program koji koristi stek će raditi savršeno bez obzira na konkretne procedure koje rade u pozadini. Štaviše, ako je prvo koristio paket iz Primera 1, a onda bio primoran da koristi paket iz Primera 2, njegov program se neće promeniti ni za liniju!

Tajna moći ADT leži u tome da su svi apstraktni tipovi koji imaju istu specifikaciju *izomorfni* (u algebarskom smislu). Ideja izomorfizma je upravo naša mantra:

izomorfne strukture u suštini predstavljaju jednu istu stvar; razlika je jedino u načinu realizacije te jedne stvari!

10.6 Zašto je dobro raditi sa ADT

Upotreba ADT ima nekoliko prednosti nad običnim (jako konkretnim) programiranjem. Prvo, programer ne mora da razmišlja o implementaciji jedne stvari dok implementira drugu. Tako su njegovi moždani talasi fokusirani samo na jedno. Implementaciju pomoćnih objekata slobodno može da zaboravi i da ih koristi znajući samo njihove apstraktne osobine.

Drugo, na taj način se jednostavnije radi timski. Svaki član tima dobije posao da implementira jedan ADT. On to uradi, spakuje u modul i podeli ostalima u timu samo specifikaciju svog ADT. Ako se ostali u timu strogo pridržavaju specifikacije, implementator garantuje da će sve da radi. Takođe, ostali ne moraju da se opterećuju detaljima o implementaciji tuđeg ADT, jer imaju i svog posla preko glave.

Treće, kada onaj ko je implementirao ADT nađe bolje i brže algoritme da uradi svoj posao, slobodno može da menja kōd po svom modulu, da doteruje i ubrzava, sve dotle dok ne menja specifikaciju. Izmene koje on vrši uopšte ne utiču na ostale u timu (jedino mogu biti prijatno iznenađeni kada vide da njihov program odjednom radi brže).

10.7 ADT i modularna struktura jezika

Modul u programskom jeziku je deo programa koji sadrže paket procedura i funkcija koji predstavlja logičnu celinu. Moduli se nalaze u posebnim datotekama i prevode se nazavisno jedan od drugog, i nezavisno od glavnog programa koji ih koristi. Veliki programi se najčešće sastoje iz modula, jer se tako omogućuje razbijanje programa na manje celine koje se nezavisno razvijaju i testiraju.

Standardni Pascal nema podršku za modularno programiranje, i to mu je jedan od osnovnih nedostataka. Novije verzije Pascala, uključujući i FreePascal, imaju takve konstrukcije. Modul se u FreePascalu zove *unit* (engl. unit = jedinica prevođenja; ime potiče odatle što se unit prevodi nezavisno od ostatka programa) i ima sledeći strukturu:

```
unit ⟨ime-unita⟩;
interface
    ⟨deklaracije javnih konstanti⟩;
    ⟨deklaracije javnih tipova⟩;
    ⟨deklaracije javnih promenljivih⟩;
```

```
    ⟨deklaracije javnih procedura i funkcija⟩;
implementation
    ⟨deklaracije privatnih konstanti⟩;
    ⟨deklaracije privatnih tipova⟩;
    ⟨deklaracije privatnih promenljivih⟩;
    ⟨implementacije javnih procedura i funkcija⟩;
    ⟨implementacije privatnih procedura i funkcija⟩;
end.
```

Deo koji je označen sa `interface` sadrži deklaracije konstanti, tipova, promenljivih, kao i zaglavlja funkcija i procedura koji su *javni*, što znači da su to podaci koje modul nudi korisnicima. Deo označen sa `implementation` je privatni deo modula i njega vidi samo deo tima koji implementira modul. Konstante, tipovi i promenljive koje su tu deklarisane, kao i procedure i funkcije koje se tu nalaze, ali nisu najavljene u interfejsu se ne vide izvan modula i njima se ne može pristupiti spolja.

ADT se jako lepo slaže sa modularnom strukturom jezika. Uglavnom se jedan ADT implementira kao jedan modul, interfejs modula se razdeli potencijalnim korisnicima kao specifikacija tipa, a u implementacionom delu se tip realizuje. Na primer, apstraktni tip `IntStack` koji implementira celobrojni stek se u FreePascalu može realizovati ovako:

```
unit IntStackUnit;

interface

    type IntStack = ↑Elem;
         Elem = record
             entry: integer;
             link: IntStack
         end;

    procedure Open(var s: IntStack);
    procedure Close(var s: IntStack);
    procedure Push(var s: IntStack; n: integer; var OK: Boolean);
    procedure Pop(var s: IntStack; var n: integer; var OK: Boolean);
    function Full(s: IntStack): Boolean;
    function Empty(s: IntStack): Boolean;

implementation

    procedure Open(var s: IntStack);
    begin
      s := nil;
```

```
end;

procedure Close(var s: IntStack);
var
  t : IntStack;
begin
  while s <> nil do begin
    t := s;
    s := s↑.link;
    dispose(t)
  end;
  s := nil
end;

procedure Push(var s: IntStack; n: integer; var OK: Boolean);
var
  t : IntStack;
begin
    new(t);
    if t = nil then
        OK := false
    else begin
        t↑.entry := n;
        t↑.link := s;
        s := t;
        OK := true
    end
end;

procedure Pop(var s: IntStack; var n: integer; var OK: Boolean);
var
  t : IntStack;
begin
    if s = nil then
        OK := false
    else begin
        t := s;
        s := s↑.link;
        n := t↑.entry;
        OK := true;
        dispose(t)
    end
end;
```

```
    function Full(s: IntStack): Boolean;
    begin
      Full := false
    end;

    function Empty(s: IntStack): Boolean;
    begin
      Empty := s = nil;
    end;

end.
```

10.8 Sakrivanje informacija

Efikasna upotreba apstraktnih tipova podataka se bazira na korišćenju specifikacije, bez virenja u implementaciju tipa. Ako korisnik makar malko čačne po implementaciji, cela ideja propada.

Praksa je pokazala da kaogod što je decu nemoguće odvići od čačkanja nosa, tako je programere nemoguće odvići od čačkanja po implementaciji tipa. Uvek se nađe neki mračni kutak svesti iz koga iskoči ideja:

> *"A zašto bih ja ovu trivijalnu stvar radio pozivom procedure (i tako gubio vreme), kada je mnogo brže reći*
>
> `a.struct[x]↑.left := nil`
>
> *Sada ću ja to časkom, da niko ne vidi!"*

Zato je za pojam ADT neraskidivo vezan pojam sakrivanja informacije (*information hiding*). Programeru se stavi na raspolaganje ime tipa i paket procedura. Implementacija tipa se *sakrije*. Čak i ako poželi da čeprka po strukturi, jezik treba to da mu zabrani.

Implementacija celobrojnog steka koju smo upravo videli ne sakriva internu strukturu tipa. Programer može da pristupi poljima sloga i da direktno menja njihov sadržaj. Srećom, FreePascal nudi mogućnost da se to spreči. Interfejs unita treba napisati ovako:

```
unit IntStackUnit;

interface

    type IntStack = ↑Elem;
         Elem = object
           private
               entry: integer;
               link: IntStack
         end;

    procedure Open(var s: IntStack);
    procedure Close(var s: IntStack);
    procedure Push(var s: IntStack; n: integer; var OK: Boolean);
    procedure Pop(var s: IntStack; var n: integer; var OK: Boolean);
    function Full(s: IntStack): Boolean;
    function Empty(s: IntStack): Boolean;

implementation

    { isto kao ranije }

end.
```

O konstrukciji `object` ćemo pisati detaljnije. Za sada je dovoljno reći da se radi o varijanti sloga (`record`) koji ima mogućnost da delove svoje strukture koji su označeni kao `private` sakrije. Potencijalno zločesti korisnik i dalje vidi strukturu sloga, ali ne može da pristupi direktno poljima sloga. Jedini način da koristi stek se svodi na pozive procedura i funkcija koje unit nudi.

10.9 Intermezzo

Do sada smo videli šta je tip podataka, šta je struktura podataka i kako se od strukture napravi tip. Videli smo kako iz toga naraste ideja ADT i dve osnovne stvari koje se vezuju za ADT:

(1) apstrahovanje (*data abstraction*)

(2) skrivanje podataka (*information hiding*)

Takođe je napomenuto da se ADT jako lepo slaže sa modularnim jezicima. To ne znači da se ideje ADT ne mogu koristiti pri radu sa programskim jezikom koji nema mehanizam modula i nema mehanizam za skrivanje (npr, standardni Pascal). Pri radu sa takvim jezicima treba pokazati više samodiscipline. Ideja ADT je *osnovna* ideja OOP.

10.10 Objekti, klase, metodi i poruke

Objekt je paket podataka koji ima svoj tajni život. Možemo ga zamisliti kao jednu skupinu podataka i procedura za manipulaciju tim podacima. Procedure koje su deo objekta se zovu *metodi* (to su metodi za menjanje stanja objekta), dok podaci opisuju stanje objekta.

Klasa se može shvatiti kao opis jedne vrste objekata koji sadrži ponašanje svih objekata koji pripadaju toj klasi. Objekti koji pripadaju nekoj klasi zovu se *instance* te klase, a klasa opisuje koja polja i koje metode će imati njene instance.

Kao primer navodimo opis matrice kao klase čiji objekti mogu da urade četiri stvari: da se inicijalizuju, da se samoubiju, da se učitaju i da se ispišu. U FreePascalu klase se uvode kao novi tipovi (što, na kraju krajeva, klase i jesu):

```
const MaxN = 50;

type
    MATRIX = object
        private
            entry : array [1 .. MaxN, 1 .. MaxN] of real;
            M, N : integer;
        public
            procedure init(p, q : integer);
            procedure init(p : integer);
            procedure free;
            procedure input;
            procedure print;
    end;
```

Ova deklaracija znači da je `MATRIX` klasa čije instance će imati polja `M`, `N` i `entry` i metode `init`, još jedan `init` (o tome kasnije!), `free`, `input` i `print`.

