Gintautas GRIGAS

P R O G R A M A V I M A S

P A S K A L I U

Vilnius 1998

Ši knyga yra programavimo pradžiamokslis. Aprašomos pagrindinės Paskalio kalbos konstrukcijas bei jų bei jų panaudojimas algoritmams ir programoms užrašyti. Medžiaga pateikiama taip, kad skaitytojas galėtų ją įsisavinti spręsdamas uždavinius. Pateikiami uždaviniai ir praktikos darbai. Uždaviniai skirti savikontrolei. Knygos pabaigoje pateikiami jų sprendimai (atsakymai). Praktikos darbai – tai nedideli programavimo darbai, skirti programavimo įgūdžiams pagilinti bei darbo su kompiuteriu praktikai.

Knyga turėtų būti naudinga vyresniųjų klasių moksleiviams, informatikos mokytojams ir pirmųjų kursų pedagoginės krypties studentams.

Pastebėtos klaidos buvo ištaisytos 2000 m.

Darbas atliktas Matematikos ir informatikos institute
ir Vilniaus pedagoginiame universitete

Darbą rėmė Atviros Lietuvos fondas.

TURINYS

Į V A D A S

Mūsų tikslas – išmokti programuoti. Programavimo darbai prasideda nuo uždavinio formulavimo ir baigiasi galutinio produkto – programos – sukūrimu. Uždavinio formuluotėje pasakoma *ką* reikia padaryti, bet nepasakoma *kaip*. Pavertimas *ką* į *kaip* ir yra programavimo tikslas. Pirmiausia reikia sugalvoti, *kaip* išspręsti uždavinį, t.y. rasti jo sprendimo būdą. Po to reikia sprendimą išreikšti algoritmu (programa), o programą išbandyti kompiuteriu.

Parašyti tobulą, t.y. teisingą ir patogią naudotis, programą iš karto ne visada pavyksta (teisingiau – beveik niekada nepavyksta). Programą tenka daug kartų taisyti bei tobulinti. Programos rašymas yra ilgas darbas, reikalaujantis sumanumo, kruopštumo, atkaklumo. Tačiau visus vargus atperka kūrybinis džiaugsmas, kai pagaliau gaunamas tobulas, veikiantis produktas, kuriuo galima ne tik pačiam pasidžiaugti, bet ir kitiems parodyti.

Programas rašysime Paskalio kalba. Tačiau pagrindinis mūsų tikslas yra programavimas, o ne programavimo kalba. Programavimo kalba (šiuo atveju Paskalis) yra tik darbo įrankis pagrindiniam tikslui pasiekti. Tačiau norint sėkmingai naudotis įrankiu, reikia gerai žinoti tą įrankį, turėti darbo su juo įgūdžių. Todėl būtų logiška pirmiau išmokti įrankį – Paskalį, o po to programuoti. Bet šis įrankis sudėtingas ir jam tenka skirti nemažai laiko ir pastangų.

Bet kokį įrankį studijuoti geriau ir maloniau, kai su juo dirbamas realus darbas. Todėl Paskalį ir programavimą studijuosime pakaitomis. Išmokę vos keletą Paskalio konstrukcijų, spręsime paprastesnius uždavinius, kuriems tų Paskalio žinių pakaks, po to nagrinėsime naujas Paskalio konstrukcijas, spręsime naujus, sudėtingesnius uždavinius. Knygos pradžioje daugiau kalbėsime apie Paskalį, o toliau – apie programavimą.

Nesileisime į Paskalio subtilybes. Trumpai aptarsime tik tas Paskalio konstrukcijas, kurių prireiks rašant konkrečias programas. Norintiems giliau susipažinti su Paskaliu, rekomenduojame paskaityti knygą Vlado Tumasonio knygą apie Paskalį ir Turbo Paskalį [1].

Medžiagą pateiksime taip, kad ji būtų suprantama dar nesimokiusiam programavimo. Tikimės, kad knyga nebus nuobodi ir tiems, kas jau yra ragavę programavimo: jie ras pažįstamų Paskalio kalbos konstrukcijų bei programavimo sąvokų. Tačiau tikimės, kad pakartojimas nebus nuobodus, nes čia medžiagą pateiksime kiek kitokiu požiūriu, negu buvo mokoma bendrajame informatikos kurse pagrindinėje mokykloje.

Ačiū Sigitai Berušaitytei, Jolantai ir Artūrui Moskvinams už pastebėtas ir ištaisytas klaidas.

Pastabas ir pasiūlymus prašome siųsti adresu: grigas@ktl.mii.lt.

1. TRUMPAI APIE PASKALĮ IR JO PROGRAMAS

Gera pradžia – pusė darbo, – sako liaudies išmintis. Tik kur rasti tą gerą pradžią?

Pažintį su programavimu pradėsime nuo nedidelių, bet veikiančių programų pavyzdžių. Tuos pavyzdžius dėliosime šen ir ten, bandysime suprasti, ką su jais kompiuteris veikia, t.y. kaip kompiuteris atlieka Paskalio programą. Naujų programų dar nerašysime, bet bandysime šiek tiek modifikuoti parašytas programas – programuosime pagal pavyzdžius.

1.1. Paskalis, programuotojas ir kompiuteris

Paskalio kalbą sukūrė žymus informatikas ir pedagogas Virtas (Niklaus Wirth) apie 1970 metus. Paskalis, o taip pat ir kitos N. Virto sukurtos programavimo kalbos (Modula-2, Modula-3, Oberonas-2) pasižymi tuo, kad jos yra universalios, t. y. gerai tinka dažniausiai praktikoje pasitaikančių uždavinių programoms užrašyti. Jos paprastos, logiškos, turi nedaug konstrukcijų, o pačios konstrukcijos yra paprastos ir lengvai įsimenamos. Be to turi gerą apsaugą nuo klaidų. Šios Paskalio savybės naudingos ir besimokančiajam, ir profesionaliam programuotojui. Na, o iš N. Virto sukurtų kalbų pasirinkome Paskalį todėl, kad jis daugiausiai naudojamas ir Lietuvoje, ir užsienyje. Dauguma (virš 70 procentų) pasaulinių informatikos olimpiadų dalyvių programoms rašyti renkasi Paskalio kalbą.

Paskalio žymenys vartojami algoritmams užrašyti. Algoritmus skaito ir nagrinėja žmogus. Taigi Paskalio kalba skiriama ne tik žmogaus bendravimui su kompiuteriu, bet ir žmogaus (programuotojo) bendravimui su kitu žmogumi (programuotoju). Paskalis yra patogi priemonė algoritmavimo idėjoms ir metodams išreikšti, kad su jais galėtų susipažinti kiti bendraminčiai – programuotojai. Todėl į Paskalį reikia žiūrėti visų pirma kaip į algoritminių žymenų sistemą, skirtą žmogui. Informatikoje (konkrečiau algoritmavime ir programavime) jis atlieka analogišką vaidmenį kaip matematiniai žymenys matematikoje, chemijos formulių kalba chemijoje, natų žymenys muzikoje ir pan.

Pateiksime labai paprastą programos, užrašytos Paskaliu, pavyzdį.

```
program vidurkis;
  var a, b, vid: real;
begin
  read(a);
  read(b);
  vid := (a+b)/2;
  writeln(vid: 8: 2)
end.
```

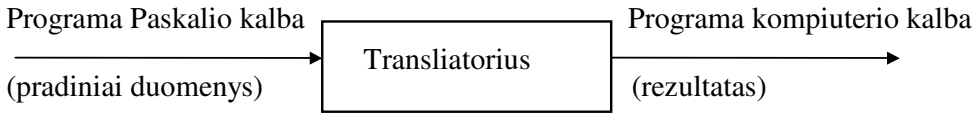
Net ir menkai išmanančiam programavimą nesunku suvokti, kad ši programa skirta dviejų skaičių aritmetiniam vidurkiui skaičiuoti.

Dabar į programą pažvelkime iš kompiuterio pozicijų. Ar kompiuteris ją supras, ar galės įvykdyti, t.y. suskaičiuoti dviejų skaičių vidurkį?

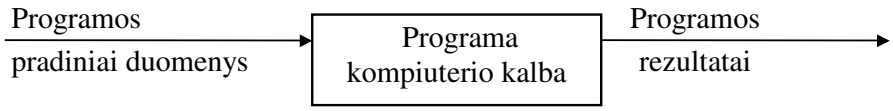
Kompiuteris gali atlikti tiktai tokias programas, kurios sudarytos iš jam suprantamų komandų, kurios žmogui atrodo kaip beprasmis dvejetainių (arba šešioliktainių) skaičių rinkinys.

Kaip įveikti barjerą tarp žmogaus ir kompiuterio, t.y. kaip padaryti, kad kompiuteris suprastų ir galėtų įvykdyti Paskalio kalba parašytą programą?

Reikia programą iš Paskalio kalbos išversti į kompiuterio kalbą. Šį darbą atlieka programa, vadinama transliatoriumi (angl. *translator* – vertėjas) (1 pav.). Išverstą programą kompiuteris jau gali vykdyti (2 pav.). Kompiuteriui pateikę programą, o po to pradinius duomenis (du skaičius), iš kompiuterio gausime rezultatą (vieną skaičių – aritmetinį vidurkį).



1 pav. Transliatorius išverčia programą iš Paskalio kalbos į kompiuterio kalbą



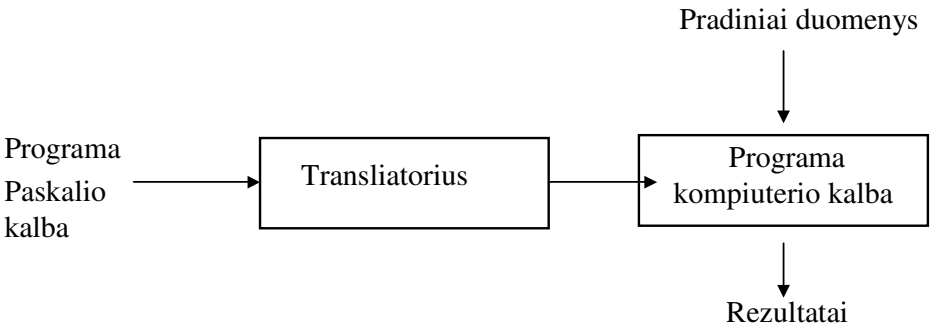
2 pav. Kompiuteris atlieka į jo kalbą išverstą programą

Pastaruoju metu dažniau vartojami kompiliatoriai (angl. *compiler* – kompiliatorius). Kompiliatorius ne tik išverčia programos tekstą iš Paskalio kalbos į kompiuterio kalbą, bet ir į programą įjungia (įkompiliuoja) tam tikras iš anksto parengtas programas arba programų fragmentus.

Paskalio kalbos kompiliatorius yra didelė ir sudėtinga programa. Tačiau nesibaiminkime – jos veikimo nereikia žinoti. Mes būsime tik Paskalio kompiliatoriaus naudotojai. Na, o tiems, kas nori apie kompiliatorių sužinoti daugiau ir giliau – pamatyti jį iš vidaus, galima parekomenduoti Vaivos Grabauskienės knygelę „Susipažinkime – transliatorius“ [2].

Dažnai sakome, kad duomenis paduodame ne kompiuteriui, bet programai ir rezultatus gauname iš programos. Taip kalbėti patogiau. Nors iš tikrųjų veiksmus atlieka kompiuteris, bet atlieka tik tuos, kurie užrašyti programoje. Taigi, visą kompiuterio darbą apsprendžia programa.

1 ir 2 paveiksluose programos vaizduojamos stačiakampiais, o duomenys – rodyklėmis, jungiančiomis programas (į stačiakampį ateinančios rodyklės vaizduoja pradinius duomenis, o išeinančios – rezultatus). Tie patys duomenys vienai programai gali būti pradiniai duomenys (ateinanti rodyklė), kitai – rezultatas (išeinanti rodyklė). Tai, kad sąveikaujant programoms keičiasi duomenų rolė (rezultatai virsta pradiniais duomenimis) savaime suprantama. Tačiau atidžiau panagrinėję minėtus paveikslus galime pastebėti įdomesnį dalyką: duomenys gali būti laikomi programomis ir atvirkščiai – rezultatai programomis. Žmogaus parašytą Paskalio kalba programą transliatorius laiko pradiniais duomenimis ir ją perdirba į kitus duomenis – rezultatą. Kai šis rezultatas pateikiamas kompiuteriui, kompiuteris ją laiko programa ir ima ją vykdyti. Dabar kompiuteriui reikės pradinių duomenų, kurie nurodyti toje programoje (nagrinėtu atveju – dviejų skaičių) ir atiduos programoje numatytus rezultatus. Tai pavaizduota 3 paveiksle.



3 pav. Programos transliavimas ir vykdymas. Transliavimo rezultatas virsta programa

Programavimo terpė. Darbo su programomis ir duomenimis schema pateikta 3 paveiksle, yra paprasta ir akivaizdi. Tačiau dirbti pagal šią schemą būtų nelabai patogu: reikia operuoti daugeliu duomenų ir programų bylų. Programos tekstą Paskalio kalba bei jos pradinius duomenis reikia parašyti su kokiu nors tekstų redaktoriumi. Po to programos tekstą reikia pateikti transliatoriui, iš jo gautą rezultatą (sutransliuotą programą) pateikti kompiuteriui, o kad jis galėtų ją vykdyti – pateikti programos pradinius duomenis, o iš jos gautą rezultatų bylą skaityti vėl su kokiu nors redaktoriumi. Taigi, reikia atlikti daug veiksmų su bylomis. Šie veiksmai ypač juntami, kai programa yra dar tik rašoma, nes tada ją dažnai tenka taisyti, o pataisius visus tuos veiksmus vėl reikia kartoti.

Tam, kad mažiau rūpesčių keltų darbas su bylomis, programuotojui pateikiamas ne vien transliatorius (kompiliatorius), bet visa programuotojui reikalinga terpė – *programavimo sistema*. Transliatorius (kompiliatorius) yra svarbiausias sistemos komponentas. Todėl kartais visa programavimo sistema sutapatinama su transliatoriumi (kompiliatoriumi).

Kitas svarbus programavimo sistemos komponentas yra *programos tekstų redaktorius.* (redaktorius). Tada programos tekstui rinkti bei taisyti nereikia atskiro kokio nors kito redaktoriaus. Pradėjus darbą su sistema, iškart įsijungia redaktorius ir kompiuterio ekrane galima rinkti programos tekstą. Programavimo kalbos redaktoriumi su tekstu galima atlikti tokias pat operacijas, kaip ir kitais redaktoriais: jį rinkti, taisyti, išbraukti, kopijuoti, įterpti į kitą tekstą, įkelti iš bylos, užrašyti į bylą ir pan.

Programavimo terpė paslepia ir patį transliavimo procesą. Ja naudojantis susidaro įspūdis, kad kompiuteris supranta Paskalio kalba parašytą programą ir čia pat ją atlieka.

Uždaviniai

1.1.1. Programą `vidurkis` pakeiskite taip, kad ji apskaičiuotų trijų skaičių aritmetinį vidurkį. Jis skaičiuojamas pagal formulę

(a + b + c) / 3.