Konstrukcija `object` se ponaša kao `record`, s tim što može da okuplja ne samo polja raznih tipova, već i deklaracije procedura i funkcija. Polja `M`, `N` i `entry` opisuju unutrašnje stanje objekta (dimenzije matrice i njene elemente), dok procedure `init`, `init`, `free`, `input` i `print` opisuju metode koji se mogu izvršiti nad instancama klase `MATRIX`. Osim toga, objekt vrši i striktnu kontrolu privatnosti. U skladu sa preporukama o sakrivanju informacija, polja `M`, `N` i `entry` su deklarisana kao privatna i njima mogu da pristupe samo metodi koji manipulišu nad objektom. Metodi `init`, `init`, `free`, `input` i `print` su javno dostupni i njima se može menjati stanje objekta.

Nakon definicije klase sledi implementacija metoda:

```
procedure MATRIX.init(p, q : integer);
begin
  if (1 <= p) and (p <= MaxN) and (1 <= q) and (q <= MaxN) then
    begin
      M := p;
      N := q
    end
  else writeln('MATRIX.init failed')
end;

procedure MATRIX.init(p, q : integer);
begin
  if (1 <= p) and (p <= MaxN) then
    begin
      M := p;
      N := p
    end
  else writeln('MATRIX.init failed')
end;

procedure MATRIX.free;
begin
  M := 0; N := 0
end;

procedure MATRIX.input;
var
  i, j : integer;
begin
  for i := 1 to M do
    for j := 1 to N do
      readln(entry[i, j])
end;

procedure MATRIX.print;
var
  i, j : integer;
begin
  for i := 1 to M do
    for j := 1 to N do
      writeln(i:3, j:3, '->', entry[i, j]:10:2)
end;
```

Metodi nisu obične procedure i funkcije; oni su vezani za klase. To se vidi iz imena koje ima oblik ⟨*klasa*⟩.⟨*metod*⟩, čime se stavlja do znanja da sledi implementacija metoda koji pripada navedenoj klasi. Primetimo da se polja `M`, `N` i `entry` sasvim lepo vide u telu odgovarajućih metoda. To je jedino mesto u programu gde se ova imena vide! Objekt klase `MATRIX` se sada deklariše kao obična promenljiva:

```
var A : MATRIX;
```

Poruka je nešto kao poziv procedure. Ona nema značenje dok se ne uputi nekom objektu. To je samo niz znakova. Kada se objektu uputi poruka, on među svojim metodima potraži onaj koji ima isto ime i signaturu (signatura je struktura argumenata) kao što se zahteva u poruci. Ako ne uspe, objekt odbaci poruku i tako izazove grešku, ili nešto slično. Ako objekt uspe da nađe odgovarajući metod među metodima kojima raspolaže, primeni taj metod na sebi, što može da dovede do promene stanja objekta (izmene se vrednosti nekih promenljivih) i do upućivanja novih poruka novim objektima.

Vidimo, dakle, da se poruka interpretira tako što objekt pokuša da pronađe metod koji ima isto ime i *signaturu*. Č injenica da struktura argumenata utiče na interpretiranje poruke kao posledicu ima to da je moguće deklarisati metode sa istim imenom, ukoliko oni imaju različite signature.

U prethodnom primeru, imamo dva metoda `init`: jedan koji prima dva celobrojna argumenta, i on se koristi za inicijalizaciju proizvoljnih matrica, a drugi sa jednim celobrojnim argumentom koga možemo da koristimo za inicijalizaciju kvadratnih matrica. Ako je `A` instanca klase `MATRIX` kao ranije, tada

```
A.init(9, 10);
```

predstavlja poruku `init(9, 10)` upućenu objektu `A`. Kako objekt poseduje metod koji se zove `init` i prima dva argumenta tipa `integer`, to će objekt interpretirati poruku na odgovarajući način, tj. aktiviraće metod `init` sa argumentima `9` i `10`. Aktivacija metoda `init` će dovesti do promene stanja objekta: biće popunjene vrednosti za `M` i `N`. S druge strane, poruka

```
A.init(16);
```

će biti interpretirana tako što će objekt aktivirati metod `init` sa jednim celobrojnim argumentom `16` i tako inicijalizovati kvadratnu matricu formata 16×16. Ova osobina se zove *operator overloading*.

10.11 Učaureni podaci

Svaki objekt koji drži do sebe mora svoju unutrašnjost držati daleko od prljavih prstiju okolnih korisnika (ili, *klijenata*). Zato onaj deo objekta koji opisuje strukturu promenljivih koje čuvaju stanje objekta ne sme biti dostupan klijentima.

Ako neko želi da proveri i/ili da promeni stanje objekta, mora to učiniti upućivanjem poruke objektu. Na taj način objekt ima potpunu kontrolu nad samim sobom i može da štiti svoj integritet (u krajnjem slučaju, metod koji proverava/menja stanje objekta može na neki način proveriti ko je poslao poruku i, ukoliko pošiljalac nije klijent od poverenja, može odbiti da otkrije/izmeni svoje stanje).

To je u skladu sa opštom idejom o skrivanju podataka (*information hiding*) i zove se još i *encapsulation* (učaurenost). Podaci su učaureni u objektu i do njih se ne može doći lako.

10.12 Polimorfizam

Različite klase mogu imati metode sa istim imenom. Ako uputimo poruku objektu jedne klase, on će je interpretirati koristeći svoje metode. No, tu istu poruku objekt druge klase može interpretirati na sasvim drugi način, jer će on koristiti svoje metode (metodi se zovu isto, ali rade sasvim drugačije stvari). Ova osobina se zove *polimorfizam*.

Posmatrajmo klase `TraficLight` (koja opisuje semafor), i `PrimeGen` (koja opisuje generatore prostih brojeva):

```
{--------------- TrafficLight ---------------}
type
    TrafficLight = object
        private
            light : (red, redAndYel, yellow, green);
        public
            procedure init;
            procedure next;
            procedure print;
    end;

procedure TrafficLight.init;
begin
    light := red
end;

procedure TrafficLight.next;
begin
    case light of
    red       : light := redAndYel;
    redAndYel : light := green;
    yellow    : light := red;
    green     : light := yellow
    end
end;
```

```
procedure TrafficLight.print;
begin
    case light of
    red       : writeln('Crveno');
    redAndYel : writeln('Crveno i zuto');
    yellow    : writeln('Zuto');
    green     : writeln('Zeleno')
    end
end;

{--------------- PrimeGen ---------------}
type
    PrimeGen = object
        private
            p : integer;
        public
            procedure init;
            procedure next;
            procedure print;
    end;

procedure PrimeGen.init;
begin
    p := 2;
end;


function IsPrime(n : integer) : Boolean;
{ n je neparan broj >= 3 }
var
  pr : Boolean;
  d, L : integer;
begin
    pr := true;
    d := 3;
    L := trunc(sqrt(n));
    while pr and (d <= L) do begin
        pr := n mod d <> 0;
        d := d + 2
    end;
    IsPrime := pr
end;
```

```
procedure PrimeGen.next;
begin
    if p = 2 then p := 3
    else
        repeat p := p + 2
        until IsPrime(p)
end;

procedure PrimeGen.print;
begin
  writeln(p)
end;
```

Uočimo da obe klase imaju po tri metoda: `init`, `next`, i `print`. (Funkcija `IsPrime` nije metod klase `PrimeGen`; to je obična Pascal funkcija koju koristi metod `next`.) Neka su `TL` i `PG` instance ove dve klase:

```
var
  TL : TrafficLight;
  PG : PrimeGen;
```

Kada kažemo

```
TL.init; PG.init;
```

jednu istu poruku smo uputili i jednom i drugom objektu. Razlika je u tome što će poruka `init` u prvom pozivu aktivirati metod `init` iz klase `TrafficLight`, dok će poruka `init` u drugom pozivu aktivirati metod `init` iz klase `PrimeGen`. Zato, nakon

```
TL.init;
TL.next;
TL.next;
TL.next;
TL.print;
```

dobijamo poruku `Zuto` na monitoru, dok nakon

```
PG.init;
PG.next;
PG.next;
PG.next;
PG.print;
```

dobijamo broj 7 na monitoru. Ova dva objekta su na iste poruke reagovali različito, jer su ih interpretirali svako na svoj način. Jedna ista poruka upućena različitim objektima dovodi do različitog ponašanja.

10.13 Prvi stepenik objektnog programiranja: programiranje bazirano na klasama

Ako neki programski jezik podržava

- klase,
- objekte kao instance klasa,
- učaurenost,
- mehanizam poruka i
- polimorfizam,

dobijamo prvi model objektnog programiranja: *programiranje bazirano na klasama* (engl. class-based programming). Ovaj model je jednostavniji od objektno-orijentisanog modela, o kome ćemo detaljno govoriti u nastavku. Iskreno se nadamo da žar u srcima još nije zgasnuo usled nestrpljenja.

10.14 Privatni život klase

Klasa opisuje strukturu i ponašanje svojih instanci. No, ima programskih jezika u kojima i klasa može imati svoj privatni život. Drugim rečima, i klasa može imati svoje promenljive i svoje metode.

Promenljive koje pripadaju klasi (*class variables*) služe za to da klasa drži neke svoje informacije (recimo, koliko instanci te klase je do sada napravljeno). Te informacije nisu nikome dostupne direktno; one se mogu koristiti isključivo putem poruka upućenih klasi. Poruke upućene klasi se interpretiraju na isti način kao poruke upućene objektu. Jedina razlika je u tome što poruke upućene klasi traže metode iz skupa privatnih metoda klase (*class methods*).

Smalltalk, recimo, ima malo drugačiji pristup. Tu su promenljive koje pripadaju klasi u stvari globalne promenljive koje koriste instance te klase i to za međusobnu komunikaciju. Promenljive koje pripadaju klasi u Smalltalk-u su dostupne samo instancama te klase i metodima klase.

U Javi se metodi klase i promenljive koje pripadaju klasi zovu *statičke promenljive/metodi* i ponašaju se kao promenljive i procedure običnog modula. I promenljive i medoti klase u Javi mogu biti deklarisani kao `public` ili `private`. Elementima koji su deklarisani kao `public` svako može da pristupi. U FreePascalu klasa može imati metode, ali ne može da ima svoje promenljive.

Još uvek ne postoji opšteprihvaćeno mišljenje da li klasa treba da ima pravo na intimni život ili ne; da li može da ima svoju privatnost, ili treba samo da služi svojim instancama.

Glava 11

Objektno-orijentisano programiranje

U ovoj glavi konačno stižemo do pojma OOP. Saga sada počinje da liči na pravu sapunsku operu jer ćemo početi da pričamo o nasleđivanju. Srećna okolnost je to što u OOP niko ne mora da umre da bi do nasleđivanja moglo da dođe.

Osnovna odlika objektno-orijentisanih sistema je nasleđivanje. *Nasleđivanje* je relacija među klasama koja omogućuje da se definicija i implementacija jedne klase bazira na definiciji i implementaciji neke već postojeće klase. Na taj način se štedi vreme, novac, ali i memorijski prostor.

Prilikom nasleđivanja *potklasa* nasleđuje i metode i strukturu objekta od svoje *nadklase*. Činjenica da potklasa nasleđuje metode znači da njih ne treba pisati ponovo. Treba dopisati samo nove stvari, kao i one koje se razlikuju. Ova osobina se zove *code reuse* (jer potklasa koristi kōd iz nadklase).

11.1 Drugi način da napravimo klase

Neka je data klasa `Realmat` koja opisuje realne matrice:

```
const
  MaxN = 50;

type
  RealMat = class
    private
      entry : array [1 .. MaxN, 1 .. MaxN] of real;
      M, N : integer;
    public
      constructor init(p, q : integer);
      function dim1 : integer;
      function dim2 : integer;
      procedure input;
      procedure print;
      function at(x, y : integer) : real;
      procedure put(x, y : integer; val : real);
      function add(q : RealMat) : RealMat;
      function mul(q : RealMat) : RealMat;
      function mul(x : real) : RealMat;
    end;
```

Pri deklaraciji klase `RealMat` umesto FreePascal konstrukcije `object` koristili smo FreePascal konstrukciju `class`. Razlika između ove dve konstrukcije je suptilna i o njoj ćemo govoriti kasnije. Recimo za sada samo to da sve što smo do sada rekli za konstrukciju `object` važi i za konstrukciju `class`, a da konstrukcija `class` omogućuje da se nasleđivanje implementira veoma jednostavno. Zato ćemo `class` koristiti kadgod treba da radimo sa punokrvnim objektno-orijentisanim modelom (koji podrazumeva nasleđivanje), dok ćemo `object` koristiti kada radimo sa *class-based* modelom o kome smo govorili u prethodnoj glavi.

Kao i do sada, polja `entry`, `M` i `N` su privatna i njima može da pristupi samo neki od metoda koji su navedeni u klasi. Osim toga, klasa nudi i niz javnih metoda za manipulisanje instancama ove klase. Metod `init` je specijalan metod koji se koristi za pravljenje instanci klase. Takvi metodi se zovu *konstruktori*.

Nakon deklaracije klase sledi implementacija metoda. Iako to na osnovu deklaracije nije jasno, konstruktor `init` je zapravo funkcija koja vraća objekt tipa `RealMat`, što će postati jasnije kada budemo pokazali kako se on koristi.

```
{ poseban metod cijim pozivom se konstruisu instance klase RealMat }

constructor RealMat.init(p, q : integer);
begin
  if (1 <= p) and (p <= MaxN) and (1 <= q) and (q <= MaxN) then
    begin
      M := p;
      N := q
    end
  else writeln('RealMat.init failed')
end;

{ metodi koji vraćaju prvu i drugu dimenziju matrice }

function RealMat.dim1 : integer;
begin
  dim1 := M
end;

function RealMat.dim2 : integer;
begin
  dim2 := N
end;

{ metodi za ucitavanje i ispis matrice }

procedure RealMat.input;
var
  i, j : integer;
begin
  for i := 1 to M do
    for j := 1 to N do
      readln(entry[i, j])
end;

procedure RealMat.print;
var
  i, j : integer;
begin
  for i := 1 to M do
    for j := 1 to N do
      writeln(i:3, j:3, '->', entry[i, j]:10:2)
end;
```

```
{ metod vraca element matrice na poziciji (x, y) }

function RealMat.at(x, y : integer) : real;
begin
  if (1 <= x) and (x <= M) and (1 <= y) and (y <= N) then
    at := entry[x, y]
  else begin
    writeln('RealMat.at failed');
    at := 0
  end
end;

{ na poziciju (x, y) u matrici metod postavlja vrednost val  }

procedure RealMat.put(x, y : integer; val : real);
begin
  if (1 <= x) and (x <= M) and (1 <= y) and (y <= N) then
    entry[x, y] := val
  else
    writeln('RealMat.put failed')
end;

{ metod sabira matricu kojoj je poruka upucena sa matricom q }

function RealMat.add(q : RealMat) : RealMat;
var
  res : RealMat;
  i, j : integer;
begin
  if (M = q.dim1) and (N = q.dim2) then begin
    res := RealMat.init(M, N);
    for i := 1 to M do
      for j := 1 to N do
        res.put(i, j, entry[i, j] + q.at(i, j));
    add := res
  end
  else begin
    writeln('RealMat.add failed');
    add := nil
  end
end;
```

```
{ metod mnozi matricu kojoj je poruka upucena matricom q }

function RealMat.mul(q : RealMat) : RealMat;
var
  res : RealMat;
  pom : real;
  i, j, k : integer;
begin
  if N = q.dim1 then begin
    res := RealMat.init(M, q.dim2);
    for i := 1 to M do
      for j := 1 to q.dim2 do begin
        pom := 0;
        for k := 1 to N do
          pom := pom + entry[i, k] * q.at(k, j);
        res.put(i, j, pom)
      end;
    mul := res
  end
  else begin
    writeln('RealMat.mul failed');
    mul := nil
  end
end;

{ metod mnozi matricu kojoj je poruka upucena realnim brojem }

function RealMat.mul(x : real) : RealMat;
var
  res : RealMat;
  i, j : integer;
begin
  res := RealMat.init(M, N);
  for i := 1 to M do
    for j := 1 to N do
      res.put(i, j, entry[i, j] * x);
  mul := res
end;
```

Nakon deklaracija:

```
    var a, b: RealMat;
```

a i b su instance klase RealMat. To znači da oba imaju polja entry, M, i N (koja su privatna stvar) i imaju metode init, dim1, dim2, input, print, at, put, mul,

`mul` i `add`. Evo i primera koji učitava matricu `a` formata 10×10, postavlja `b` na $2a^2$ i potom ispisuje `b[3,3]`:

```
begin
    a := RealMat.init(10, 10);
    a.read;
    b := a.mul(a).mul(2.0);    { tj. b := a * a * 2.0 }
    writeln(b.at(3,3))         { ispise b[3,3] }
end.
```

Primetimo da se konstruktor `init` poziva tako što se *klasi* `RealMat` uputi poruka `init(10, 10)`; klasa formira odgovarajući objekt i dodeli ga promenljivoj `a`. Kada radimo sa konstrukcijom `class`, dakle, konstruktori su metodi pridruženi klasi!

11.2 Nasleđivanje

Želeli bismo sada da naravimo klasu `SqRealMat` koja će predstavljati kvadratne realne matrice. Jedan način je da uzmemo klasu `RealMat` i da je malo preradimo. Na taj način dobijamo dve klase kod kojih je 90% kōda identično. Gubimo vreme i memorijski prostor. Zar ne bi bilo lepo kada bismo mogli da kažemo da je klasa `SqRealMat` "veoma slična" sa klasom `RealMat`, pa da dopišemo samo razliku?