1.1.2. Vietoj daugtaškių įterpkite tinkamus žodžius:

Paskalio kalbos sistemos terpėje esančiu … renkami ir taisomi programų tekstai. Paskalio kalbos … arba … išverčia programos tekstą iš … kalbos į … kalbą.

1.2. Paskalio kalbos dialektai

Kompiliatorių autoriai stengiasi programavimo kalbą kiek galima geriau pritaikyti programuotojų poreikiams, t.y. patobulinti. Šitaip atsiranda kalbos variantai arba dialektai. Tai gerai, nes kalba vystosi, tobulėja. Tačiau pasidaro sunkiau susikalbėti skirtingų dialektų naudotojams. Dėl to programavimo kalbos norminamos, apibrėžiami jų standartai. Paskalio kalbos standartas yra kalbos branduolys, į kurį surinktos pačios svarbiausios jos konstrukcijos. Jis yra tarsi orientyras, vienijantis programuojančius įvairiais Paskalio dialektais.

Standartinis Paskalis turi universalų, bet nedidelį konstrukcijų rinkinį, kuriuo patogu programuoti įvairių žmogaus veiklos sričių (matematikos, fizikos, chemijos, ekonomikos ir kt.) uždavinius. Tokie uždaviniai egzistavo visą laiką ir buvo sprendžiami įvairiais tuo metu naudotais kompiuteriais: didelėmis skaičiavimo mašinomis, stovėjusiomis skaičiavimo centruose, pirmaisiais kuklių galimybių mikrokompiuteriais ir šiuolaikiniais IBM PC genties kompiuteriais. Jų algoritmus galima pavadinti klasikiniais. Jų mokomasi mokyklose, jie rašomi informatikos olimpiadose, jais užrašomi įvairių žmogaus veiklos sričių uždavinių matematiniai sprendimai.

Paskalis ir buvo suprojektuotas taip, kad jis kuo geriau tiktų minėtų bendros paskirties uždavinių algoritmams užrašyti ir kuo mažiau priklausytų nuo kompiuterio.

Daugiausiai vartojamas Paskalio kalbos dialektas yra *Turbo Paskalis.* Jis paplito po visą pasaulį ir tapo netgi daugiau žinomas, negu standartinis Paskalis.

Turbo Paskalis turi beveik visas standartinio Paskalio konstrukcijas, o taip pat daugybę naujų konstrukcijų (papildymų). Dalis papildymų suteikia naujų, dažniausiai alternatyvių galimybių klasikiniams algoritmams užrašyti, tačiau daugiausia papildymų skirti kompiuterio įrenginiams valdyti. Jie įgalina programuoti grafinį žmogaus ir kompiuterio dialogą, tiesiogiai prieiti prie duomenų, saugomų kompiuterio atmintinėje ir jais operuoti. Dėl to Turbo Paskalis tinka taip vadinamoms sisteminėms programoms – įvairioms kompiuterio įrenginių tvarkyklėms, įvairiems operacinių sistemų komponentams rašyti.

1.3. Kaip kompiuteris atlieka programą

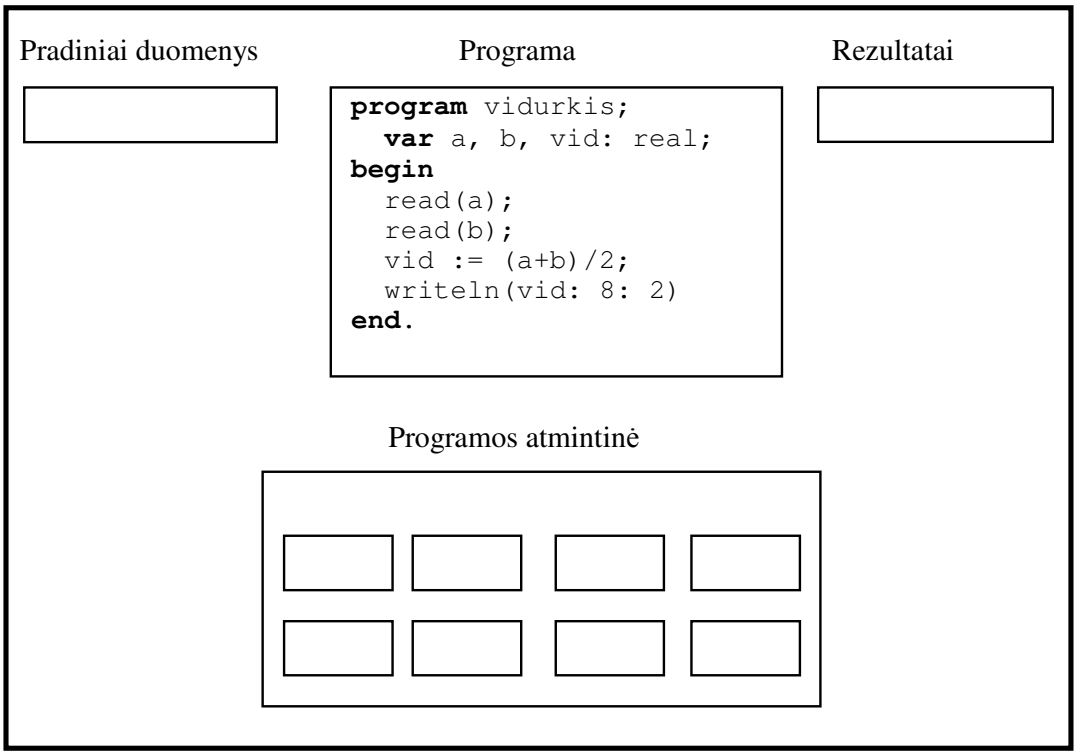
Kompiuteris iš tikrųjų atlieka į jo kalbą išverstą programą. Tačiau mums patogiau kompiuterio darbą sieti su Paskalio programa. Iš tikrųjų, ir Paskalio programa, ir išversta programa atlieka tuos pačius veiksmus. Todėl nagrinėjant veiksmus nesvarbu, kokia kalba tie veiksmai užrašyti. Todėl pasirenkame mums suprantamesnį variantą – programą, užrašytą Paskalio kalba.

Kompiuteris atlieka programoje užrašytas operacijas su duomenimis, saugomais jo atmintinėje (tiksliau atmintinėje, skirtoje programai). Kaip tie duomenys atsiranda atmintinėje ir kaip kompiuteris atskiria vienus duomenis nuo kitų?

Kompiuterio atmintinę galima įsivaizduoti kaip popieriaus lapą, arba dar geriau – klasės lentą, nes joje galima ištrinti nebereikalingus duomenis ir į jų vietą rašyti naujus. Atmintinė suskirstyta į daugybę langelių duomenims rašyti. Tam tikra jos dalis skiriama programos duomenims saugoti. Kai programa pradedama vykdyti, joje duomenų dar nėra.

Pasekime, kaip kompiuteris atlieka programą.

Jis skaito programą ir atlieka joje užrašytus veiksmus. Panagrinėkime jau pažįstamą aritmetinio vidurkio skaičiavimo programą: ką reiškia užrašai (t.y. kaip ją supranta žmogus) ir kaip ją atlieka (t.y. supranta) kompiuteris (4 pav.).



4 pav. Kompiuteris pasiruošęs atlikti programą

Pirmoji programos eilutė

```
program vidurkis;
```

yra jos antraštė. Ji prasideda žodžiu **program.** Tai bazinis Paskalio kalbos žodis. Baziniai žodžiai Paskalio programose turi griežtai nustatytą prasmę (apie juos kalbėsime 1.5 skyr.).

Toliau einantis žodis `vidurkis` yra programos vardas. Kaip vadinti programą, sugalvoja programuotojas. Jis gali programą pavadinti bet kokiu vardu. Kompiuteriui programos vardas nerūpi. Svarbu tik, kad jis būtų sudarytas taisyklingai. Apie vardų sudarymo taisykles kalbėsime 1.5 skyrelyje.

Antraštė yra svarbi žmogui, kad jis vieną programą atskirtų nuo kitos. Tuo tarpu kompiuteriui ji nenurodo jokių veiksmų.

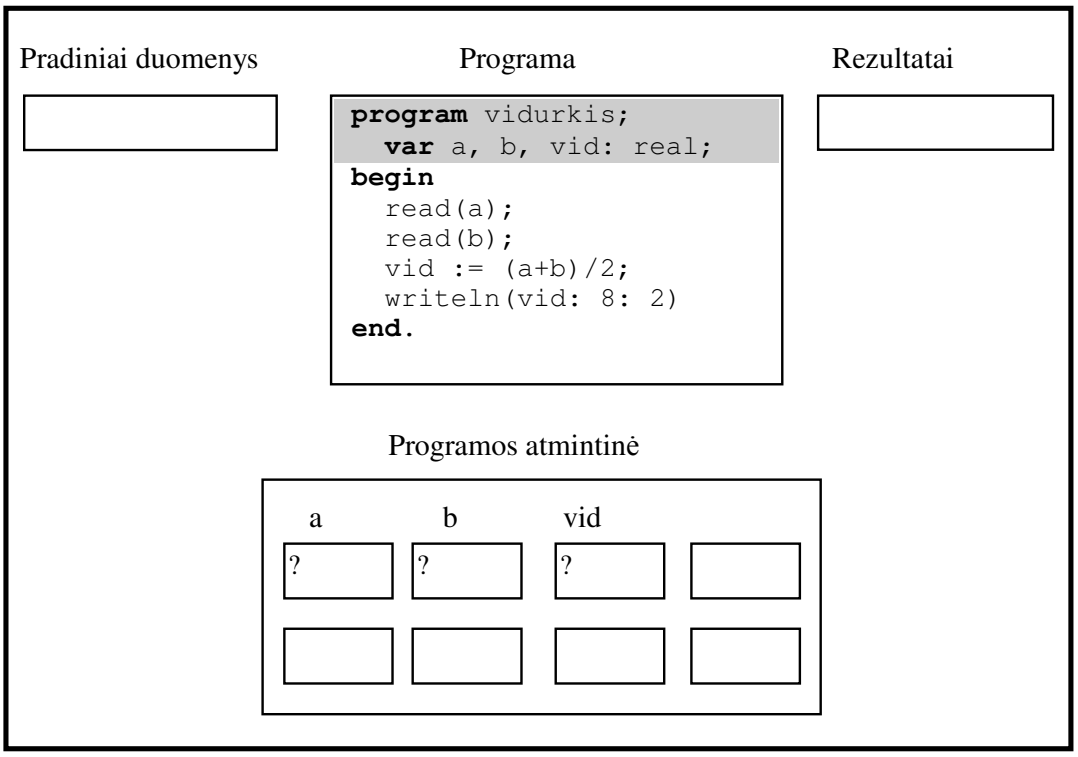
Antroji eilutė:

```
var a, b, vid: real;
```

yra *kintamųjų aprašas*. Apie tai pasako bazinis žodis **var**, kuris yra angliško žodžio *variable* (*kintamasis*) santrumpa.

Po žodžio **var** išvardijami kintamųjų, kurie bus naudojami programoje, vardai. Vardus parenka programuotojas. Kintamieji žymi duomenis. Programoje duomenys nurodomi vardais, o kompiuterio atmintyje saugomos tų duomenų reikšmės. Duomenys gali būti įvairūs: sveikieji skaičiai, realieji skaičiai, tekstai ir t.t. Su skirtingo tipo duomenimis atliekamos skirtingos operacijos, skirtingo tipo duomenims reikia skirtingo vietos kiekio kompiuterio atmintinėje. Todėl reikia žinoti, kokio tipo reikšmes galės įgyti kintamasis ir yra nurodomas *kintamųjų tipas*. Žodis `real` pasako, kad prieš jį išvardyti kintamieji žymi realiuosius skaičius. Vadinasi, kintamieji `a`, `b` ir `vid` galės įgyti tik realiųjų skaičių reikšmes, o kiekvienam jų skiriama vietos atmintinėje tiek, kiek reikia vienam realiajam skaičiui įrašyti.

Su aprašais turi darbo ir kompiuteris: jis paskiria vietą atmintinėje kintamųjų reikšmėms saugoti. 5 paveiksle pavaizduota situacija, kai kompiuteris perskaitė dvi pirmąsias programos eilutes (perskaitytos ir apdorotos programos eilutės paveiksle patamsintos). Atmintinės vietos, paskirtos kintamiesiems, sužymėtos kintamųjų vardais. Į langelius rašysime kintamųjų reikšmes. Tik ką aprašytų kintamųjų reikšmės pažymėtos klaustukais. Tai reiškia, kad ten gali būti užsilikę nežinomi duomenys iš prieš tai veikusių programų. Šie nežinomi duomenys bus ištrinti, kai kintamiesiems bus priskiriamos reikšmės.



5 pav. Kompiuteris paskyrė vietas atmintinėje kintamųjų reikšmėms saugoti

Vietos atmintinėje skyrimas yra pagalbinis veiksmas. Tikrųjų veiksmų pradžią rodo žodis **`begin`**. Toliau aprašomi veiksmai. Veiksmų užrašai vadinami *sakiniais*.

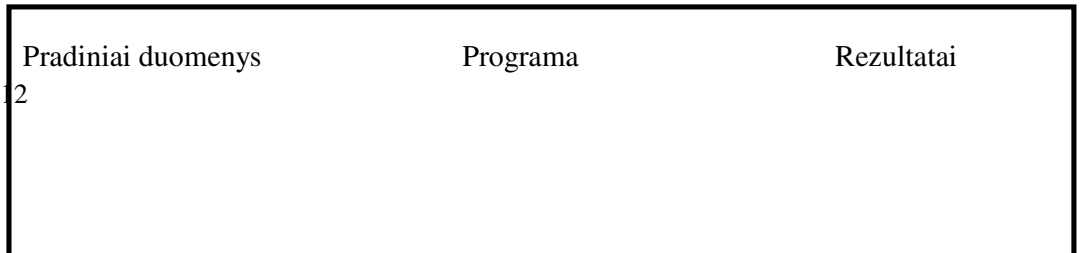
Operacijas kompiuteris atlieka su duomenimis, esančiais jo atmintinėje. Pradžioje programai skirta atmintinė tuščia. Todėl pirmiausia reikia įvesti (skaityti) pradinius duomenis, arba bent dalį jų – kad būtų ką veikti.

Pirmasis sakinys

```
read(a)
```

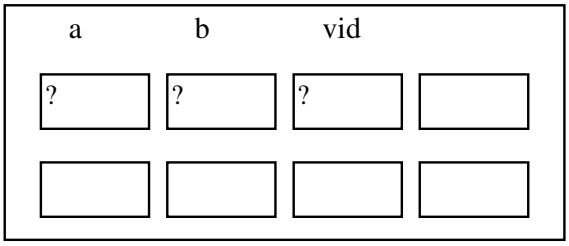
rodo, kad reikia skaityti pradinį duomenį iš klaviatūros ir jį priskirti kintamajam `a`. Kompiuteris, perskaitęs šį sakinį, programos vykdymą pristabdo ir laukia, kol klaviatūra surinksime skaičių. Tarkime, kad surinkome skaičių 22,6 (6 pav.).