O, da. Ne samo da bi bilo lepo, nego je to i moguće! Mehanizam koji nam to omogućuje zove se *nasleđivanje* (engl. inheritance). Napravićemo novu klasu i, umesto da prekucavamo kōd iz klase `RealMat`, reći ćemo da nova klasa nasleđuje sve osobine klase `RealMat`, pa ćemo samo dopisati nove (specifične) metode i redefinisati neke stare. Recimo, ovako:

```
type
  SqRealMat = class(RealMat)
    public
      constructor init(p : integer);
      function det : real;
      function inv : SqRealMat;
    end;
```

Konstrukcija `class(RealMat)` znači da klasa koju upravo deklarišemo nasleđuje klasu `RealMat`, sva njena polja i metode, i dodaje novi konstruktor i dva nova metoda: metod `det` koji računa determinantu kvadratne matrice, i metod `inv` koji računa inverznu matricu date matrice.

```
constructor SqRealMat.init(p : integer);
begin
  inherited init(p, p)
end;
```

Konstruktoru `init` za kvadratne matrice je dovoljan samo jedan parametar, zato što kvadratna matrica ima obe dimenzije jednake. Međutim, on u suštini radi isto što i stari (nasleđeni) metod `init` iz klase `RealMat`. Zato je dovoljno pozvati nasleđeni konstruktor `init`, što se postiže upotrebom ključne reči `inhetired`.

```
function SqRealMat.det : real;
begin
  { racuna determinantu }
end;

function SqRealMat.inv : SqRealMat;
begin
  { racuna inverznu matricu }
end;
```

Nakon deklaracije

```
var a, b : SqRealMat;
```

`a` i `b` su instance klase `SqRealMat`. To znači da oba imaju polja `entry`, `M`, i `N` (koje nasledili od klase `RealMat`), nasleđene metode `init`, `dim1`, `dim2`, `input`, `print`, `at`, `put`, `mul`, `mul`, `add`, i nove metode `init`, `det` i `inv`. Evo i primera koji učitava matricu `a` formata 10×10, postavlja `b` na $\mathtt{a}^2$ i potom ispisuje `b`:

```
begin
  a := SqRealMat.init(10);
  writeln('Ucitavanje matrice 10x10');
  a.input;
  b := a.mul(a) as SqRealMat;
  b.print;
end.
```

Primetimo da naredba dodele ima i specijalni modifikator (engl. type cast) oblika "`as SqRealMat`". Problem je u tome što je metod `mul` nasleđen iz klase `RealMat` i on vraća rezultat tipa `RealMat`. Promenljiva `b` je tipa `SqRealMat` i kompajler bi se pobunio, jer pokuzv savamo da pridružimo nekompatibilne tipove. Modifikator "`as SqRealMat`" kaže sistemu da iako rezultat formalno pripada klasi `RealMat`, mi znamo da će to u stvari biti kvadratna matrica, pa sme da ga tretira kao instancu klase `SqRealMat`.

11.3 Tri tradicionalne upotrebe nasleđivanja

Pri nasleđivanju, situacija iz nadklase može malo da se izmeni, tako da možemo razlikovati tri "tradicionalne" upotrebe nasleđivanja:

Specijalizacija. To je situacija u kojoj potklasa doda neka nova polja instancama i eventualno još neke metode. Primer:

```
type
    PlaneFigure = class
        private
            centerX, centerY: integer;
            name : string;
            color : integer;
        public
            constructor init;
            procedure setCenter(x, y: integer);
            procedure setName(newName: string);
            procedure setColor(newColor: integer);
            procedure move(dx, dy: integer);
    end;

    Circle = class(PlaneFigure)
        private
            radius: integer;
        public
            constructor init;
            procedure setRadius(r: integer);
            procedure draw;
    end;
```

Klasa `Circle` je iz klase `PlaneFigure` nasledila strukturu objekta (tj. polja `centerX`, `centerY`, `name`, `color`) i sve metode, i tome još dodala polje `radius` i dva nova metoda. Uočimo da klasa `Circle` ništa nije menjala od onoga što je nasledila.

Proširenje funkcionalnosti. To je situacija u kojoj potklasa ne dodaje nova polja, već samo doda nove metode. Primer:

```
type
    Menu = class
        private
            menuBar: ListOfSubMenu;
        public
```

```
        procedure create;
        procedure remove;
        procedure display;
        procedure clear;
        function getSellection: integer;
end;

CrazyMenu = class(Menu)
    public
        procedure displayEvery(millisec: longint);
        procedure displayRandomly;
        function refuseSellection: integer;
        procedure monkeyWithUser;
end;
```

Modifikacija. To je situacija u kojoj potklasa redefiniše neke (ili sve) metode koje je nasledila iz nadklase. Primer smo videli kod klase `SqRealMat` gde je mnogo metoda preuzeto iz nadklase, ali je jedan (metod `init`) redefinisan.

11.4 Posledice nasleđivanja

Kao što smo pomenuli, nasleđivanje je relacija među klasama. Ona uređuje klase u jednu hijerarhijsku strukturu. Ako je klasa C_1 potklasa klase C_0, onda ćemo to označiti sa $C_0 > C_1$, ili $C_1 < C_0$. Potklasa može biti direktna ili indirektna. Nezavisno od toga koristimo oznaku "$<$". Na primer, ako je $C < C_2 < C_1 < C_0$ niz klasa koje su jedna drugoj direktne potklase, slobodno možemo reći da je $C < C_0$ jer je C doista potklasa od C_0, mada ne direktna.

Neka je X instanca klase C i neka je $C < C_2 < C_1 < C_0$. Ako objektu X uputimo poruku $m(a)$: $X.m(a)$ potrebno je naći implementaciju odgovarajućeg metoda. Prvo se pregleda klasa C. Ako u njoj postoji metod sa imenom "m", onda se poruka $m(a)$ protumači kao poziv metoda "m" iz klase C.

No, ako u klasi C ne postoji metod sa imenom "m", objekt neće odbaciti poruku, već će proveriti da li u nekoj od nadklasa postoji odgovarajući metod. Potraga za metodom se nastavlja u njegovoj direktnoj nadklasi (to je, u ovom primeru, klasa C_2). Ako u klasi C_2 postoji metod koji se zove "m", onda je sve u redu. Poruka $X.m(a)$ se protumači kao primena metoda "m" iz klase C_2 na objekt X.

U suprotnom se proverava da li u klasi C_1 postoji metod "m", itd... Potraga se nastavlja sve dok se ne pronađe odgovarajući metod, ili dok ne stignemo do vrha hijerarhije. Ako nigde nema željenog nam metoda, on se odbacuje (uz odgovarajuće posledice).

Jedna od lepih (ali i zapetljanijih) stvari vezanih za nasleđivanje je da se pro-

menljivoj koja je deklarisana kao promenljiva klase C_1 može dodeliti objekt bilo koje klase C_2 za koju je $C_2 < C_1$. Na primer, ako imamo da je

```
var
  a: PlaneFigure;
  b: Circle;
  m: RealMatrix;
```

onda je dodela `a := b` ispravna, jer je `Circle` potklasa klase `PlaneFigure` (krug je ravanska figura), dok dodela `a := m` nije ispravna, jer `RealMatrix` nije potklasa klase `PlaneFigure` (matrica nije ravanska figura). Ovo se stručno kaže da je `Circle` "assignment compatible" sa `PlaneFigure`, ali da `RealMatrix` to nije. Takođe, kažemo da je `b` "assignment compatible" sa `a`, ali da `m` to nije.

11.5 Vezivanje metoda

Kada imamo promenljivu koja je deklarisana kao promenljiva klase C_1, njoj možemo dodeliti objekt bilo koje klase C_2 za koju je $C_2 < C_1$. To ima interesantne posledice na tumačenje poruke koja je upućena promenljivoj klase C_1.

Tumačenje poruka se još zove i *vezivanje metoda* (engl. method binding) jer se odgovarajući metod "veže" za poruku. *Dinamičko vezivanje metoda* znači da se metodi vezuju za poruke u toku izvršavanja programa, a ne u toku kompilacije. Razlog je krajnje jednostavan: u toku kompilacije niko živ ne zna kojoj klasi će pripadati objekt kome se upućuje poruka. Zato se nikada ne zna odakle treba početi potragu za onim pravim metodom.

Evo primera. Neka je data klasa `PlaneFigure` (kao ranije) i njene potklase `Circle`, `Rectagle`, i `Triangle`:

```
type
    PlaneFigure = class
        private
            centerX, centerY: integer;
            name : string;
            color : integer;
        public
            constructor init;
            procedure setCenter(x, y: integer);
            procedure setName(newName: string);
            procedure setColor(newColor: integer);
            procedure move(dx, dy: integer);
    end;

    Circle = class(PlaneFigure)
```

```
        private
            radius: integer;
        public
            constructor init;
            procedure setRadius(r: integer);
            procedure draw;
    end;

    Rectangle = class(PlaneFigure)
        private
            width, height: integer;
        public
            constructor init;
            procedure setWidth(w: integer);
            procedure setHeight(h: integer);
            procedure draw;
    end;

    Triangle = class(PlaneFigure)
        private
            x1, y1, x2, y2, x3, y3: integer;
        public
            constructor init;
            procedure setVertices(p1, q1, p2, q2, p3, q3: integer);
            procedure draw;
    end;
```

Posmatrajmo ovakve deklaracije:

```
    var
        pf: PlaneFigure;
        c : Circle;
        r : Rectangle;
        t : Triangle;
```

Pretpostavimo da smo inicijalizovali promenljive `c`, `r` i `t`. Sada je `c` neki krug, `r` je neki pravougaonik, a `t` je neki trougao. Ako kažemo

```
    c.draw
```

jasno je da će biti pozvan metod `draw` iz klase `Circle`. No šta se dešava ako kažemo

```
    pf := c;
    pf.draw; { ? }
```

Ideja dinamičkog vezivanja je u tome da run-time sistem proveri šta se nalazi u promenljivoj `pf`. Kada sazna kojoj potklasi klase `PlaneFigure` pripada sadržaj

promenljive pf, počinje da traži metod u odgovarajućoj klasi. Dakle, nakon navedene sekvence opet će biti aktiviran metod draw iz klase Circle. Slično, naredna sekvenca poziva metode draw iz sve tri potklase klase PlaneFigure, redom:

```
pf := c; pf.draw;   { pozove metod iz klase Circle    }
pf := r; pf.draw;   { pozove metod iz klase Rectangle }
pf := t; pf.draw;   { pozove metod iz klase Triangle  }
```

Jasno je da se u toku kompilacije nikako ne može odrediti šta će biti sadržaj promenljive pf, pa tako do samog kraja ostaje neizvesno u kojoj klasi treba početi potragu za metodom draw.

S druge strane, sistem koji podržava *statičko vezivanje*, što znači, vezivanje metoda tokom prevođenja programa, bi u poslednjem primeru jednostavno potražio metod draw u klasi PlaneFigure. Kako tamo nema metoda sa imenom draw i kako PlaneFigure nema nadklasu, sistem sa statičkim vezivanjem bi (neopravdano) prijavio grešku. FreePascal, Java i slični jezici koji se prevode podržavaju mešavinu statičkog i dinamičkog vezivanja. Primeri koje smo naveli ne bi mogli ni da se prevedu u, recimo, FreePascalu. Kasnije ćemo videti tehnike koje nam omogućuju da prevaziđemo probleme.

Dinamičko vezivanje je jako zgodna stvar. Sada ćemo demonstrirati jednu interesantnu primenu. Pozmatrajmo niz ravanskih figura:

```
var
    fig: array [1 .. Max] of PlaneFigure;
    N  : integer; { broj figura u nizu }
```

Priča o assignment compatibility nam daje da element niza fig može biti objekt bilo koje od klasa: Circle, Rectangle, Triangle. Pretpostavimo da smo na neki način popunili elemente niza fig raznim krugovima, pravougaonicima i/ili trouglovima. To se, na primer, može učiniti ovako:

```
N := 0;
repeat
    N := N + 1;
    WhichFigureToRead(kind);
    case kind of
        rect : ReadRectangle(fig[N]);
        circ : ReadCircle(fig[N]);
        tri  : ReadTriangle(fig[N]);
    end
until NoMoreDesireToEnterFigures;
```

Ako treba iscrtati svih *N* figura, ovo je najelegantniji način da se to učini:

```
for i := 1 to N do fig[i].draw;
```

Zbog priče o dinamičkom vezivanju, run-time sistem prvo proveri šta se nalazi u promenljivoj `fig[i]`, pa onda odluči od koje klase da počne pretraživanje. Na taj način će za svaku figuru biti pozvan metod `draw` baš iz njene klase (na trouglove će biti primenjen metod `draw` iz klase `Triangle`, na krugove metod iz klase `Circle`, a na pravougaonike metod iz klase `Rectangle`).

Ako treba pomeriti celu konstelaciju figura za vektor `(DX, DY)`, to se može učiniti ovako:

```
for i := 1 to N do fig[i].move(DX, DY);
```

Uočimo da je `move` metod iz klase `PlaneFigure`. On nije implementiran ni u jednoj od potklasa koje koristimo, ali nasleđivanje obezbeđuje da on ipak bude pronađen i primenjen kako treba. Uočimo da se i u ovom slučaju primenjuje dinamičko vezivanje metoda! Drugim rečima, tek run-time sistem određuje tumačenje poruke "`move(DX, DY)`". Razlog je krajnje jednostavan: u toku prevođenja ne možemo biti sigurni da metod `move` nije redefinisan u nekoj od potklasa! Takođe, može da se desi da će on biti definisan kasnije. U ovom primeru metod `move` nije bio redefinisan ni u jednoj potklasi, pa je pozvan onaj zajednički iz klase `PlaneFigure`.

Da smo kojim slučajem, recimo, u klasi `Rectangle` redefinisali metod `move`, na sve pravougaonike u listi bi bio primenjen njihov metod `move`, dok bi na ostale bio primenjen zajednički metod `move` (tj. onaj iz klase `PlaneFigure`).

11.6 Vrste nasleđivanja

Nasleđivanje može biti *jednostruko* i *višestruko*. Jednostruko nasleđivanje je ono kod koga klasa može da nasledi osobine najviše jedne klase. Kod višestrukog nasleđivanja klasa može da nasledi osobine nekoliko klasa.

Primeri koje smo do sada videli su bazirani na modelu jednostrukog nasleđivanja. Primere višestrukog nasleđivanja ćemo videti malo kasnije. Razlog za to je relativna složenost modela sa višestrukim nasleđivanjem. Ima dosta tehnikalija koje treba pomenuti da bi se dobila potpuna slika.

Još uvek se vodi diskusija o tome da li je dovoljno da programski jezik podržava jednostruko nasleđivanje, ili je mnogo bolje da podržava višestruko. I jedan i drugi model imaju i prednosti i mane. FreePascal podržava samo jednostruko nasleđivanje. Java, s druge strane, podržava jedan međuoblik: osnovni model nasleđivanja u Javi je jednostruko nasleđivanje, ali se mehanizam interfejsa ko-

risti da simulira efekte primitivne varijante višestrukog nasleđivanja. Programski jezik Eiffel podržava punokrvno višestruko nasleđivanje.

11.7 Objektno-orijentisani model

Ako neki programski jezik podržava

- klase,
- objekte kao instance klasa,
- učaurenost,
- nasleđivanje,
- mehanizam poruka, i
- polimorfizam.

to je objektno-orijentisani jezik. Nasleđivanje može biti i jednostruko i višestruko. Model nasleđivanja ne utiče na objektnu orijentisanost jezika, važno je da jezik podržava neki model nasleđivanja. Takođe, neki objektni jezici podržavaju jedino statičko vezivanje metoda. To ne znači da oni nisu objektni jezici, već samo da imaju manju izražajnu moć.

Ovim je završen osnovni deo Sage o OO jezicima i varijetetima čija je namena bila da ustanovi definicije. U nastavku Sage će se govoriti o nekim naprednim tehnikama koje jesu bitne za OO, ali nisu ono što definiše OO. Stvari o kojima ćemo govoriti uglavnom spadaju u tehnike implementacije OO jezika, ili su posledica implementacije.

Glava 12

Napredne OO teme

Prvi deo Sage (prethodne dve glave) uveo je osnovne pojmove OOP i demonstrirao ih na primerima. Sada ćemo malo pričati o tehnikama implementacije OO programskih jezika i svim dosetkama i začkoljicama koje su potrebne da bi se od ideje moglo preći na nešto što je upotrebljivo. Tu spadaju konstruktori i destruktori, generičke klase, apstraktni metodi ... Dakle, sve ono o čemu ljudi glasno govore kada žele da se spontano pročuje u društvu da su veliki magovi OOP-a.

Treba sve vreme imati na umu da ono o čemu će biti reči u ovom delu Sage *nije ono što karakteriše OOP*, već je samo posledica odluka u dizajniranju OO jezika i implementacije njihovih prevodilaca. Pri kraju Sage ćemo reći i nekoliko reči o višestrukom nasleđivanju, njegovim prednostima i problemima koji se javljaju prilikom njegove implementacije.

Neki OO jezici se prevode, a neki se interpretiraju. Neki podržavaju *strong typing* i imaju statičku proveru tipova, a neki su tzv. *typeless jezici*, tj. jezici kod kojih se provera tipova vrši u toku izvršenja programa (*run-time*).

Sve tehnike o kojima ćemo govoriti u ovom delu su karakteristične za jezike koji imaju statičku proveru tipova i koji se prevode. OO jezici koji su *typeless* i interpretiraju se nemaju potrebu za tolikim komplikacijama. Ovim se samo još jednom potvrđuje da priča koja sledi *nije ono što karakteriše OOP*, već je samo posledica implementacije prevodilaca za *strong-typed* OO jezike.

To ne znači da su *typeless* OO jezici koji se interpretiraju "manje OO". Oni su u istoj meri objektno orijentisani, s tom razlikom što mnogo više provera ostavljaju run-time sistemu. Time se dobija elegantniji jezik, ali se gubi na efikasnosti.

12.1 Problem sa smeštanjem objekata

Posmatrajmo klasu `PlaneFigure` i njene potklase `Circle`, `Rectangle` i `Triangle` koje su nam sve poznate još iz prethodne glave:

```
type
    PlaneFigure = class
        private
            centerX, centerY: integer;
            name : string;
            color : integer;
        public
            constructor init;
            procedure setCenter(x, y: integer);
            procedure setName(newName: string);
            procedure setColor(newColor: integer);
            procedure move(dx, dy: integer);
    end;

    Circle = class(PlaneFigure)
        private
            radius: integer;
        public
            constructor init;
            procedure setRadius(r: integer);
            procedure draw;
    end;

    Rectangle = class(PlaneFigure)
        private
            width, height: integer;
        public
            constructor init;
            procedure setWidth(w: integer);
            procedure setHeight(h: integer);
            procedure draw;
    end;

    Triangle = class(PlaneFigure)
        private
            x1, y1, x2, y2, x3, y3: integer;
        public
            constructor init;
            procedure setVertices(p1, q1, p2, q2, p3, q3: integer);
            procedure draw;
    end;
```

Ako pretpostavimo da tip `integer` zauzima 2 bajta, a tip `string` 256 bajta, za pamćenje instance klase `PlaneFigure` nam je potrebno 262 bajta, za instancu klase `Circle` nam je potrebno 264 bajta (jer ona od klase `PlaneFigure` nasleđuje sva polja i dodaje još polje `radius` što daje $262+2=264$ bajta), za instancu klase `Circle` nam je potrebno 266 bajta, a za instancu klase `Triangle` čak 274 bajta. Pretpostavimo da se negde u programu javlja deklaracija:

```
var pf: PlaneFigure;
```

Recite, koliko bajtova treba alocirati za promenljivu `pf`? Da li 262 (zato što je to osnovna veličina) ili 264 (zato što priča o assignment compatibility daje da se promenljivoj `pf` uvek može dodeliti promenljiva tipa `Circle`)? A šta ako negde u budućnosti napravimo potklasu klase `PlaneFigure` čija instanca zahteva 12012 bajtova za smeštanje? Da li odmah treba alocirati najveći mogući prostor, pa neka zvrji prazan ako ga ne upotrebimo?