Renkamo skaičiaus trupmeninę dalį nuo sveikosios reikia skirti tašku, o ne kableliu, kaip priimta matematikoje. Toks nukrypimas atsirado dėl to, kad pirmosios programavimo kalbos buvo projektuojamos JAV, o amerikiečiai vietoj kablelio rašo tašką.



22.6	**program** vidurkis;
	  var a, b, vid: real;
	begin
	  read(a);
	  read(b);
	  vid := (a+b)/2;
	  writeln(vid: 8: 2)
	end.	

Programos atmintinė



6 pav. Surinktas, bet dar neperskaitytas pirmasis pradinis duomuo

Kaip pasakyti kompiuteriui, kad skaičius jau surinktas ir jį jau galima skaityti?

Reikia paspausti įvesties klavišą. Tada kompiuteris perskaito skaičių, esantį pagalbinėje klaviatūros atmintinėje ir jį įrašo į kintamojo `a` reikšmei saugoti skirtą tikrosios atmintinės vietą (7 pav.).

Analogiškai atliekamas sakinys

```
read(b)
```

Antrasis pradinis duomuo priskiriamas kintamajam `b` (8 ir 9 pav.). Tarkime, kad tai buvo skaičius 12,2.

Toliau eina prieskyros sakinys

```
vid := (a+b)/2
```

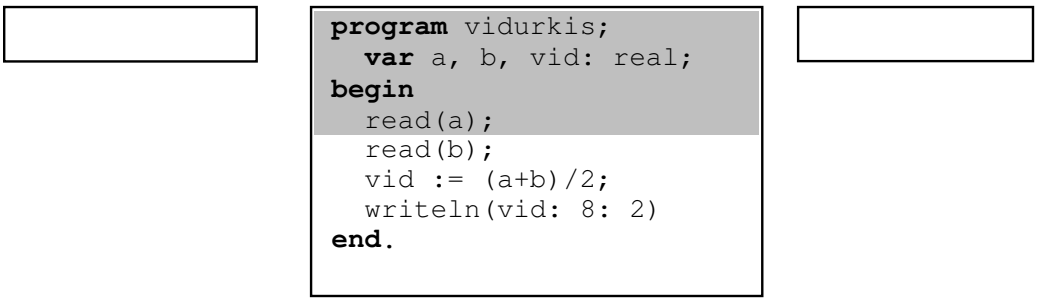
Jis sako, kad reikia apskaičiuoti dešinėje prieskyros ženklo := pusėje esančio reiškinio reikšmę (sudėti du skaičius ir gautą sumą padalyti iš dviejų) ir gautą reikšmę priskirti kintamajam `vid`, t.y. įrašyti į kintamajam `vid` skirtą vietą atmintinėje.

Situacija, gauta atlikus šį sakinį, pavaizduota 10 paveiksle.

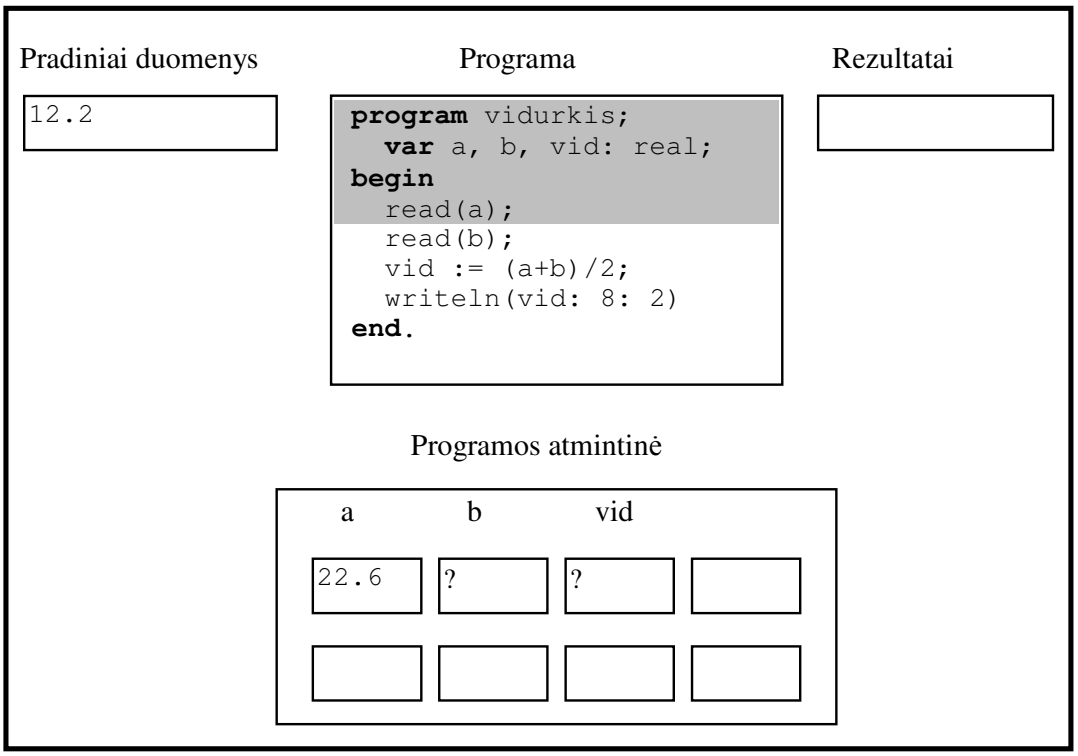
Liko dar vienas sakinys (veiksmas)

```
writeln(vid: 8: 2)
```

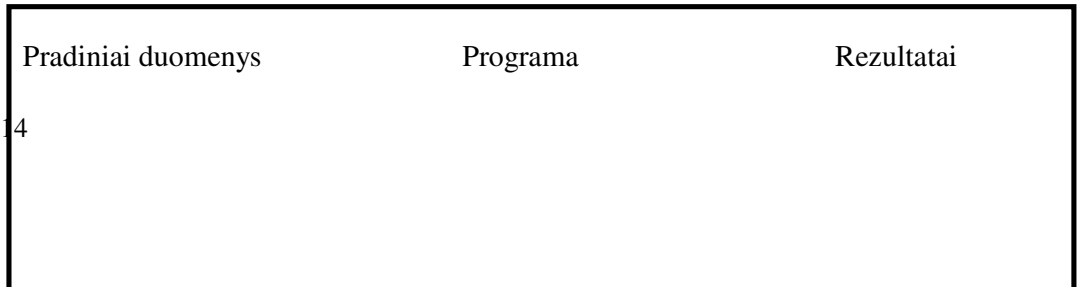
Pradiniai duomenys	Programa	Rezultatai

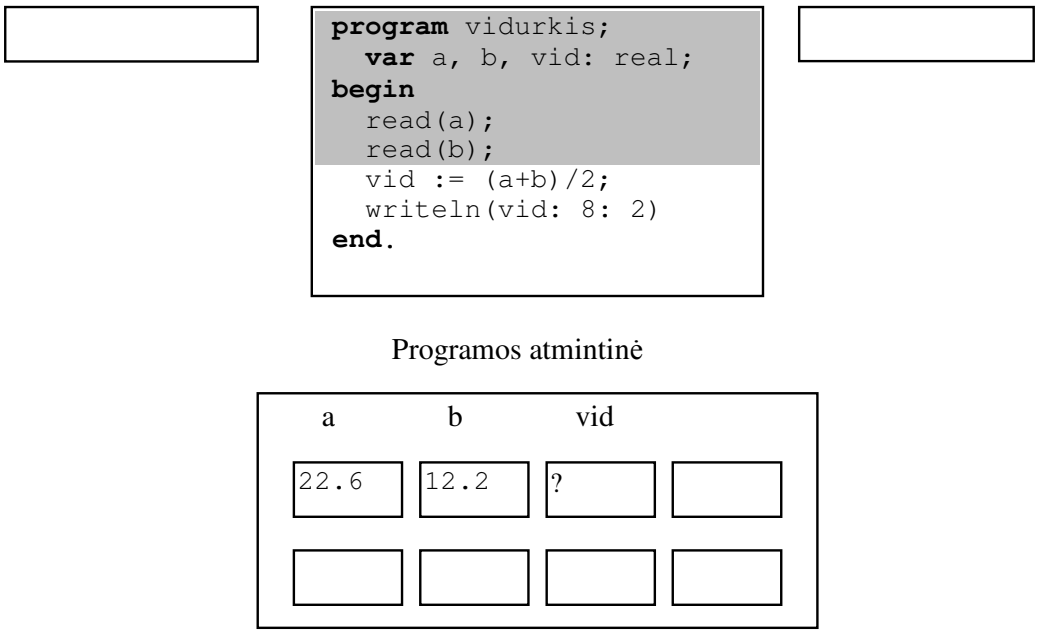


7 pav. Perskaitytas pirmasis duomuo – skaičius 22,6 ir įrašytas į programai skirtą atmintinę (kintamojo `a` vietą)

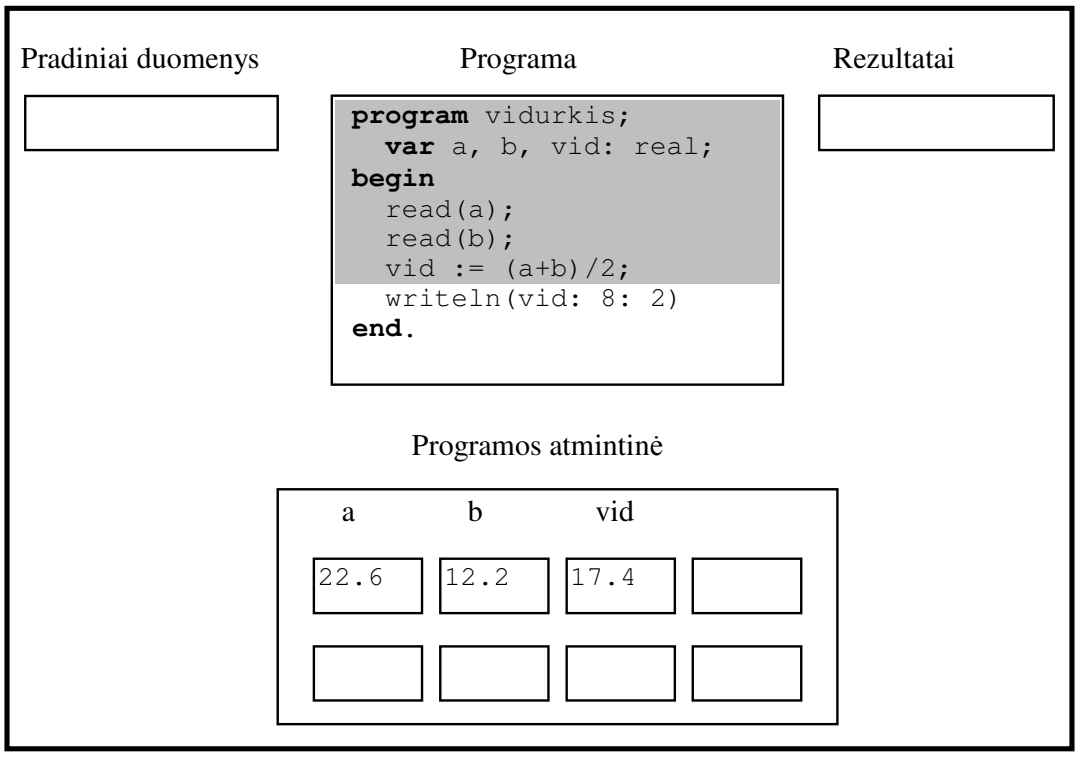


8 pav. Surinktas antrasis duomuo – skaičius 12,2, bet dar neperskaitytas ir neįrašytas į atmintinę





9 pav. Perskaitytas antrasis duomuo – skaičius 12,2 ir įrašytas į atmintinę (kintamajam `b` skirtą vietą)



10 pav. Apskaičiuota reiškinio `(a+b)/2` reikšmė ir priskirta kintamajam `vid`

Juo kompiuteriui sakoma, kad reikia į vaizduoklio ekrano rezultatų langą įrašyti kintamojo `vid` reikšmę.

Situacija, gauta atlikus šį sakinį, pavaizduota 11 paveiksle.

Skaičiai 8 ir 2 rašymo sakinyje vadinami rašymo formatais. Pirmasis skaičius parodo, kiek vietos (kiek pozicijų) reikia skirti skaičiui, o antrasis – kiek skilčių po kablelio reikia parašyti. Dėl to rezultatas (11 pav. dešinėje viršuje) pavaizduotas su dviem ženklais po kablelio, nors šiuo atveju pakaktų ir vieno.

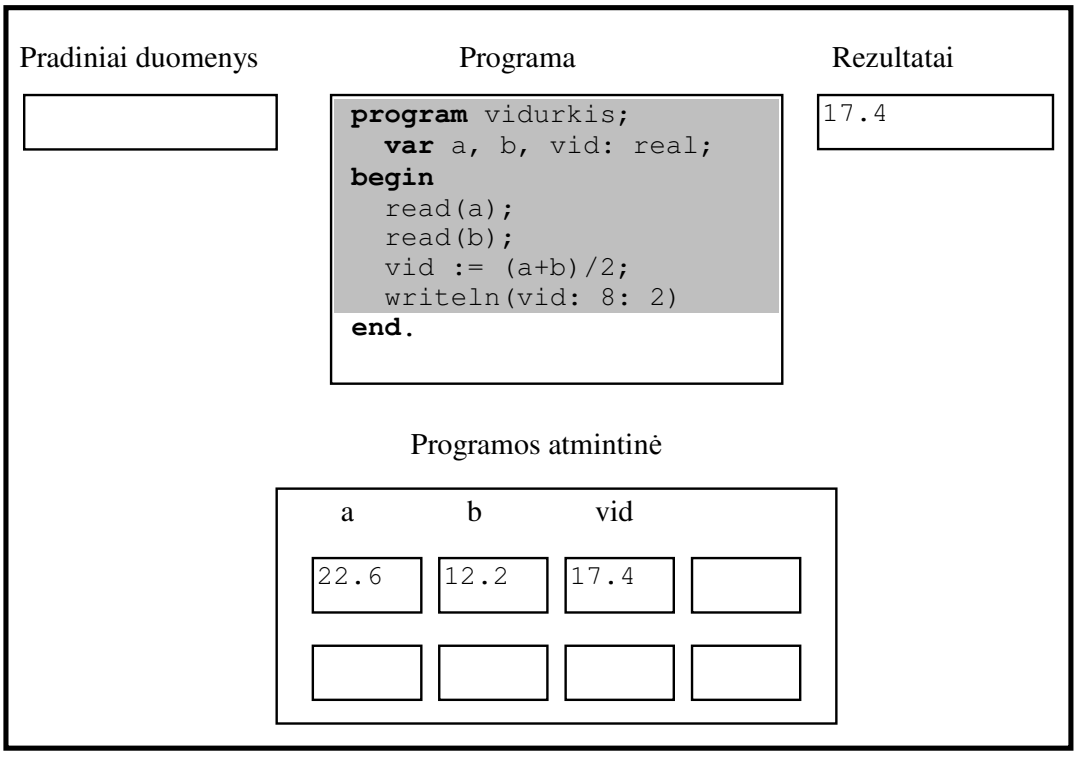
Sakiniai skiriami kabliataškiais. Todėl reikia dėti kabliataškius tarp greta einančių sakinių, bet nebereikia kabliataškio po paskutinio sakinio (prieš **`end`**) – nebėra ką nuo jo atskirti. Nebus klaidos, jeigu ten ir padėsime kabliataškį: tada bus laikoma, kad po kabliataškio eina dar vienas – tuščias sakinys, neatliekąs jokio veiksmo.