12.2 Rešenje problema: pokazivačka semantika

Skoro svi OO programski jezici ovu dilemu razreše na najjednostavniji mogući način: instanca klase se tretira kao pokazivač na nešto. Zato će deklaracija

```
var pf: PlaneFigure;
```

navesti prevodilac da za promenljivu `pf` alocira samo 4 bajta (koliko je, recimo, potrebno za pokazivač), dok sadržaj, tj. ono na šta `pf` pokazuje, alocira naknadno na heapu, u toku rada programa. Na ovaj način se obezbeđuje uniformnost pri dodeljivanju: kada se promenljivoj `pf` dodeljuje instanca klase `Circle`, recimo, tada se prebacuju samo pokazivači, čija veličina je konstantna na fiksiranoj platformi.

Za ovakav pogled na objektivnu realnost kaže se da implementira *pokazivačku semantiku* (engl. reference semantics). Svi popularni jezici koriste pokazivačku semantiku. Ako je jezik još i dobro dizajniran, pokazivači se uopšte ne vide ili se tek naslućuju.

FreePascal takođe implementira pokazivačku semantiku i sada je pravo mesto da objasnimo razliku između klasa deklarisanih sa `object` i klasa deklarisanih sa `class`:

u FreePascalu `class` je pokazivač na `object`!

To, dalje, znači da

- instance klase definisane sa `object` su *statičke*, nalaze se u delu memorije sa ostalim statičkim promenljivim i prevodilac sam rezerviše prostor za njih, dok

- instance klase definisane sa `class` su *dinamičke*, nalaze se na heapu i za njih moramo eksplicitno alocirati prostor.

12.3 Problemi sa rešenjem problema

Postoji jedna od posledica Marfijevog zakona (Beckhapov zakon) koja kaže

$$\boxed{\text{lepota}} * \boxed{\text{pamet}} = \text{const.}$$

Zato jako pametno rešenje uglavnom nije i jako lepo. To važi i za pokazivačku semantiku: mada rešava jedan problem, pokazivačka semantika sa sobom donosi nove.

Prvi veliki problem je u tome da prevodilac alocira samo pokazivač na objekt. To znači da se prostor za sadržaj mora alocirati naknadno. Tako dolazimo do pojma konstruktora. Konstruktor je nešto (procedura, metod, specijalna oznaka ili slično) što alocira prostor za instancu date klase. On se mora pozivati eksplicitno.

Na primer, u FreePascalu su konstruktori realizovani kao posebni metodi (metodi pridruženi klasi). Svaka klasa ima metod označen sa `constructor` koji alocira prostor za instancu. U primeru sa ravanskim figurama, ako u promenljivu `c` želimo da smestimo instancu klase `Circle`, to radimo tako što klasi `Circle` uputimo poruku `init`. Ona napravi novu instancu i vrati pokazivač:

```
var c : Circle;
...
c := Circle.init;
```

U Javi se to radi pozivom posebnog metoda (koji ima isto ime kao i klasa, pa tako znamo da je to konstruktor), a u Eiffelu koristeći specijalnu sintaksnu konstrukciju (eventualno uz neku inicijalizacionu poruku):

```
x: SOMETHING;
y: SOMETHING_ELSE;
 ...
!!x;
!!y.createProcedure(...);
```

Druga vrsta problema su destruktori. Destruktor služi da počisti đubre, tj. instance koje su odslužile svojoj svrsi, pa memoriju koju su zapremale treba osloboditi i reciklirati. U najvećem broju kulturnih OO jezika recikliranje radi poseban deo run-time sistema koji se zove *garbage collector*. Tako rade, recimo, Smalltalk, Eiffel i Java. No, FreePascal zahteva da se i destruktori pozivaju eksplicitno. Svaka dinamička klasa u FreePascalu standardno poseduje destruktor `destroy` koji se može pozvati, na primer, ovako:

```
var
  a, b : SqRealMat;

begin
  a := SqRealMat.init(10);
  writeln('Ucitavanje matrice 10x10');
  a.input;
  b := a.mul(a) as SqRealMat;
  b.print;
  a.destroy;
  b.destroy
end.
```

Dakle, u FreePascalu programer ne samo da mora da brine o kreiranju objekata, već i o njihovom uništavanju.

Treća vrsta problema se odnosi na dodeljivanje. Ako su `c` i `d` promenljive, one sadrže pokazivače na sadržaj objekta, pa dodela

```
c := d;
```

samo prebaci pokazivač. Zato se kod svih OO jezika koji realizuju pokazivačku semantiku (a to su skoro svi OO jezici) mora voditi računa o dva nivoa dodele:

- prebacivanje pokazivača, i
- pravljenje kopije objekta.

Dodela oblika `c := d` u skoro svim OO jezicima predstavlja samo prebacivanje pokazivača. Pravljenje kopije objekta se mora eksplicitno zahtevati. Npr, u Smalltalku tome služe metodi `shallowCopy` i `deepCopy`, a u Eiffelu tome služe metodi `copy` i `clone`, dok u FreePascalu programer mora sam da piše metode koji prave kopiju objekta.

12.4 Problem sa upućivanjem poruka

Skoro svi moderni OO jezici koji se prevode dizajnirani su tako da podržavaju strog sistem tipova i shodno tome statičku proveru tipova. To znači da se provere tipova vrše u toku prevođenja. S jedne strane, to je dobro zato što se još u ranim fazama otkrivaju mnoge greške. S druge strane, statička provera tipova je u svađi sa dinamičkim vezivanjem metoda. Evo dva primera.

Posmatrajmo ponovo klasu `PlaneFigure` i njene potklase `Circle`, `Rectagle` i `Triangle`. Uzmimo još da je `fig` niz ravanskih figura deklarisan ovako:

```
var fig: array [1 .. MaxN] of PlaneFigure;
```

i da su u fig[1] do fig[N] nekako smešteni razni krugovi, pravougaonici i trouglovi. Već smo konstatovali da je najelegantniji način da se iscrta svih N figura ovaj:

```
for i := 1 to N do
    fig[i].draw;
```

No, to su samo naše želje! Pogledajmo kako se prevodilac ponaša kada počne da prevodi ove dve linije. Naredba for je jasna (pod pretpostavkom da su i i N promenljive deklarisane na odgovarajući način). Prevodilac počinje da prevodi drugu liniju. Pogleda promenljivu fig[i], primeti da je ona instanca klase PlaneFigure i da joj se upućuje poruka draw. Prevodilac bi sada želeo da obavi proveru tipova, pa razgleda malo po definiciji klase PlaneFigure. No, tamo nema metoda koji bi odgovarao poruci draw! *Compilation error*.

Pretpostavimo sada da smo ovoj grupi klasa dodali i novu klasu FilledRect koja nasleđuje klasu Rectangle i predstavlja popunjene pravougaonike:

```
type
    FilledRect = class(Rectangle)
        public
            constructor init;
            procedure draw;
    end;
```

Geometrija popunjenog pravougaonika je ista geometriji "običnog" pravougaonika. Jedina razlika je u načinu na koji se popunjeni pravougaonici isrtavaju. Zato klasa FilledRect samo redefiniše metod za crtanje figure. Neka je sada

```
    var
        r : Rectangle;
        fr : FilledRect;
    begin
        fr := FilledRect.init;
        fr.setName('Filled Rect 1');
        fr.setCenter(12,14);
        fr.setColor(255);
        fr.setWidth(186);
        fr.setHeight(212);

        r := fr;
        r.draw
    end.
```

Kod poruke r.draw, koji draw metod će biti pozvan: onaj iz klase Rectangle (zato što je Rectangle statički tip promenljive r), ili onaj iz klase FilledRect (zato što nakon dodele r := fr promenljiva r sadrži pokazivač na objekt klase

FillRect, pa je FillRect njen dinamički tip)? Voleli bismo da bude pozvan metod iz klase FilledRect, ali će FreePascal koristiti statičku proveru tipova i tokom prevođenja programa se ipak opredeliti za metod iz klase Rectangle.

Dakle, statička provera tipova nam kvari želje. Izgleda da se statička provera tipova i dinamičko vezivanje metoda ne trpe. A zbog lepote obe ideje želimo da zadržimo i jedno i drugo. Šta ćemo sad?

12.5 Rešenje problema: virtuelni i apstraktni metodi

Rešenje drugog problema je dosetka koja se zove *virtuelni metodi*. Klasu Rectangle ćemo opisati ovako:

```
type
    Rectangle = class(PlaneFigure)
        private
            width, height: integer;
        public
            constructor init;
            procedure setWidth(w: integer);
            procedure setHeight(h: integer);
            procedure draw; virtual;
    end;
```

Deklaracija virtual znači da je metod draw virtuelni metod, i da se može desiti da će ga neka od potklasa klase Rectangle redefinisati. Potklasu FilledRect sada opišemo ovako:

```
type
    FilledRect = class(Rectangle)
        public
            constructor init;
            procedure draw; override;
    end;
```

Deklaracija override znači da ova deklaracija metoda treba da prekrije deklaraciju odgovarajućeg metoda iz natklase. Sada će prevodilac pri prevođenju programa

```
    var
        r : Rectangle;
        fr : FilledRect;
    begin
        fr := FilledRect.init;
        fr.setName('Filled Rect 1');
        fr.setCenter(12,14);
```

```
        fr.setColor(255);
        fr.setWidth(186);
        fr.setHeight(212);

        r := fr;
        r.draw
    end.
```

primetiti da je metod `draw` virtuelni, pa će generisati dodatni kōd koji tokom izvršavanja programa proverava dinamički tip promenljive `r` i aktivira metod `draw` prema dinamičkom tipu objekta.