Paskutinė programos eilutė

`end`.

pasako kompiuteriui, kad reikia baigti programą.

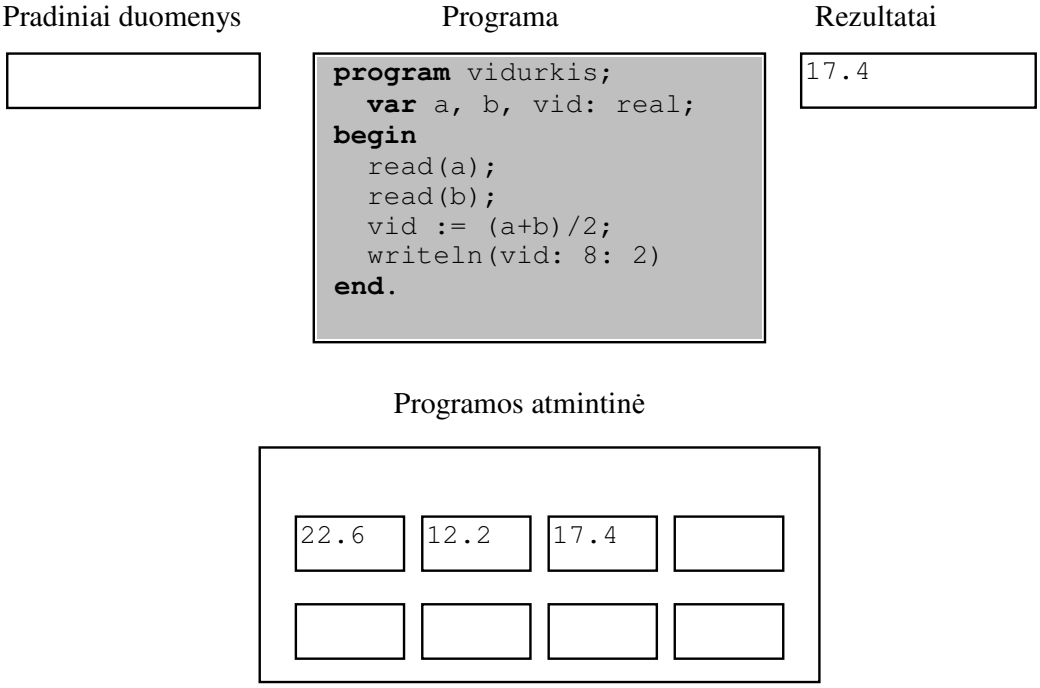
Kai programa baigiama, nutrūksta kintamųjų sąsajos su jų reikšmėmis, kurios saugomos atmintinėje (12 pav.). Viskas, kas buvo surašyta programai skirtoje atmintinėje, tampa pamestais ir nebepasiekiamais duomenimis. Jeigu programą paleistume iš naujo, tiems patiems kintamiesiems galėtų būti paskirtos kitos vietos atmintinėje. Taigi, programai baigus darbą, jos kintamųjų reikšmės iš tikrųjų dingsta. Išlieka tik tie rezultatai, kurie buvo įrašyti į vaizduoklio ekraną arba į bylas (apie tai kalbėsime 2.5–2.7 skyr.). Štai todėl reikia nepamiršti veiksmų, kur nors įrašančių gautus rezultatus.



11 pav. Kintamojo `vid` reikšmė (skaičius 17,4) įrašyta į rezultatų langą vaizduoklio ekrane

Štai šitaip devyniais paveikslėliais, primenančiais devynis filmo kadrus, parodėme, kokius veiksmus atlieka kompiuteris vykdydamas labai paprastą programą.

Jeigu programai `vidurkis` pateiktume kitus pradinius duomenis, gautume kitą rezultatą. Tačiau bet kuriuo atveju tai būtų dviejų pateiktų skaičių aritmetinis vidurkis.



12 pav. Kai programa baigia darbą, jos duomenų atmintinė lieka tuščia

Visur kalbėjome tik apie duomenų rašymą į atmintinę. Niekur neužsiminėme apie duomenų ištrynimą. Ta atmintinės vieta, į kurią rašomi nauji duomenys, visada automatiškai išvaloma (ištrinami seni duomenys). Taigi nebereikalingi duomenys išlieka atmintinėje iki tol, kol į jų vietą neužrašomi nauji. Iš tikrųjų kompiuterio atmintinėje išliko ir vidurkio skaičiavimo pradiniai duomenys ir rezultatai programai baigus darbą. Tačiau nebeišliko sąsajos su kintamaisiais. Todėl tų duomenų nebegalima rasti.

Uždaviniai

1.3.1. Ar teisinga šitokia aritmetinio vidurkio skaičiavimo programa?

```
program vidurkis3;
  var a, vid: real;
begin
  read(vid);
  read(a);
  vid := (vid + a)/2;
  writeln(vid: 8: 2)
end.
```

1.3.2. Ką parašytų programos `KasBus1` ir `KasBus2`, jeigu joms pateiktume tuos pačius pradinius duomenis: 11 ir 255?

```
program KasBus1;
  var a, b:integer;
begin
  read(a);
  read(b);
  a := b;
  b := a;
  writeln(a, b: 4)
```

```
end.
```

```
program KasBus2;
  var a, b, t: integer;
begin
  read(a);
  read(b);
  t := a;
  a := b;
  b := t;
  writeln(a, b: 4)
end.
```

1.3.3. Ką parašys šitokia programa?

```
program laipsnis;
  var a: integer;
begin
  a := 2;
  a := a*a;
  a := a*a;
  a := a*a;
  writeln(a)
end.
```

Praktikos darbas

1.3.1. Kėlimas šimtuoju laipsniu. Parašykite programą, kuri pradinį duomenį – realųjį skaičių pakeltų šimtuoju laipsniu. Paskalio kalba kėlimo laipsniu operacijos neturi. Todėl kėlimui laipsniu naudokite daugybą. Programą parašykite tokią, kad būtų panaudotas mažiausias daugybos operacijų kiekis (pakanka 8 daugybos operacijų).

1.4. Dialogas su kompiuteriu

Gera programa turi būti ne tik teisinga ir suprantama programuotojui, bet ir patogi naudotojui. Naudotojas dažniausiai nėra programuotojas. Jis neskaito programos teksto ir nežino, kas jame parašyta. Todėl apie viską, ką jis turi daryti paleidęs programą, turi jam pasakyti kompiuteris. O kompiuteris pasakys tik tada, kai programoje bus parašyta, ką jis turi pasakyti. Tai reiškia, kad programoje turi būti užrašyti ne tik uždavinio sprendimo veiksmai, bet ir dialogo veiksmai su tuo programos naudotoju – žmogumi, kuris sprendžia uždavinį naudodamasis kompiuteriu ir to uždavinio programa.

Su dialogo trūkumu jau susidūrėme atlikdami programą `vidurkis`. Jeigu programą atlikome Turbo Paskalio terpėje, tai kompiuteris nieko nepranešė apie tai, kad jis laukia pradinių duomenų, nepasakė, kokie duomenys turi būti ir kiek jų turi būti. Sustojo ir tiek. Tam, kad būtų aišku, kokių žmogaus veiksmų laukia kompiuteris, programą *`vidurkis`* papildysime dialogo veiksmais.

1 pavyzdys.

```
program vidurkis;
  var a, b, vid: real;
begin
  write('Surinkite skaičių: ');
  read(a);
```

```
  write('Surinkite kitą skaičių: ');
  read(b);
  vid := (a+b)/2;
  writeln(vid: 8: 2)
end.
```

Dabar kompiuteris taip pat sustoja laukdamas pradinio duomens. Tačiau prieš sustodamas jis atliko sakinį

```
write('Surinkite skaičių: ')
```

pagal kurį į ekraną parašė tekstą

```
Surinkite skaičių:
```

Jį matydami mes jau žinome ką daryti ir renkame skaičių. Kai paprašo surinkti kitą skaičių – jį surenkame.

Panašūs pranešimai ypač reikalingi, kai programai reikia pateikti daugiau pradinių duomenų ir kai tie duomenys yra įvairūs.

2 pavyzdys. Pateiksime programą, kuri apskaičiuoja, kiek pinigų turėsite banke po `mn` mėnesių, jeigu dabar padėjote terminuotą `prad` Lt indėlį, o bankas moka `proc` procentų metinių palūkanų ir apskaičiuoja jas laikydamas, kad visi mėnesiai lygūs, t.y. kiekvienas jų sudaro 1/12 metų dalį.

```
program pinigai;
  var prad: real;
      proc: real;
      mn: integer;
      galut: real;
begin
  write('Kokią sumą padėjote į banką? Lt: ');
  read(prad);
  write('Kokias metines palūkanas moka bankas? %: ');
  read(proc);
  write('Kiek mėnesių laikysite pinigus banke? ');
  read(mn);
  galut := prad + prad*(proc/100*mn/12);
  writeln('Pasibaigus terminui turėsite Lt:', galut: 10: 2)
end.
```

Paleidę programą iš karto matome tekstą

```
Kokią sumą padėjote į banką? Lt:
```

Dabar aišku, kurį skaičių reikia rinkti. Jį surinkę ir paspaudę įvesties klavišą, matome naują pranešimą:

```
Kokias metines palūkanas moka bankas? %:
```

ir t.t.

Rezultatą sudaro ne vien skaičius, bet ir tekstas, paaiškinantis, ką tas skaičius reiškia.

Atkreipiame dėmesį, kad dialogo tekste nevartojami kintamųjų vardai, o duomenys aprašomi žodžiais taip, kaip mes juos vadiname kasdieniniame gyvenime. Taip daroma dėl to, kad programos naudotojas nemato programos teksto, ir nežino, kas kokiu vardu pavadinta. Programos tekstą matome ir gerai žinome tik mes, programuotojai, kol jį rašome. O programos naudotojui programos tekstas nerūpi.

Dialogo veiksmus lengva programuoti. Faktiškai reikia rašyti klausimus, kuriuos pateikia kompiuteris žmogui. Tačiau jų rašymas reikalauja kruopštumo ir atidumo – tekstus reikia estetiškai išdėstyti, kad jie žmogui būtų ne tik naudingi, bet ir būtų malonu į juos žiūrėti. Dar daugiau darbo reikia įdėti, kai tekstas įrėminamas arba pateikiamas meniu pavidalu.

Tuo tarpu uždavinio sprendimo veiksmai yra sudėtingesni ir reikalauja daugiau galvojimo. Todėl pagrindinį dėmesį ir kreipsime į juos. Na, o dialogo veiksmus dažniausiai praleisime, nes jų tekstai būna ilgi ir užtemdytų pagrindinius uždavinio veiksmus.

1.5. Iš ko daroma programa: Paskalio kalbos leksika

Bet kurios gyvosios kalbos (pvz., lietuvių) sakinį sudaro žodžiai ir skyrybos ženklai, sutvarkyti pagal tos kalbos gramatikos (sintaksės) taisykles. Panašiai galima pasakyti ir apie programavimo kalbos (pvz., Paskalio) programą. Tiktai jos sudėtinių dalių, vadinamų leksemomis, yra daugiau. Paskalio kalbos leksemų klasifikacija pateikta 13 paveiksle.

Operacijų ir skyrybos ženklai Paskalio kalboje vartojami tik tokie, kuriuos turi bet kuris kompiuteris. Todėl jų nedaug. Štai jie:

```
+ - * / = < > . , : ; ( ) [ ] { } ' ^
```

Šių ženklų nepakanka. Todėl kai kurios operacijos užrašomos ženklų poromis:

<> nelygu (≠);

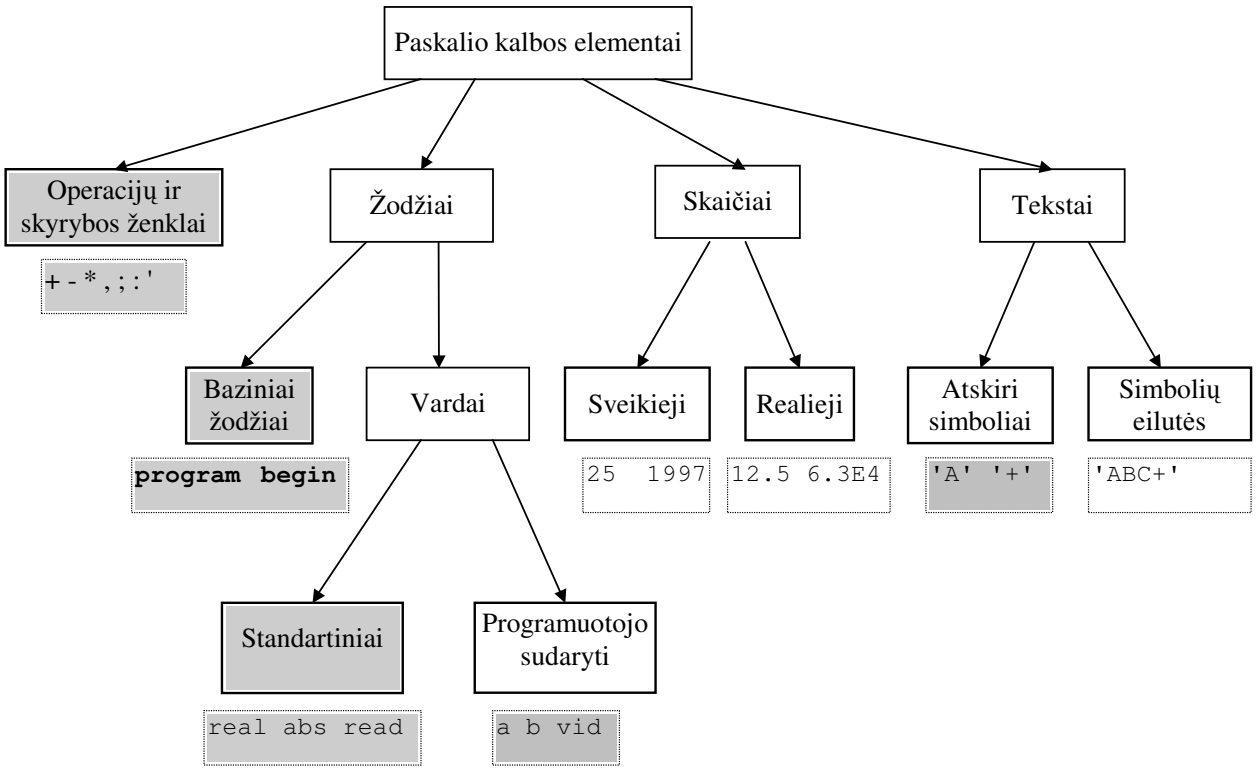
<= nedaugiau (≤);

>= nemažiau (≥);

:= prieskyra;

.. intervalas.

Dar kiti ženklai išreiškiami baziniais žodžiais.



13 pav. Paskalio kalbos leksikos elementų klasifikacija. Punktyriniuose stačiakampiuose pateikti leksemų pavyzdžiai. Užtamsintuose stačiakampiuose pateiktos tos leksemos, kurios apibrėžtos

Paskalio kalboje. Jų yra nedaug, jas galima išvardyti. Kitas leksemas sudaro programuotojas iš kompiuterio abėcėlės simbolių; jų galima sudaryti tiek, kiek reikia (be galo daug).

Pastaba. Paskalio kalbos standarte vartojamas simbolis ↑. Kadangi jo neturi asmeniniai kompiuteriai, tai jų kompiliatoriuose šis simbolis pakeistas simboliu ^. Ankstesni kompiuteriai neturėjo simbolių `{}`. Todėl vietoj jų galima vartoti ir simbolių poras `(* *)`.

Bazinis žodis programoje laikomas vienu simboliu. Galima sakyti, kad baziniai žodžiai – tai Paskalio kalbos simboliai, kurių nėra kompiuterio abėcėlėje ir todėl užrašomi žodžiais. Baziniai žodžiai vartojami operacijoms bei kitoms programų konstrukcijoms žymėti. Visi baziniai žodžiai yra *rezervuoti*, t.y. jie turi apibrėžtą prasmę ir kitiems tikslams jų vartoti negalima. Taigi negalima sudaryti naujų vardų, sutampančių su baziniais žodžiais.

Kad baziniai žodžiai būtų lengviau pastebimi, jie paryškinami – spausdinami pusjuodžiu šriftu (pvz., **`begin`**), o Paskalio sistemų ekrane rodomame tekste jie dažniausiai kitaip nuspalvinami. Paskalis turi 35 bazinius žodžius.

Vardai. Programas vadiname vardais. Kintamuosius – taip pat vardais. Apskritai, jeigu programoje yra aprašomas koks nors objektas (kintamasis, duomenų tipas, funkcija, procedūra ir kt.), kurį vėliau reikia paminėti (įvardinti), jam suteikiamas vardas. Visi vardai, nesvarbu ką jie žymėtų, sudaromi iš raidžių ir skaitmenų pagal tas pačias taisykles:

- pirmasis vardo simbolis turi būti raidė;
- tolesni vardo simboliai turi būti raidės arba skaitmenys;
- vardas neturi sutapti su baziniu žodžiu;
- vardo viduje negali būti tarpų;
- vardo ilgis formaliai neribojamas, tačiau vardas turi tilpti į vieną eilutę (jis negali būti keliamas į kitą eilutę).

Vardų pavyzdžiai.

Teisingi:

```
a                p25              a22krc7     žodis
Jonas            vardas           m1998       raidėA
suma             integer          x           ABBA
IlgasVardas
DarIlgesnisVardas
```

Klaidingi:

`7a`	pirmasis vardo simbolis turi būti raidė;
`eil'nr`	vardas sudaromas tik iš raidžių ir skaitmenų;
`eil nr`	vardo viduje negali būti tarpų;
`begin`	vardas negali sutapti su baziniu žodžiu.

Didžiosios ir mažosios raidės varduose laikomos vienodomis. Todėl vardai

```
SUMA  suma  Suma  SuMa
```

laikomi sutampančiais.

Didžiųjų ir mažųjų raidžių vienodumu varduose nereikėtų naudotis, nes matydami du užrašus, parašytus tomis pačiomis, bet skirtingo lygio raidėmis, esame linkę juos laikyti skirtingais. Juo labiau, kad kai kuriose kitose programavimo kalbose bei operacinėse sistemose didžiosios ir mažosios raidės varduose laikomos skirtingomis. Jų vienodumas Paskalyje yra laikytinas ankstesnių kompiuterių, turėjusių tik vieno lygio (dažniausiai didžiąsias) raides, reliktu, o ne programavimo patogumu.

Turbo Paskalio varduose gali būti vartojamas pabraukimo simbolis _. Jis (panašiai, kaip ir skaitmuo) negali būti pirmasis vardo simbolis.

Pavyzdžiai:

```
eil_nr          Ilgas_vardas
m_1998          dar_ilgesnis_vardas
```

Šiuolaikinėse programavimo kalbose (Adoje, C++ (standartinėje), Komponentiniame Paskalyje ir kt.) varduose galima vartoti bet kurios abėcėlės raides, tarp jų ir specifines lietuviškas (pvz., ą, č). Turbo Paskalio transliatoriaus paskutinė versija yra išleista maždaug prieš 10 metų ir šios galimybės neturi. Paskalio programų varduose vartosime visas lietuviškas raides, kadangi tokius vardus atpažįsta naujesni transliatoriai, o taip ir redaktoriaus „Word“ sąsaja su Turbo Paskaliu (t.y., visas šioje knygoje pateiktas programas galima atlikti su Turbo Paskaliu, paleidžiamu iš redaktoriaus „Word“).

Visi vardai, nesvarbu ką jie žymėtų, sudaromi iš raidžių ir skaitmenų pagal tas pačias taisykles. Todėl kiekvieną naujai sugalvotą vardą reikia aprašyti. Apraše pasakoma vardo paskirtis – nurodoma, kokios rūšies objektą (konstantą, kintamąjį, duomenų tipą, funkciją, procedūrą, modulį) tas vardas žymi. Kaip vardai aprašomi, nagrinėsime vėliau, kalbėdami apie tais vardais žymimus objektus.

Yra vardų, kurie įtraukti į Paskalį. Tai tarsi paties Paskalio „sugalvoti“ vardai. Jie vadinami standartiniais arba integruotais.

Integruotų vardų pavyzdžiai:

```
integer  real  read  writeln  maxint
```

Kuo skiriasi integruoti vardai nuo bazinių žodžių?

Su integruotu vardu gali sutapti programuotojo sudarytas vardas. Tokiu atveju klaidos nebus. Tiktai ten, kur galioja programuotojo sudarytas vardas, nebus galima panaudoti integruoto – jį nustelbs programuotojo aprašytas vardas. Todėl integruotų vardų galima neprisiminti. Jeigu programuotojas nežino integruoto vardo, tai, aišku, jo ir nevartos. Tai nieko blogo, kad jis bus nustelbtas. Tuo tarpu baziniai žodžiai yra rezervuoti ir jeigu bandysime sudaryti vardą, sutampantį su baziniu žodžiu, bus klaida.

Skaičiai. Paskalyje vartojami dviejų rūšių skaičiai: *sveikieji* ir *realieji*.

Sveikieji skaičiai sudaromi iš skaitmenų. Neigiamo skaičiaus pradžioje rašomas minusas. Teigiamo skaičiaus pradžioje galima rašyti pliusą. Bet jo galima ir nerašyti – taip paprastai ir daroma.

Pavyzdžiai:

```
1998        +667
0           -42
```

Neteisingai užrašytų skaičių pavyzdžiai:

```
300 000 000      skaičiaus viduje negali būti tarpų,
300.000.000      sveikojo skaičiaus viduje negali būti kitokių ženklų, išskyrus
```
skaitmenis.

Realieji skaičiai užrašomi dviem būdais.

1. Dešimtaine trupmena. Skaičiai rašomi taip, kaip ir dešimtainės trupmenos matematikoje, išskyrus tai, kad trupmeninė dalis nuo sveikosios skiriama tašku (ne kableliu), pavyzdžiui:

```
643.25
  3.1415926536
-20.0
```

Ženklas (+ arba -) rašomas prieš skaičių be tarpo. Tačiau jeigu ir paliktume tarpą, tai klaidos nebūtų. Tik toks užrašas reikštų atimties operaciją (žr. 2.2 skyr).

2. Rodikliniu pavidalu. Dešimtainės trupmenos užrašas papildomas daugikliu 10^n. Paskalyje (o taip pat ir kitose programavimo kalbose) jis rašomas kitaip, negu matematikoje. Vietoj laipsnio pagrindo 10 rašoma raidė `E` (arba `e`) ir po jos – sveikuoju skaičiumi išreikštas laipsnio rodiklis *n*, nepakeltas į viršų. Pateiksime realiųjų užrašų skaičių pavyzdžių matematikoje ir programavime:

643,25	`643.25E0`
$6{,}4325\times10^2$	`6.4325E2`
64325×10^{-2}	`64325E-2`

Jeigu skaičiaus užrašas turi bent vieną realiojo skaičiaus požymį – trupmeninę skaičiaus dalį skiriantį tašką arba raidę `E` – jis laikomas realiuoju, priešingu atveju – sveikuoju.

Atkreipiame dėmesį, kad programavime sveikieji ir realieji skaičiai sudaro atskiras, nesikertančias aibes (t.y. sveikieji skaičiai nėra realiųjų skaičių poaibis). Todėl reikia griežtai skirti sveikuosius skaičius nuo realiųjų.

Tekstas. Duomenys gali būti ne tik skaičiai, bet ir tekstai: atskiri simboliai arba jų eilutės.

Simbolis yra mažiausias teksto elementas. Tai bet kuris kompiuterio abėcėlės simbolis – raidė, skaitmuo, skyrybos ženklas, operacijos ženklas ir pan. Tam, kad būtų aišku, jog programoje parašytas simbolis yra teksto simbolis, jis rašomas tarp apostrofų, pavyzdžiui,

```
'A'  'Ą'  'B'  '5'  ';'  '>'  ' '
```

Paskutinis čia užrašytas yra tarpo simbolis.

Čia tiesūs vertikalūs apostrofai yra Paskalio abėcėlės simboliai, atliekantys skyrybos ženklų vaidmenį.

Apostrofas taip pat yra ir kompiuterio abėcėlės simbolis. Vadinasi, jis gali būti panaudotas ir tekste. Kaip jį užrašyti?

Apostrofo simbolis vaizduojamas dviem apostrofais. Taigi, kartu su simbolį ribojančiais apostrofais gauname šitokį užrašą:

```
''''
```

Simbolių eilutės. Vienas simbolis – per mažas tekstinės informacijos kiekis. Dažnai patogiau operuoti su didesniais jos kiekiais (pvz., žodžiais, sakiniais, sakinio fragmentais). Todėl beveik visuose Paskalio dialektuose naudojama dar vienas tekstinių duomenų tipas –*simbolių eilutė.* Tai simbolių seka, užrašyta tarp apostrofų.

Pavyzdžiai:

```
'ABBA'
'Čia yra eilutė'
'1999.12.31'
''
```

Svarbi eilutės charakteristika yra jos ilgis. Tai teksto simbolių skaičius. Pirmoji eilutė yra keturių simbolių ilgio, antroji – 14 simbolių (tarpai taip pat simboliai). Paskutinės eilutės ilgis lygus nuliui. Ji tuščia – neturi nė vieno simbolio.

Eilutę gaubiantys apostrofai atlieka skyrybos ženklų vaidmenį ir nelaikomi eilutės simboliais. Tai galima įsitikinti, kai eilutė rašoma į vaizduoklio ekraną. Pavyzdžiui, pagal sakinį

```
writeln('Čia yra eilutė')
```

bus parašytas tekstas:

```
Čia yra eilutė
```

Jeigu simbolių eilutėje reikalingas apostrofas (kaip simbolis), tai jis rašomas du kartus (kad skirtųsi nuo eilutės pradžią arba pabaigą žyminčių apostrofų).

Pavyzdys:

```
'Rašytojas 0''Henry'
```

Simbolių eilutę reikia skirti nuo teksto eilutės (žr. 2.5 skyr.). Teksto eilutė – tai ištisa teksto eilutė, kurią matome knygos puslapyje arba vaizduoklio ekrane. Ją sudaro taip pat simboliai. Tiktai teksto skirstymą į tokias eilutes apsprendžia pasirinktas puslapio plotis. Tuo tarpu čia aptartos simbolių eilutės yra teksto gabaliukai su kuriais patogu operuoti programoje.

Štai ir žinome pačius mažiausius programos elementus (leksemas). Iš jų, tarsi iš plytų, statomas pastatas – programa.

Uždaviniai

1.5.1. Kurie vardai sudaryti neteisingai ir kodėl jie neteisingi?

```
abc       cba       ABC       a(5)     ženklas
a5        5a        Lt.       a'       PrekėsKaina
sandauga  begin     a54s547   aš       prekės kaina
```

1.5.2. Kiek čia yra skirtingų vardų?

```
vid         vidurkis       VID
Vid         TrikPlotas     Trikplotas
```

1.5.3. Čia pateiktus duomenis sugrupuokite į keturis duomenų tipus: sveikuosius skaičius, realiuosius skaičius, simbolius ir simbolių eilutes:

```
1998      1.25E4      125E2      1.25
'1.25'    -56         0          0.0
'STALAS'  '+'        'a+b'       'nulis'
''''      ' '        '5'         ''
```

1.5.4. Šiuos skaičius užrašykite trumpiau, rodikliniu pavidalu:

```
0.0000000000012
3000000.0
-0.0000000001
```

1.5.5. Kiek čia yra skirtingų eilučių

```
'JONAS IR ONA'
'ONA IR JONAS'
'JONAS ''IR'' ONA'
JONAS  IR  ONA'
'Jonas ir Ona'
'12'
'+12'
'012'
'12.0'
```

1.5.6. Programos `vidurkis` (žr. 1.1.1 skyr.) žodžius sugrupuokite į: bazinius žodžius, standartinius vardus ir programuotojo sudarytus vardus.

Praktikos darbas

1.5.1. Eilėraštis. Sudarykite ir su kompiuteriu išbandykite programą, kuri parašytų šį Kristijono Donelaičio posmą:

```
Sveiks, svieteli margs!
    Šventes pavasario šventęs;
Sveiks ir tu, žmogau!
    Sulaukęs vasarą mielą;
```

Kiekvienos eilutės rašymą nurodykite atskiru sakiniu `writeln`.

Rašomas tekstas turi būti išdėstytas lygiai taip, kaip čia parodyta.

1.6. Programos struktūra

Ankstesniame skyrelyje nagrinėjome pačias mažiausias plyteles – *leksemas*, iš kurių sudaroma programa. Tai buvo žvilgsnis į programą iš apačios. Dabar pažvelgsime į programą iš viršaus – iš kokių stambiausių dalių sudaroma programa.

Paskalio programą sudaro keturios dalys:

- programos antraštė,
- aprašų dalis,
- veiksmų dalis,
- programos pabaigos simbolis – taškas.

Svarbiausios yra aprašų ir veiksmų dalys. Aprašų dalyje pateikiami mums jau pažįstami kintamųjų aprašai, o kaip vėliau matysime – ir kitokių objektų (konstantų, duomenų tipų, funkcijų bei procedūrų) aprašai.

Veiksmų dalį sudaro sakiniai, kuriais užrašomi veiksmai.

Ar kiekviena programa privalo turėti visas išvardytas dalis?