Rešenje prvog problema koristi dosetku koja se zove *apstraktni metodi*. Metod je *apstraktan* ako nije deklarisan u toj klasi, ali je njegova deklaracija prisutna u opisu klase. Na primer, klasu `PlaneFigure` treba definisati ovako:

```
type
    PlaneFigure = class
        private
            centerX, centerY: integer;
            name : string;
            color : integer;
        public
            constructor init;
            procedure setCenter(x, y: integer);
            procedure setName(newName: string);
            procedure setColor(newColor: integer);
            procedure move(dx, dy: integer);
            procedure draw; virtual; abstract;
    end;
```

Deklaracija `virtual` znači da je metod virtuelni, čime je omogućeno da potklase redefinišu metod. Deklaracija `abstract` predstavlja obećanje prevodiocu. Ona znači da će metod `draw` biti definisani negde u nekoj potklasi, ali da ćemo se na njih pozivati pre nego što budu implementirani.

Deklaracija `abstract` je slična deklaraciji `forward` u standardnom Pascalu. Kao što `forward` znači da će odgovarajuća procedura biti definisana kasnije, ali da je moramo koristiti pre nego što bude definisana, tako i `abstract` obećava prevodiocu da ćemo odgovarajući metod definisati u nekoj potklasi klase, ali da ipak moramo da ga koristimo pre toga.

Ako je klasa `PlaneFigure` definisana na ovaj način, prevođenje do sada problematičnog parčeta kōda

```
for i := 1 to N do
    fig[i].draw;
```

teče bez problema. Kada shvati da je `fig[i]` instanca klase `PlaneFigure`, i da joj je upućena poruka `draw`, prevodilac pronjuška po definiciji klase `PlaneFigure`, vidi da tamo postoji metod `draw`, proveri tipove (u ovom slučaju nema šta da proverava zato što metod nema argumenata), i uradi sve ostalo što treba da uradi. Činjenica da je metod `draw` apstraktan ga eventualno može naterati da generiše i neki dodatni kod koji će u toku izvršenja programa proveravati da li je programer ispunio obećanje i obezbedio metod `draw` u svim potklasama koje se koriste. Potklase klase `PlaneFigure` sada moraju biti deklarisane ovako:

```
type
    Circle = class(PlaneFigure)
        private
            radius: integer;
        public
            constructor init;
            procedure setRadius(r: integer);
            procedure draw; override;
    end;

    Rectangle = class(PlaneFigure)
        private
            width, height: integer;
        public
            constructor init;
            procedure setWidth(w: integer);
            procedure setHeight(h: integer);
            procedure draw; override;
    end;

    Triangle = class(PlaneFigure)
        private
            x1, y1, x2, y2, x3, y3: integer;
        public
            constructor init;
            procedure setVertices(p1, q1, p2, q2, p3, q3: integer);
            procedure draw; override;
    end;
```

Apstraktni metodi se javljaju u onim situacijama u kojima se metod implementira na potpuno različite načine u različitim potklasama te klase. Kao primer može da posluži klasa `PlaneFigure` i njene potklase `Rectangle`, `Circle` i `Triangle`. Krug, pravougaonik i trougao se crtaju na potpuno različite načine. Jasno je da nije moguće da se u klasi `PlaneFigure` obezbedi metod koji bi bio u stanju da nacrta sve što može da bude ravanska figura. Zato se u klasi `PlaneFigure` samo označi da postoji metod `draw` kojga moraju imati sve potklase klase `PlaneFigure`

i da će on biti implementiran tek tamo. Najviše što se može očekivati od klase `PlaneFigure` je to da nagovesti potrebu za ovim metodom. Ona ništa ne može da kaže o njegovoj implementaciji.

Kada se prevodi program napisan na nekom OO jeziku, a koji ima apstraktne metode, statička provera tipova utvrđuje da li su ispunjena sva obećanja koja je programer dao. Dakle, prevodilac proverava da li postoji bar jedna implementacija za svaki apstraktni metod koji se javlja u programu. Naravno, može ih postojati više. Dinamičko vezivanje metoda određuje koju implementaciju apstraktnog metoda treba shvatiti kao interpretaciju poruke.

Klasa koja sadrži apstraktne metode se zove *apstraktna klasa*. Klasa koja nema apstraktne metoda se zove *konkretna klasa*. Potklasa apstraktne klase ne mora ponuditi implementaciju za apstraktne metode. Štaviše, potklasa apstraktne klase može dodati i nove apstraktne metode. Bitno je samo to da za svaku apstraktnu klasu postoji njena potklasa koja je konkretna. Time se obezbeđuje da svaki apstraktni metod bude nekad i negde implementiran.

Takođe, moguće je da potklasa neke konkretne klase bude apstraktna. Za nju tada važi sve što važi i za ostale apstraktne klase.

12.6 Generički kōd i generičke klase

Posmatrajmo klasu `Stack` čije instance su stekovi. Za stek nije bitno šta se na njega stavlja. Mi znamo da je stek nešto na čega možemo staviti nešto drugo i to kasnije sa njega skinuti. S druge strane, kada implementiramo stek, moramo eksplicitno navesti šta je to što se na stek stavlja. Na primer, u standardnom Pascalu se stek na koga se stavljaju celi brojevi opisuje ovako:

```
type
    IntStack = ↑IntStackEntry;
    IntStackEntry = record
        info: integer;
        link: IntStack
    end;
```

dok se stek na koga se stavljaju realni brojevi opisuje ovako:

```
type
    RealStack = ↑RealStackEntry;
    RealStackEntry = record
        info: real;
        link: RealStack
    end;
```

Opis je skoro isti. Štaviše, operacije za manipulaciju ovim stekovima su još sličnije, baš zato što stek kao struktura podataka savršeno ne zavisi od onoga što se na

njega stavlja. Bilo bi jako lepo kada bi nam jezik pružao mogućnost da opišemo stek nezavisno od toga šta se na njega stavlja, pa da od te jedinstvene specifikacije kasnije nekako napravimo konkretnije stvari (kao što su stek celih brojeva i stek realnih brojeva).

Mehanizam koga mnogi jezici eksploatišu da bi ovo ostvarili se zove "generičke klase" (ili "generički moduli"). Evo primera klase `Stack` u pseudojeziku (pseudo jezik je kobajagi jezik; to je jezik koji zapravo ne postoji):

```
class Stack[class T];
private
    s  : array of T;
    max: integer;
    top: integer;
public
    init(M: integer) is
    begin
        new(s, M); max := M; top := 0
    end;

    push(x: T) is
    begin
        if top > max then ERROR("Stack is full")
        else begin
            s[top] := x;
            inc(top)
    end end;

    pop(var x: T) is
    begin
        if top = 0 then ERROR("Stack is empty")
        else begin
            x := s[top];
            dec(top)
    end end;

    full(): Boolean is
    begin
        return top > max
    end;

    empty(): Boolean is
    begin
        return top = 0
    end;
end.
```

Generička klasa predstavlja šemu po kojoj se mogu praviti razne klase. To je način da se odjednom opiše beskonačno mnogo klasa. Za svaku vrednost parametra `T` dobija se jedna nova klasa. Primeri upotrebe klase `Stack` izgledaju ovako (u istom pseudojeziku):

```
var IntStack : Stack[integer];
    RealStack: Stack[real];
```

Sada su `IntStack` i `RealStack` instance dve različite klase (s tim da su te dve klase nastale "konkretizacijom" jedne iste generičke klase). Proces u kome se od generičke klase pravi konkretna klasa se zove instanciranje generičke klase. Parametar generičke klase je najčešće neka druga klasa i/ili neka konstanta.

Za neke generičke klase je bitno da klasa koja je generički parametar ima neka specijalna svojstva. Na primer, kada pravimo klasu `BinSearchTree` koja opisuje uređena binarna stabla, bitno je da elementi koji se smeštaju u čvorove stabla budu takvi da se mogu upoređivati. Dobro je da se sva takva posebna svojstva posebno naglase u deklaraciji generičke klase. Evo primera. Neka je data apstraktna klasa `LinOrder` koja opisuje linearno uređene elemente:

```
class LinOrder;
public
    lt (y: LinOrder): Boolean is abstract;

    le (y: LinOrder): Boolean is
    begin
        return self.lt(y) or (self = y)
    end le;

    gt (y: LinOrder): Boolean is
    begin
        return y.lt(self)
    end gt;

    ge (y: LinOrder): Boolean is
    begin
        return self.gt(y) or (self = y)
    end ge;

end.
```

Klasa `BinSearchTree` tada može da se opiše ovako:

```
class BinSearchTree[class T(inherits LinOrder)];
private
    info       : T;
    left, right: BinSearchTree[T];
```

```
    public
        new(x: T) is
        begin
            info := x;
            left := NIL; right := NIL
        end;

        insert(x: T) is
        begin
            if null(self) then
                ERROR("Argh!")
            else if x.lt(info) then
                if null(left) then
                    left.new(x)
                else
                    left.insert(x)
            else if null(right) then
                right.new(x)
            else
                right.insert(x)
        end;
    end.
```

Posebnom deklaracijom koju smo naveli u zagradi iza generičkog argumenta naglašavamo da se kao argument `T` može pojaviti bilo koja klasa koja je potklasa klase `LinOrder`. To nam treba zato što elemente koje smeštamo u čvorove stabla moramo porediti.

Život se i dalje komplikuje. Jedna od lepih komplikacija je to da se mogu praviti potklase generičke klase. U takvim slučajevima i potklasa mora biti generička. No, tu treba uočiti jednu suptilnost. U ovom primeru:

```
class WeirdStack;
inherits Stack[integer];
...
end.
```

klasa `WeirdStack` nije potklasa generičke klase. Klasa `WeirdStack` je potklasa obične klase, s tim da je ta obična klasa nastala instanciranjem generičke klase. Evo, sada, primera klase koja je doista potklasa generičke klase:

```
class AnotherWeirdStack[T];
inherits Stack[T];
...
end.
```

Ova klasa je generička i nasleđuje generičku klasu. Tako, možemo praviti čitave mreže generičkih klasa i na taj način pisati veoma apstraktan kod.