Nebūtinai. Kiekviena programa privalo turėti tik antraštę ir veiksmų dalį. Taigi pati trumpiausia programa galėtų būti tokia:

```
program tuščia;
begin
end.
```

Programa `tuščia` turi veiksmų dalį (žodis **`begin`** rodo veiksmų pradžią, o **`end`** – jų pabaigą), bet joje – nė vieno veiksmus nurodančio sakinio (galima sakyti ir kitaip – veiksmų dalis turi tik vieną tuščią sakinį). Taigi, programa neatlieka jokio veiksmo. Bet yra taisyklinga. Ją galima pateikti kompiuteriui. Kompiuteris pagal visas taisykles ją įvykdys, bet neduos jokio rezultato, nes programoje nėra rezultato gavimo ir rašymo (išdavimo) veiksmų.

Dabar pateiksime programą, turinčią tik veiksmų dalį.

```
program pasveikinimas;
begin
  writeln('Laba diena')
end.
```

Ši programa rašo tekstą

```
Laba diena
```

Programą galima papildyti *komentarais*. Komentarai rašomi į skliaustus { }. Programą `vidurkis` papildysime komentarais.

```
program vidurkis;
  var a, b,              { pradiniai duomenys }
      vid: real;         { rezultatas }
begin
  read(a, b);            { vienu sakiniu galima užrašyti kelių }
                         { pradinių duomenų skaitymą }
  vid := (a+b)/2;        { apskaičiuojamas vidurkis }
  writeln(vid: 8: 2)     { rašomas į ekraną rezultatas }
end.
```

Komentarai – tai paaiškinimai, skirti tik žmogui, bet ne kompiuteriui. Kompiuteris nekreipia dėmesio į tai, kas parašyta komentaruose ir programą atlieka taip, lyg komentarų nebūtų. Dėl to į komentarus galima rašyti bet kokį tekstą.

Komentarais paaiškinamos sunkiau suprantamos programos vietos. Komentarai ypač tinka kintamiesiems paaiškinti. Žinoma, galima parinkti ilgus vardus tokius, kad jie būtų aiškūs ir be komentarų. Tačiau tada tą patį ilgą vardą vėliau tektų dar daugelį kartų rašyti (atlikti daugiau darbo). Be to pernelyg ilgi vardai ilgina programą ir pasidaro ją sunkiau (o gal nuobodžiau) skaityti. Komentarai padeda pasiekti kompromisą: galima naudoti trumpesnius, bet išsamiai paaiškintus (pakomentuotus) vardus.

Pastaba. Turbo Paskalio ir Virtualiojo Paskalio varduose vartojamos tik angliškos raidės. Todėl, kai programa pateikiama kompiuteriui, tik nedidelę dalį joje panaudotų objektų (pvz., kintamųjų) galima vadinti nesutrumpintais lietuviškais vardais. Tais atvejais, kai vardo parašyti negalima, reikia vartoti vardo santrumpą ar kitokį jį primenantį raidinį žymenį ir jį paaiškinti komentaru, pavyzdžiui,

```
var sav,              { savaitė }
    mn: integer;      { mėnuo }
    z: string;        { žodis }
```

Beveik kiekvienai programai būdinga veiksmų triada:

- duomenų skaitymas,
- duomenų apdorojimas,
- rezultatų rašymas.

Šiame paprastame pavyzdyje kiekviena triados dalis išreiškiama vienu sakiniu. Didesnėse programose kiekvieną triados dalį gali sudaryti daugelis sakinių.

Uždaviniai

1.6.1. Programą `pasveikinimas` modifikuokite taip, kad ji ekrane parodytų tekstą

```
Sveiki, visi
```

1.6.2. Programos `pinigai` (žr. 1.4 skyr.) kintamųjų aprašus papildykite komentarais.

Praktikos darbas

1.6.1. Ritinys. Turime programą ritinio pagrindo plotui ir tūriui skaičiuoti

```
program ritinys;
  const pi = 3.1415626536;
  var h,                    { ritinio aukštis }
      d,                    { pagrindo skersmuo }
      p,                    { pagrindo plotas }
      v: real;              { tūris           }
begin
  read(h, d);
  p := d*d/4*pi;
  v := p*h;
  writeln('Plotas: ', p: 8: 2);
  writeln('Tūris: ',  v: 8: 2)
end.
```

Šią programą atlikite kompiuteriu ir išbandykite, kokius rezultatus gausite pateikę šiuos pradinius duomenis:

- ritinio aukštis 15 cm ir pagrindo skersmuo 2,0 cm;

• ritinio aukštis 2 m ir pagrindo skersmuo 50 cm.

Pakeiskite programą taip, kad pirmiau reikėtų surinkti pagrindo skersmenį, po to – aukštį.

Papildykite programą taip, kad ji skaičiuotų ir ritinio paviršiaus plotą.

2. DUOMENYS IR VEIKSMAI SU JAIS

Programoje užrašomi veiksmai su duomenimis. Todėl pirmiausia aprašomi duomenys, po to veiksmai su tais duomenimis. Duomenų kaita vyksta kompiuterio atmintinėje. Su šiais dalykais susidūrėme ankstesniame skyriuje. Ten nagrinėjome jau parašytų programų pavyzdžius. Dabar visa tai pateiksime išsamiau, nes tai yra pačios pagrindinės sąvokos su kuriomis programuotojas susiduria kiekviename žingsnyje. Pateiktų žinių pakaks, kad galėtume ir patys parašyti bet kokių skaičiavimų pagal formules programas.

2.1. Konstantos ir kintamieji

Programose vartojami dviejų rūšių dydžiai: pastovūs ir kintami. Kaip ir matematikoje, pastovūs dydžiai vadinami *konstantomis*, o kintami – *kintamaisiais dydžiais* arba trumpiau – *kintamaisiais*.

Konstantos – tai į programą įrašyti skaičiai, loginės reikšmės, tekstai (simboliai, simbolių eilutės) ir kitos reikšmės. Konstantų pavyzdžiai:

Duomenų tipas	Tipo vardas	Konstantų pavyzdžiai
sveikasis skaičius	integer	`26`
realusis skaičius	real	`721.25` `3.444E3`
loginis	boolean	`false`
simbolinis	char	`'A'`

eilutė	**string**	`'ABC'`
		`'1999 m. balandžio 1 d.'`
		`'A'`

Kiekvienas duomuo apibūdinamas *duomenų tipu* ir *reikšme*. Koks yra konstantos tipas, vienareikšmiškai galima pasakyti iš jos užrašo pavidalo, išskyrus vieną atvejį: vieno simbolio ilgio eilutė nesiskiria nuo simbolio.

Eilutės tipo standartinis Paskalis neturi, tačiau jis yra beveik visuose Paskalio dialektuose. Kol kas eilutes naudosime tik rašymo sakiniuose. Daugiau apie jas kalbėsime 9 skyriuje.

Kintamieji gali įgyti įvairias reikšmes. Jų reikšmės keičiamos programos vykdymo metu. Iš vienų kintamųjų reikšmių apskaičiuojamos naujos reikšmės ir jos priskiriamos tiems patiems arba kitiems kintamiesiems. Visus skaičiavimus galima įsivaizduoti kaip kintamųjų reikšmių keitimus. Todėl kartais sakoma: **kas suprato kintamuosius, tas suprato ir programavimo esmę.**

Programos tekste kintamieji žymimi vardais. Kompiliatorius kiekvienam kintamajam paskiria vietą kompiuterio atmintinėje. Tose atmintinės vietose saugomos kintamųjų reikšmės. Tai duomenys (skaičiai, loginės reikšmės, tekstai). Yra patogiau, kai tas pats kintamasis gali įgyti tik vieno to paties tipo reikšmes. Tada aišku, kiek vietos toms reikšmėms reikės kompiuterio atmintinėje ir tą vietą iš anksto galima rezervuoti. Be to, aišku kokias operacijas bus galima atlikti su tomis reikšmėmis.

Kaip minėjome kintamieji žymimi vardais. Visi vardai sudaromai pagal tas pačias taisykles (žr. 1.5 skyr.), todėl iš vardo užrašo pavidalo negalima nustatyti, kokio tipo kintamąjį žymi tas ar kitas vardas. Dėl to kintamieji aprašomi. Apraše nurodomas kintamojo tipas. Aprašų pavyzdžiai:

```
var plotas: real;
    a, kiekis: integer;      { viename apraše gali būti keli kintamųjų }
                                    { vardai skiriami kableliais }
    raidė: char;
    žodis: string[20];              { didžiausias eilutės ilgis }
    sakinys, frazė: string; { kai ilgis nenurodytas, laikoma, kad jis yra }
                                    { didžiausias leistinas – 255 simboliai }
```

To paties aprašo kintamųjų vardai skiriami kableliais. Visas aprašas baigiamas kabliataškiu. Tam, kad aprašas būtų vaizdesnis, kiekviena grupė pradedama nauja eilute, o visų grupių aprašų pradžios lygiuojamos.

Kitas išdėstymo būdas, kai žodis **var** rašomas atskiroje eilutėje, pavyzdžiui,

```
var
  a, kiekis: integer;
  plotas: real;
```

Aprašai suteikia kintamajam duomenų tipą, kurio vėliau pakeisti nebegalima – jis išlieka tas pats visą kintamojo gyvavimo laiką. Taigi kintamojo duomens tipas yra pastovus, o reikšmė – kintama.

Aprašas nesuteikia kintamajam jokios reikšmės. Sakoma, kad tik ką aprašyto kintamojo reikšmė yra neapibrėžta. Kintamiesiems reikšmės priskiriamos vėliau, kai atliekami programoje užrašyti veiksmai. Jos gali būti daug kartų pakeistos.

Konstantų aprašai. Konstantas galima pažymėti ir vardais. Vardais pažymėtas konstantas reikia aprašyti. Konstantų aprašai pradedami žodžiu **const**, po to išvardijamos konstantos ir jų reikšmės, pavyzdžiui,

```
const pi = 3.1415626536;
      n = 2;
      alg = 'Algoritmai ir programos';
```

Šitaip aprašytas vardas `pi` tampa skaičiaus `3.1415626536` sinonimu, o vardas `n` – skaičiaus `2` sinonimu.

Kam reikalingos vardais pažymėtos konstantos, juk kur reikia programoje galima parašyti ir tikrus skaičius ar kitokias reikšmes?

Vardais patogu pažymėti konstantas, kurios programoje panaudojamos keletą kartų. Tada patogiau rašyti trumpą ir informatyvų vardą, pavyzdžiui, `pi` – vietoj ilgo skaičiaus (kurį rašant nesunku suklysti), `alg` – vietoj ilgos frazės. Šitaip išvengiama klaidų, kurios gali atsirasti kai programoje tą patį skaičių arba tą pačią eilutę reikia rašyti daugelį kartų. Prireikus programą su vardais pažymėtomis konstantomis lengviau pataisyti, kai reikia pakeisti konstantos reikšmę, nes ją pakanka pakeisti vieną kartą – konstantos apraše.

Pavyzdys. Programa ritinio pagrindo plotui, paviršiaus plotui ir tūriui skaičiuoti

```
program ritinys;
  const pi = 3.1415626536;
        n = 2;                  { kiek skaitmenų po kablelio }
  var h,                        { ritinio aukštis }
      d,                        { pagrindo skersmuo }
      p,                        { pagrindo plotas }
      pavp,                     { viso paviršiaus plotas }
```

```
      v: real;                  { tūris        }
begin
  read(h, d);
  p := d*d/4*pi;
  pavp := 2*p + pi*d*h;
  v := p*h;
  writeln('Plotas: ', p: 8: n);
  writeln('Paviršiaus plotas: ', pavp: 8: n);
  writeln('Tūris: ', v: 8: n)
end.
```

Jeigu nuspręstume padidinti visų rašomų rezultatų tikslumą, pavyzdžiui, rašyti tris, keturis ar daugiau skaitmenų po kablelio, pakaktų programą pataisyti vienoje vietoje – pakeisti skaičių konstantos n apraše.

Konstantų aprašuose duomenų tipas nenurodomas, nes konstantos tipą vienareikšmiškai galima nustatyti iš jai suteikiamos reikšmės užrašo pavidalo.

Konstantos reikšmės programoje keisti negalima. Ji išlieka ta pati visą konstantos gyvavimo laiką.

Uždavinys

2.1.1. Programoje `ritinys` yra aprašyta konstanta `n = 2`. Kodėl prieskyros sakinyje

```
pavp := 2*p + pi*d*h
```

rašomas skaičius 2, o ne konstantos `n` vardas?

2. 2. Aritmetiniai reiškiniai ir jų reikšmių priskyrimas kintamiesiems

Sveikieji ir realieji skaičiai. Matematikoje sveikieji skaičiai laikomi realiųjų skaičių aibės poaibiu. Programavime realieji ir sveikieji skaičiai priklauso skirtingoms aibėms, neturinčioms bendrų elementų. Taigi, programoje skaičius `5` skiriasi nuo skaičiaus `5.0`. Skirtumai atsiranda dėl to, kad sveikieji ir realieji skaičiai skirtingai koduojami, o svarbiausia – skirtingai atliekamos ir operacijos su jais.

Aritmetinės operacijos su sveikaisiais skaičiais atliekamos tiksliai. Tačiau skaičiaus dydis ribojamas ir tą ribą galima greitai pasiekti. Pats didžiausias sveikasis skaičius žymimas vardu `maxint`. Tai integruota konstanta. Jeigu rezultatas viršija šį skaičių, jis prarandamas. Tokia situacija vadinama *perpildymu*.

Koks yra `maxint`, Paskalio standartas neapibrėžia. Jis priklauso nuo kompiuterio genties ir Paskalio dialekto. Turbo Paskalyje `maxint = 32767`, Paskalyje-E `maxint` = – 2147483647.

Leistini realiųjų skaičių rėžiai kompiuteryje kur kas platesni ir retai kada viršijami. Todėl retai kada tenka rūpintis, kad neįvyktų perpildymas. Tačiau aritmetinės operacijos su realiaisiais skaičiais atliekamos apytiksliai. Atsiranda paklaidos. Dėl to negalima garantuoti, kad visada bus tenkinama, pavyzdžiui, šitokia lygybė

```
a/b*b = a
```

Dažniausiai paklaidos būna labai mažos ir jų galima nepaisyti. Todėl su realiaisiais skaičiais mėgstama dirbti: ir perpildymo pavojus negresia, ir paklaidos nejaučiamos. Tačiau retkarčiais pasitaiko atvejų, kai paklaidos gaunamos neįtikėtinai didelės ir iškreipia rezultatą.

Sveikųjų skaičių operacijos. Su sveikaisiais skaičiais atliekamos aritmetinės operacijos

`+` sudėtis,

`-` atimtis,

* daugyba,

div dalyba (rezultatas – dalmuo),

mod dalyba (rezultatas – liekana).

Kiekvienos operacijos rezultatas taip pat sveikasis skaičius.

Painiausios yra dalybos operacijos. Jos ir žymimos neįprastai – baziniais žodžiais, o ne operacijų ženklais. Kai abu operandai teigiami skaičiai, tai dalmens ir liekanos apskaičiavimas problemų nekelia. Pavyzdžiui, aišku, kad

```
7 div 3 = 2;
7 mod 3 = 1;
3 div 7 = 0;
3 mod 7 = 3.
```

Tačiau kai bent vienas operandas neigiamas, dalmuo ir liekana nebetenka gyvenimiškos interpretacijos. Tokiu atveju susitarta, kad dalmens ženklas nustatomas taip, kaip priimta matematikoje: jei abiejų operandų ženklai vienodi – dalmuo teigiamas, jei skirtingi – neigiamas. Na, o dalmens *d* ir liekanos *l* dydžiai, kai dalijamasis yra *a*, o daliklis *b* randami tokie, kad būtų tenkinamos sąlygos:

$a = b \times d + l$

$l < |b|$

Taigi,

```
-7 div 3 = -2
-7 mod 3 = -1
-7 div -3 = 2
```

Kai daliklis neigiamas, Paskalis liekanos neapibrėžia – perdaug negyvenimiška situacija.

Su realiaisiais skaičiais atliekamos operacijos:

\+ sudėtis,

\- atimtis,

* daugyba,

/ dalyba.

Kiekvienos operacijos rezultatas yra realusis skaičius.

Dalijama apytiksliai tol, kol trupmeninė rezultato (dalmens) dalis dar telpa į realiajam skaičiui skirtą vietą atmintyje. Todėl liekanos sąvoka realiųjų skaičių atveju netenka prasmės.

Akivaizdu, kad sveikųjų ir realiųjų skaičių dalybos operacijos yra skirtingos – jos ir žymimos skirtingai. Tačiau iš tikrųjų skirtingos ir kitos operacijos: sudėtis, atimtis ir daugyba, nors jos žymimos tais pačiais ženklais. Jeigu kam teko susidurti su kompiuterio komandų kodais, tai tas žino, kad sveikųjų skaičių operacijų kodai yra kitokie, negu joms analogiškų realiųjų skaičių operacijų kodai.

Kaip kompiuteris nustato, kurios operacijos variantą atlikti, kai programoje abiem atvejais operacija žymima tuo pačiu ženklu?

Iš operandų tipo. Jeigu operacijų +, −, * abu operandai yra sveikieji skaičiai, tai atliekama sveikųjų skaičių operacija ir rezultatas gaunamas sveikasis skaičius (koks bus rezultato tipas, reikia visada žinoti rašant programą). Jeigu abu operandai realieji skaičiai, tai atliekama realiųjų skaičių operacija ir gaunamas realiojo tipo rezultatas.

O ką daryti, kai operacijų +, − ir * operandai yra skirtingo tipo (vienas – sveikasis, kitas – realusis)? Tokiu atveju sveikasis operandas paverčiamas realiuoju skaičiumi, atliekama realiųjų skaičių operacija ir gaunamas realusis rezultatas.

Dalybos / operacija atliekama tik su realiaisiais skaičiais. Jei vienas arba net abu operandai yra sveikieji skaičiai, tai prieš atliekant operaciją jie paverčiami realiaisiais. Todėl operacijos / rezultatas visada realusis skaičius.

Pateiksime pavyzdžių, iš kurių matyti, kaip priklauso rezultato tipas nuo operandų tipų ir operacijų.

```
5 + 2 = 7;
5.0 + 2 = 7.0;
5 + 2.0 = 7.0;
5 - 2 = 3;
5.0 - 2 = 3.0;
5 - 2.0 = 3.0;
5 * 2 = 10;
5.0 * 2 = 10.0;
5 * 2.0 = 10.0;
5.0 * 2.0 = 10.0;
5 div 2 = 2;
5 mod 2 = 1;
10 / 2 = 5.0;
10.0 / 2 = 5.0;
10.0 / 2.0 = 5.0;
10.0 / 2.0 = 5.
```

Operacijų **div** ir **mod** operandai gali būti tik sveikieji skaičiai. Jeigu bent vienas jų realusis – gaunama klaida. Šią klaidą kompiuteris aptinka transliuodamas programą (t.y. dar nepradėjęs jos vykdyti), nes kiekvieno kintamojo ar reiškinio duomenų tipą galima nustatyti iš programos teksto nepriklausomai nuo pradinių duomenų .

Dar viena bet kokiai dalybos operacijai (**div**, **mod** ir /) būdinga klaida – dalyba iš nulio. Žinome, kad iš nulio dalyti negalima. Kai pamirštame ir tokį nelestiną veiksmą bandome pateikti kompiuteriui, jis apie tai iš karto primena, informuodamas apie klaidą. Šią klaidą kompiuteris aptinka vykdydamas programą, nes daliklio reikšmė gali priklausyti nuo pradinių duomenų, o kai programa transliuojama, jie dar nežinomi.

Iš operandų – konstantų ir kintamųjų – sudaromi aritmetiniai reiškiniai. Operacijų atlikimo tvarka nurodoma skliaustais. Kai nėra skliaustų, operacijos atliekamos pagal prioritetus:

```
-       (unarinė atimtis)
* / div mod
+ -
```

Aukščiausią prioritetą turi (t.y. atliekama pirmiausiai) unarinė atimtis (skaičiaus ženklo keitimo operacija), žemesnį – daugybos ir dalybos operacijos, o žemiausią (atliekama paskiausiai) – sudėties ir atimties (binarinės) operacijos.

Aritmetinių reiškinių pavyzdžiai:

```
5
a
a + 5
(a + 5) * 2 div 14
-a * ((c + d) - (alfa + beta))
```

Atkreipiame dėmesį, kad reiškinys gali būti sudarytas tik iš vienos konstantos arba tik vieno kintamojo.

Kiekvienas reiškinys turi reikšmę. Tai paskutinės atliktos operacijos rezultatas.

Reiškinio reikšmę galima priskirti kintamajam. Tai užrašome prieskyros sakiniu. Jo pavidalas yra šitoks:

kintamojo vardas := reiškinys.

Prieskyros sakinių pavyzdžiai:

```
k := 5;
k := a;
k := (a + 5) * 12 div 14
```

Atlikus prieskyros veiksmą, kintamasis, kurio vardas parašytas kairėje simbolio := pusėje, įgyja reikšmę reiškinio, esančio dešinėje simbolio := pusėje.

Kompiuteris atlieka programoje užrašytas operacijas su kintamųjų reikšmėmis bei konstantomis. Operacijų rezultatus vėl įrašo į atmintį, t.y. priskiria kintamiesiems. Taigi kintamųjų reikšmės nuolat keičiamos: iš vienų reikšmių apskaičiuojamos kitos, kol galų gale gaunamos reikšmės, kurias galima pavadinti galutiniais rezultatais.

Reiškinio užrašas yra vaizdesnis, kai abipus žemesnį prioritetą turinčių operacijų paliekama po tarpą. Dėl to, pavyzdžiui, reiškinys

```
a*b + c*d
```

yra vaizdesnis, negu

```
a*b+c*d,
```

o užrašas

```
a * b+c * d
```

yra netgi klaidinantis, nes jis skaitytoją orientuoja į kitokią operacijų atlikimo tvarką, negu iš tikrųjų ji yra.

Kadangi reikšmės priskyrimo veiksmas atliekamas paskiausiai, tai abipus prieskyros ženklo visada derėtų palikti po tarpą, ką ir darėme ankstesniuose pavyzdžiuose.

Sveikojo tipo reiškinio reikšmę galima priskirti ir sveikojo, ir realiojo tipo kintamajam. Pastaruoju atveju ji automatiškai pakeičiama realiąja.

Realiojo tipo reiškinio reikšmę galima priskirti tik realiojo tipo kintamajam. Mat jeigu tokį priskyrimą parašytume, kompiuteris nežinotų, ką daryti su realiojo skaičiaus trupmenine dalimi: ar ją atmesti, ar skaičių apvalinti. Realųjį skaičių galima pakeisti sveikuoju panaudojant integruotas funkcijas `trunc` ir `round`. Funkcijos `trunc` rezultatas yra lygus realiojo skaičiaus sveikajai daliai, o `round` – apvalintam realiajam skaičiui, pavyzdžiui:

```
trunc(5.6) = 5;
round(5.6) = 6.
```

Funkcija iš tikrųjų yra operacija, tik kitaip užrašyta. Pirmiausia eina funkcijos vardas, o po jo skliaustuose išvardijami operandai. Pavyzdžiui, jeigu vietoj operacijos **`div`** programavimo kalboje būtų funkcija, pavadinta tokiu pat vardu, tai vietoj

```
a div b
```

reikėtų rašyti

```
div(a, b).
```

Kitos į Paskalį integruotos aritmetinės funkcijos:

`abs(x)`	apskaičiuoja $\lvert a \rvert$,
`sqr(x)`	x^2,
`sin(x)`	sin x, čia x išreikštas radianais,
`cos(x)`	cos x, čia x išreikštas radianais,
`exp(x)`	e^x,
`ln(x)`	ln x, $x > 0$,
`sqrt(x)`	$\sqrt{x}$, $x \geq 0$,
`arctan(x)`	arctg x reikšmę radianais,
`succ(x)`	x = x+1,

`pred(x)` x = x–1.

Tai į standartinę Paskalio kalbą integruotos funkcijos. Paskalio dialektai jų turi daugiau.

Visų išvardytų funkcijų argumentas `x` gali būti sveikojo arba realaus tipo reiškinys. Funkcijų `abs` ir `sqr` rezultato tipas sutampa su argumento tipu. Visų kitų čia išvardytų funkcijų rezultatas yra realiojo tipo.

Programuotojas gali sudaryti naujas funkcijas. Apie tai kalbėsime 6 skyriuje.

Uždaviniai

2.2.1. Nustatykite, kurio tipo: sveikojo ar realiojo yra šių reiškinių reikšmės:

a) 12 + 6 / 4;

b) 12 + 6 / 3;

c) 12 + 6 **div** 4;

d) 12 + 6 **div** 3;

e) 12 + 6 **mod** 4;

f) 4 + 8.5 * 2

2.2.2. Apskaičiuokite šių reiškinių reikšmes:

a) 16 **div** 2 **div** 2 **div** 2;

b) 16 **mod** 3 **mod** 3.

2.2.3. Kurie reiškiniai klaidingi?

a) 10 **div** 2 / 2;

b) 10 / 2 **div** 2;

c) a **div** (b / 2);

d) a - - b;

e) a + + b;

f) a * * b;

g) a * - b;

h) a + 2 * b;

i) a + 2b;

j) a + b2.

2.2.4. Parašykite programą dviejų skaičių geometriniam vidurkiui rasti. Skaičių *a* ir *b* geometrinis vidurkis skaičiuojamas pagal formulę

$g = \sqrt{ab}$

2.2.5. Parašykite sakinius, kuriais realusis skaičius, reiškiantis pinigų sumą, būtų paverstas dviem sveikaisiais skaičiais: litais ir centais.

2.3. Duomenų skaitymas iš klaviatūros ir rašymas į ekraną

Bendru atveju pradiniai duomenys skaitomi iš bylų, kurios yra paruošiamos prieš atliekant programą, o rezultatai rašomi į bylas, kurios išlieka atlikus programą. Apie tai kalbėsime 2.5 skyrelyje.

Kitas, paprastesnis pradinių duomenų pateikimo būdas yra jų surinkimas klaviatūra. Klaviatūra surinktų duomenų skaitymas nurodomas šitokio pavidalo sakiniu

`read`(k_1, k_2, ..., k_n);

arba

`readln`*(k1, k2, ..., kn)*.

Kintamiesiems $k_1, k_2, ..., k_n$ priskiriamos klaviatūroje surinktos pradinių duomenų reikšmės. Jos priskiriamos kintamiesiems iš eilės: kintamajam k_1 – pirmoji reikšmė, kintamajam k_2 – antroji ir t.t. Kintamieji gali būti skirtingų tipų. Reikšmių tipai turi atitikti kintamųjų tipus pagal tokias pačias taisykles, kaip ir prieskyros sakinyje. Pavyzdžiui, sveikojo tipo kintamąjį gali atitikti tik sveikasis skaičius, o realiojo tipo kintamąjį – ir realusis, ir sveikasis skaičius. Mat abiem atvejais – skaitymo sakinyje ir prieskyros sakinyje – kintamajam priskiriama reikšmė. Tik vienu atveju apskaičiuota reiškinio reikšmė, o kitu atveju – klaviatūroje surinkta reikšmė.

Apie duomenų skaitymo sakinio formą `readln` (angl. *read line* – skaityti eilutę) kalbėsime 2.6 skyr.

Rezultatus paprasčiausia parodyti ekrane. Rezultatų rašymas į ekraną nurodomas šitokio pavidalo sakiniais:

`write`($r_1, r_2, ..., r_n$);

`writeln`($r_1, r_2, ..., r_n$);

Skliaustuose išvardijami reiškiniai, kurių reikšmes reikia rašyti į ekraną.

Priminsime, kad kintamasis ir konstanta čia laikomi reiškiniais (jie turi reikšmes). Iki šiol rašymo sakiniuose nurodėme tik kintamuosius arba konstantas (simbolių eilutes). Dabar, žinodami, kad čia gali būti bet kokie reiškiniai, galime programas sutrumpinti, iš jų pašalindami rezultatų kintamuosius.

Pavyzdys.

```
program ritinys;
  const pi = 3.1415626536;
        n = 2;                { kiek skaitmenų po kablelio }
  var h,                      { ritinio aukštis }
      d,                      { pagrindo skersmuo }
      p: real;                { pagrindo plotas }
begin
  read(h, d);
  p := d*d/4*pi;
  writeln('Plotas:', p: 8: n);
  writeln('Paviršiaus plotas:', 2*p + pi*d*h: 8: n);
  writeln('Tūris:', p*h: 8: n)
end.
```

Palikome tik vieną rezultato kintamąjį `p`, nes jo reikšmė vartojama ne tik rašymo sakinyje, bet ir kitų rezultatų skaičiavimo reiškiniuose.

Rašymo sakinyje gali būti nurodytas bet koks rezultatų (reiškinių) skaičius. Nėra skirtumo, ar tuos pačius reiškinius surašysime į vieną sakinį, ar išskirstysime į kelis. Todėl sakinį

`write`($r_1, r_2, ..., r_n$)

galima laikyti sakinių

`write`(r_1);
`write`(r_2);
...
`write`(r_n)

santrumpa.

Koks skirtumas tarp sakinių `write` ir `writeln`?

Sakiniu `write` nieko nepasakoma apie rezultatų suskirstymą į eilutes. Sakiniai `write` rašo rezultatus vieną po kito. Kai nebetelpa, pereinama į kitą eilutę. Perkeliama ten, kur pasiekiamas eilutės kraštas. Taigi skaičius ar tekstas gali būti skeliamas bet kurioje vietoje. Šitaip mechaniškai į eilutes suskaidytus rezultatus nepatogu skaityti. Todėl teksto skirstymą į eilutes geriau nurodyti programoje. Ten, kur reikia, kad būtų pereinama į naują eilutę, rašome sakinį

```
writeln
```

Šio sakinio veiksmas prilygtų eilutės pabaigos klavišo paspaudimui, jeigu kompiuteris rašytų rezultatus klaviatūra.

Sakinyje `writeln` galima nurodyti ir rezultatų. Tada kompiuteris tuos rezultatus parašys, o po to „paspaus“ eilutės pabaigos klavišą. Taigi sakinys

`writeln`(r_1, r_2, ..., r_n)

yra ekvivalentus sakinių porai:

`write`(r_1, r_2, ..., r_n);
`writeln`

Jeigu norima palikti *n* tuščių eilučių, reikia parašyti *n* sakinių `writeln`.