12.7 Višestruko nasleđivanje

Višestruko nasleđivanje je mehanizam u kome jedna klasa može nastati nasleđivanjem osobina nekoliko klasa. Evo primera (i dalje u pseudojeziku):

```
class Color;
private
    col: integer;
public
    setColor(c: integer) is ...;
    colorOf(): integer   is ...;
end.

class BwPlaneFigure;
private
    centerX, centerY: integer;
    name            : string;
public
    setCenter(x, y: integer)  is ...;
    setName(newName: string)  is ...;
    move(dx, dy: integer)     is ...;
end.

class ColoredCircle;
inherits Color, BwPlaneFigure;
private
    radius: integer;
public
    draw() is ...;
    wipe() is ...;
end.
```

Klasa `ColoredCircle` je klasa koja je nasledila osobine od dve klase: `Color` i `BwPlaneFigure`. Ona nasleđuje polja i metode od obe klase, i uz put doda još koje polje i metod. Ako je `cc` deklarisana ovako

```
var cc: ColoredCircle;
```

onda je ovo validan niz poruka:

```
cc.setColor(7);
cc.setName("Crveni krug");
cc.setCenter(10,10);
cc.move(-17,6);
...
if cc.colorOf() = 13 then ... ;
```

Instanca klase `ColoredCircle` ima osobine i jedne i druge klase.

Problemi nastaju kada klase od kojih naša klasa nasleđuje osobine imaju metode sa istim imenom. Na primer ovako:

```
class Col;
private
    col: integer;
public
    set(c: integer) is ...;
    get(): integer  is ...;
end.

class BwPF;
private
    centerX, centerY: integer;
    name            : string;
public
    set(x, y: integer)       is ...;
    setName(newName: string) is ...;
    move(dx, dy: integer)    is ...;
end.

class ColCirc;
inherits Col, BwPF;
private
    radius: integer;
public
    draw() is ...;
    wipe() is ...;
end.
```

Prilikom prevođenja klase `ColCirc` prevodilac uočava da klase `Col` i `BwPF` imaju metod sa istim imenom: `set`. *Compilation error: Name clash.* To se često dešava zato što se neke operacije, mada nad različitim vrstama podataka, najbolje opisuju istim imenom.

Najjednostavniji način da se razreše problemi sa istoimenim metodima je da se uvede mehanizam promene imena. Ideja je u tome da se u sklopu `inherits` deklaracije omogući da se nekim metodima promeni ime samo za potrebe tekuće klase. I to radi fino. Rešenje problema `ColCirc` bi moglo da izgleda ovako:

```
class ColCirc;
inherits Col(rename set as setCol),
         BwPF(rename set as setPos);
private
    radius: integer;
public
```

```
    draw() is ...;
    wipe() is ...;
end.
```

Kada pišemo kōd za klasu `ColCirc` jedan metod ima ime `setCol`, a drugi `setPos` i time je konflikt razrešen. Treba imati u vidu da je ova promena imena, kao što je već rečeno, lokalna. Imena `setCol` i `setPos` važe samo u klasi `ColCirc`. Ostatak univerzuma te metode vidi pod starim imenom i, ako zatreba, može da im (opet lokalno) promeni imena. I pored svega, ostaje još nekoliko komplikacija kod mehanizma vezivanja metoda u višestrukom nasleđivanju, o kojima nećemo ovom prilikom.

12.8 Ponovljeno nasleđivanje

Posmatrajmo klasu `A`. Neka su `B1` i `B2` njeni naslednici i neka je klasa `C` naslednik klasa `B1` i `B2`:

```
class A;
...
end.

class B1;           class B2;
inherits A;         inherits A;
...                 ...
end.                end.

class C;
inherits B1, B2;
...
end.
```

Klasa `A` obezbeđuje neka polja i metode. Svaka instanca klase `B1` ima svoju kopiju polja iz klase `A`, dok svaka instanca klase `B2` ima svoju kopiju. Da li instance klase `C` treba da imaju dve kopije polja koje potiču iz klase `A` ili samo jednu? Drugim rečima, koliko puta će klasa `C` naslediti klasu `A`? Da li dva puta ili jednom?

Ima opravdanja i za jedno i za drugo! Ima situacija kada je potrebno klasu `A` naslediti jednom, a ima situacija kada je potrebno klasu `A` naslediti više puta. Evo primera:

```
class Address;
...
private
    street: string;
    no    : integer;
```

```
    city  : string;
    zip   : integer;
...
end.

class Home;                 class Business;
inherits Address;           inherits Address;
...                         ...
end.                        end.

class HomeBusiness;
inherits Home, Business;
...
end.
```

Ako pretpostavimo da klasa `Address` opisuje podatke o adresi (poštanskoj) neke zgrade, ako klasa `Home` opisuje kuću u kojoj neko stanuje, klasa `Business` opisuje posao kojim se neko bavi, a klasa `HomeBusiness` opisuje posao kojim se neko bavi kod svoje kuće, onda klasa `HomeBusiness` treba samo jednom da nasledi klasu `Address`. Razlog je jednostavan: obe stvari se dešavaju na istoj adresi!

Pogledajmo, sada, drugi primer:

```
class Color; ... end.

class Jacket;    inherits Color; ... end.
class Trousers;  inherits Color; ... end.
class Shirt;     inherits Color; ... end.
class Underwear; inherits Color; ... end.
class Shoes;     inherits Color; ... end.

class Outfit;
inherits Jacket, Trousers, Shirt, Underwear, Shoes;
...
end.
```

Kako svaki deo odeće ima svoju boju, i kako sve boje mogu biti različite, klasa `Outfit` treba da nasledi klasu `Color` pet puta.

Ako jedna klasa nasleđuje drugu klasu više od jednom, takav oblik nasleđivanja se zove *ponovljeno nasleđivanje* (engl. repeated inheritance). Jasno je da se takav oblik nasleđivanja može javiti samo kod OO jezika sa višestrukim nasleđivanjem.

Razni jezici rešavaju problem ponovljenog nasleđivanja na razne načine. Neki jezici jednostavno zanemaruju ovaj problem i opredele se isključivo za jednu ili drugu varijantu kod ponovljenog nasleđivanja: ili se svaka klasa nasledi tačno jednom, ili se svaka klasa nasledi onoliko puta koliko se puta njene potklase pojave i inherits listi.

To su inferiorna rešenja. Bolji jezici to reše tako što omoguće oba mehanizma nasleđivanja, pa programer odabere verziju koja mu odgovara. Kao što smo videli, oba pogleda su potrebna.

Ovakav pristup ima Eiffel. On omogućuje da se klasa nasledi jednom ili više puta, a to postiže kroz suptilan mehanizam promene imena (pravilo je jednostavno: ako se imena metoda iz klase preimenuju, onda se klasa nasleđuje nekoliko puta; ako se pak ne preimenuju već se zadrže isključivo originalna imena, onda se klasa nasledi samo jednom).

12.9 Čisti i hibridni OO jezici

OO jezika ima raznih vrsta i neke klasifikacije smo pomenuli još na početku. No, ima i drugih podela. Jedna od značajnijih je podela na *čiste* i *hibridne* OO jezike. OO jezik je čist ako se svi podaci opisuju u terminima klasa i sve akcije se iskazuju u obliku upućivanja poruka. Na primer, u čistim jezicima se sve akcije, pa i akcija

```
a := b.nameOf
```

shvata kao upućivanje poruke objektu (u ovom slučaju, objektu `a` upućujemo poruku `:= b.nameOf`). Čisti OO jezici su, na primer, Java, Smalltalk i Eiffel.

Hibridni jezici su mešavine OO i nekog drugog stila, recimo imperativnog. Na primer, FreePascal je hibridni jezik. Kod hibridnih jezika se klase, objekti i poruke mešaju sa tipovima, strukturama i naredbama. Hibridni (imperativni) jezici podržavaju bar dve vrste akcija: slanje poruke i poziv procedure. Tako, u velikom broju hibridnih (imperativnih) jezika se akcija

```
a := b.nameOf
```

shvata kao poziv "procedure" `:=` sa argumentima `b.nameOf` i `a`. Naravno, kao hibridni jezici se, pored imperativnih OO jezika, mogu javiti logički OO jezici, funkcionalni OO jezici,...

Zbog toga što podržavaju mešavinu pogleda na svet, hibridni jezici mogu biti veoma komplikovani. Njihova semantika je najčešće formulisana u obliku niza složenih pravila.

Za hibridne OO jezike je karakteristično još i to da se klase implementiraju kao posebna vrsta tipova podataka. Tako, jedan modul može sadržati opis nekoliko klasa. Kod čistih OO jezika to uglavnom nije tako. Tu klase služe i kao mehanizam uvođenja novih vrsta podataka i kao mehanizam razbijanja programa na module.

12.10 Zaključak

OO ideja je prilično stara (1967), ali tek 1990-ih postaje popularna. Moglo bi se čak reći da se desilo i nešto nepoželjno: ne samo da je postala priznata kao lepa i korisna ideja, već je postala moda. OO je danas *in*.

To nije dobro. Ima problema za koje nema potrebe koristiti OO. Ima problema koji se mogu mnogo brže i jednostavnije rešiti klasičnim metodama. S druge strane, projektovanje velikih softverskih sistema može biti mnogo lakše uz OO.

OO ne treba gurati tamo gde mu nije mesto. Na primer, OO sistemi nose odgovarajući overhead bar zbog dve stvari: dinamičkog vezivanja metoda i kupljenja đubreta (garbage collection). Zato oni uglavnom nisu pogodni za real-time aplikacije. OO je korisna i dobra metodologija, ali nikako ne i najbolja. Sve to treba imati u vidu kada se bira alat pri ulasku u veliki projekt.