Rezultatų rašymą rekomenduojama užbaigti sakiniu `writeln`. Paskutinė neužbaigta (t.y. be eilutės pabaigos ženklo) eilutė gali dingti.

1 pavyzdys. Skaitomi du skaičiai ir išspausdinami į atskiras eilutes.

```
program du;
  var a: integer;
begin
  read(a);
  writeln(a);
  read(a);
  writeln(a)
end.
```

Programoje vartojamas tik vienas kintamasis. Kai jau atliktas antrojo sakinio veiksmas – parašyta kintamojo `a` reikšmė, trečiuoju sakiniu skaitoma ir jam priskiriama nauja reikšmė. Ankstesnė kintamojo `a` reikšmė dingsta, nes į jos vietą rašoma nauja. Ankstesnę reikšmę galima prarasti, nes ji jau išspausdinta ir daugiau programoje nevartojama. Todėl tą patį kintamąjį (o tuo pačiu – ir tą pačią vietą kompiuterio atmintinėje) galima panaudoti kito pradinio duomens reikšmei saugoti. Naujos reikšmės spausdinimą nurodo paskutinis sakinys.

2 pavyzdys. Skaitomi du skaičiai ir išspausdinami atvirkščia tvarka, negu buvo perskaityti: į pirmą eilutę antrasis skaičius, į antrą – pirmasis.

```
program DuAtbulai;
  var a, b: integer;
begin
  read(a, b);
  writeln(b, a: 5)
end.
```

Čia vieno kintamojo nebepakanka, nes pirmojo perskaityto kintamojo reikšmę reikia išsaugoti kol bus perskaityta ir parašyta antrojo kintamojo reikšmė.

Kiek vietos skirti į ekraną rašomai reikšmei, nurodoma *formatu* – sveikuoju skaičiumi (bendru atveju – sveikojo tipo reiškiniu), parašytu po tos reikšmės ir atskirtu dvitaškiu. Pavyzdžiui, sakinyje

```
write(1234: 6, -235: 6)
```

parodoma, kad kiekvienam skaičiui skiriama po 6 pozicijas (6 simbolius) ekrane. Skaičiai bus išdėstyti šitaip

```
␣␣1234␣-235
```

Čia ženklu ∪ pažymėti tarpai. Prieš pirmąjį skaičių bus palikti du tarpai, nes skaičius užima tik 4 pozicijas iš jam skirtų 6, o prieš skaičių `-1235` liks tik vienas tarpas, nes skaičiui su minusu reikia 5 pozicijų.

Jeigu rezultatas yra realiojo tipo, tai jam galima užrašyti dar vieną (antrą) formatą. Jis nurodo, kiek vietų po taško skiriama skaičiaus trupmeninei daliai. Pavyzdžiui, sakinio

```
write(3.1415926536: 10: 5,  3.1415926536: 10: 2)
```

rezultatas bus

```
3.14159      3.14
```

Jeigu nenurodytas nė vienas realiojo skaičiaus formatas arba nurodytas tik vienas, tai skaičius rašomas rodykliniu pavidalu. Pavyzdžiui, pagal sakinį

```
write(3.1415926536)
```

Turbo Paskalis parašytų

```
3.1415926536E00
```

Kai formatas mažesnis už rezultatui reikalingą pozicijų skaičių, tai vis tiek rašomi visi rezultato simboliai, tik nepaliekama tarpų.

Kaip rašoma, kai formatas nenurodytas, priklauso nuo Paskalio dialekto. Turbo Paskalyje rezultatui skiriama vietos tiksliai tiek, kiek jis turi simbolių. Todėl nenurodžius formatų, rezultatai (skaičiai) susilieja ir pasidaro neįskaitomi. Norint atskirti vieną rezultatą nuo kito reikia parinkti formatus, didesnius už rezultato užimamą pozicijų skaičių arba tarp rezultatų įterpti tarpo simbolius, pavyzdžiui,

```
write(a, ' ', b, ' ', c).
```

Daugumoje kitų Paskalio kalbos dialektų, kai formatas nenurodytas, skaičiui skiriama viena pozicija daugiau, negu reikėtų pačiam ilgiausiam skaičiui. Tokiu atveju nesusilieja gretimi skaičiai.

Pateiksime pavyzdžių.

3 pavyzdys. Skaitomi trys skaičiai ir rašomi į vieną eilutę atbula tvarka negu jie buvo perskaityti

```
program trys;
  var a, b, c: integer;
begin
  read(a, b, c);
  writeln(c, ' ', b, ' ', a)
end.
```

Čia jau kiekvienam skaičiui reikia atskiro kintamojo, nes pirmuosius du skaičius reikia saugoti kol bus perskaitytas ir parašytas trečiasis.

Jeigu, pavyzdžiui, kompiuteriui buvo pateikti pradiniai duomenys

```
31   12  1998
```

tai jis išspausdins

```
1998 12 31
```

4 pavyzdys. Skaitomi trys skaičiai. Rašomas pirmasis skaičius, antrasis skaičius, pakartotas du kartus, ir trečiasis skaičius, pakartotas tris kartus:

```
program trys;
  var a: integer;
begin
  read(a);
  writeln(a);
  read(a);
  writeln(a, ' ', a);
  read(a);
  writeln(a, ' ', a, ' ', a)
```

```
end.
```

Jeigu kompiuteriui pateiksime pradinius duomenis

```
111
222
33
```

tai, atlikę šių sakinių veiksmus, jis išspausdins skaičius

```
111
222 222
33 33 33
```

Trečias skaičius yra dviženklis. Todėl jis nebesilygiuoja su ankstesnėse eilutėse išspausdintais triženkliais skaičiais. Sulygiuoti galima panaudojus formatus.

5 pavyzdys.

```
program trys;
  const f = 4;
  var a: integer;
begin
  read(a);
  writeln(a: f);
  read(a);
  writeln(a: f, a: f);
  read(a);
  writeln(a: f, a: f, a: f)
end.
```

Dabar kompiuteris išspausdintų:

```
111
222 222
 33  33  33
```

Skaičiai lygiuojami taip, kad vienas po kito būtų vienodo reikšmingumo skaitmenys. T.y. sveikieji skaičiai pagal dešiniąją pusę, o realieji pagal trupmeninę dalį skiriantį kablelį.

Uždaviniai

2.3.1. Ką matysime ekrane po to, kai kompiuteris atliks programą:

```
program p;
  var pirmas, antras: integer;
begin
  read(pirmas, antras);
  writeln(pirmas, ' ', antras, ' ', pirmas)
end.
```

jeigu buvo pateikti šitokie pradiniai duomenys:

```
22   33
```

2.3.2. Ką matysime ekrane po to, kai kompiuteris atliks programą:

```
program p;
  var pirmas, antras: integer;
begin
  read(pirmas, antras, pirmas);
  writeln(pirmas, ' ', antras, ' ', pirmas)
end.
```

jeigu buvo pateikti šitokie pradiniai duomenys:

```
222 333 444
```

2.3.3. Ką parašys kompiuteris atlikęs programą

```
program ppp;
  var a: integer;

begin
  writeln(a);
  read(a)
end.
```

jeigu buvo pateiktas šitoks pradinis duomuo

```
22
```

Atsakymą paaiškinkite.

2.4. Skaičiavimai pagal formules

Šiame skyriuje išnagrinėtų Paskalio kalbos konstrukcijų pakanka, kad būtų galima užrašyti bet kokių skaičiavimų pagal formules programas. Tokių uždavinių programavimas labai paprastas: reikia formulę su aritmetiniais skaičiavimais užrašyti Paskalio kalbos žymenimis.

1 pavyzdys. Tiesinės lygties *ax+b=0* sprendimas.

```
program TiesLygtis;
var a, b,          { lygties koeficientai }
    x: real;       { lygties šaknis }
begin
  read(a, b);
  x := -b/a;
  writeln(x)
end.
```

2 **pavyzdys**. Programa daugianario

$a_3x^3 + a_2x^2 + a_1x + a_0$

reikšmei rasti.

```
program daugianaris;
var x,                  { daugianario kintamasis }
    a0, a1, a2, a3      { daugianario koeficientai }
    p: real;            { daugianario reikšmė }
begin
  read(x, a0, a1, a2, a3);
  p:= a0 + x*(a1 + x*(a2 + x*a3));
  writeln(p)
end.
```

Paskalio kalba kėlimo operacijos neturi. Todėl ją reikia išreikšti daugyba. Paraidžiui užrašę formulę Paskalio kalba, gautume:

```
p := a3*x*x*x + a2*x*x + a1*x +a0
```

Tačiau šią formulę galima užrašyti ekonomiškiau:

```
p := a0 + x*(a1 + x*(a2 + x*a3))
```

3 pavyzdys. Programa, išreiškianti duotą parų skaičių valandomis ir minutėmis.

```
program laikas;
  var p,                    { parų skaičius }
      h,                    { valandų skaičius }
```

```
      min: integer;          { minučių skaičius }
begin
  read(p);
  h := p*24;     write(h: 4);
  min := h*60;   writeln(min: 6)
end.
```

Jeigu pradinis duomuo būtų skaičius 2, tai kompiuteris, atlikęs pateiktą programą, išspausdintų šitokius rezultatus:

```
48  2880
```

4 pavyzdys. Pradiniai duomenys – pirkėjo turima pinigų suma, išreikšta litais ir centais bei prekės kaina, išreikšta litais ir centais (iš viso 4 skaičiai). Rezultatas – litais ir centais išreikšta pinigų suma, kuri liks pirkėjui, kai jis nusipirks tą prekę. Sudarysime programą šiam uždaviniui spręsti.

Pirmiausia paversime abi sumas centais, o kai rasime skirtumą, jį vėl paversime litais ir centais.

```
program pirkinys;
  var LtTuri, ctTuri,                 { pirkėjas turi }
      LtPrek, ctPrek,                 { prekė kainuoja }
      LtLiko, ctLiko: integer;        { pirkėjui liko }
begin
  read(LtTuri, ctTuri, LtPrek, ctPrek);
  ctLiko := (LtTuri*100 + ctTuri)
          - (LtPrek*100 + ctPrek);
  LtLiko := ctLiko div 100;
  ctLiko := ctLiko mod 100;
  writeln(LtLiko, ctLiko: 3)
end.
```

5 pavyzdys. Programa, duotą centų kiekį, išreiškianti mažiausiu monetų skaičiumi.

```
program centai;
  var suma,                                          { pinigų suma }
      ct50, ct20, ct10, ct5, ct2, ct1: integer;      { monetos }
begin
  read(suma);
  ct50 := suma div 50; suma := suma mod 50;
  ct20 := suma div 20; suma := suma mod 20;
  ct10 := suma div 10; suma := suma mod 10;
  ct5  := suma div  5; suma := suma mod  5;
  ct2  := suma div  2;
  ct1  := suma mod  2;
  writeln(' Monetų po');
  writeln('50 ct: ', ct50);
  writeln('20 ct: ', ct20);
  writeln('10 ct: ', ct10);
  writeln(' 5 ct: ', ct5);
  writeln(' 2 ct: ', ct2);
  writeln(' 1 ct: ', ct1)
end.
```

6 pavyzdys. Trikampio plotas. Parašysime programą trikampio plotui skaičiuoti, kai žinomi visų jo kraštinių ilgiai

```
program TrikPlotas;
var a, b, c,                  { kraštinių ilgiai }
    p,                        { pusperimetris }
```

```
    s: real;                { plotas }
begin
  read(a, b, c);
  p := (a+b+c)/2;
  s := sqrt(p*(p-a)*(p-b)*(p-c));
  writeln(s)
end.
```

7 pavyzdys. Dviženklio skaičiaus skaitmenų suma. Žmogus mato skaičių užrašytą skaitmenimis. Todėl šis uždavinys jam labai lengvas. Kompiuteryje skaičiai koduojami kitaip (dvejetaine skaičiavimo sistema). Todėl kompiuteriui dešimtainius skaitmenis reikia apskaičiuoti pasinaudojant kalboje esančiomis sveikųjų skaičių operacijomis. Šiam tikslui tinka sveikųjų skaičių dalybos operacijos.

```
program SkSuma;
  var s,                    { dviženklis skaičius }
      sum: integer;         {skaitmenų suma }
begin
  read(s);
  sum := s mod 10 +         { vienetai }
         s div 10;          { dešimtys }
  writeln(sum)
end.
```

Uždaviniai

2.4.1. Parašykite programą, kuri suskaičiuotų, kokiu mažiausiu monetų skaičiumi galima išreišti duotą pinigų sumą (centais). Jos rezultatas turi būti tik vienas skaičius – monetų skaičius. Pasinaudokite programa `centai` (žr. 5 pavyzdį).

2.4.2. Turime programą

```
program Keitimas;
  var a, b, c: integer;
begin
  read(a, b);
  ...
  writeln(a, ' ', b)
end.
```

Vietoj daugtaškių reikia įrašyti tokius prieskyros sakinius, kad du skaičiai būtų rašomi atvirkščia tvarka, negu jie buvo perskaityti, t.y. sukeičiami vietomis. Pavyzdžiui, jeigu buvo perskaityti skaičiai

```
123 456
```

tai turi būti rašoma

```
456 123
```

2.4.3. Pradinis duomuo – triženklis natūralusis skaičius. Parašykite programą, kurios rezultatas būtų:

a) pradinio duomens skaitmenų suma;

b) skaičius, gautas perrašius pradinio duomens skaitmenis atvirkščiai.

Praktikos darbai

2.4.1. Laiko skaičiavimai. Parašykite programą skirtumui tarp dviejų tos pačios paros laiko momentų rasti. Pradiniai duomenys – du laikai, išreikšti valandomis ir minutėmis (keturi skaičiai).

Pirmasis laikas ne vėlesnis už antrąjį. Rezultatas išreiškiamas dviem skaičiais: valandomis ir minutėmis. Programą išbandykite su tokiais pradinių duomenų rinkiniais:

```
 0  0 23 59
14 35 15 10
14 35 16 10
 2 15  2 15
```

2.4.2. Trikampis. Parašykite programą, kuri apskaičiuotų trikampio plotą. Pradiniai duomenys – trijų taškų – trikampio viršūnių koordinatės plokštumoje (šeši skaičiai). Programos išbandymui parinkite tokius pradinius duomenis, kurie aprašo:

a) statų trikampį;

b) buką trikampį;

c) tiesę (t.y. visi trys taškai yra vienoje tiesėje);

d) tašką (t.y. visų trijų taškų koordinatės sutampa).

2.5. Duomenų saugojimas tekstinėse bylose

Klaviatūra patogu surinkti nedaug pradinių duomenų. Be to kiekvieną kartą vykdant programą tenka visus pradinius duomenis rinkti iš naujo, nors ir mažai ką norėtume pakeisti, nes seni pradiniai duomenys neišsaugomi.

Į kompiuterio ekraną telpa nedaug rezultatų. Tai, kas rodoma ekrane, neišsaugoma.

Todėl tokie paprasčiausi duomenų pateikimo ir rezultatų gavimo būdai tinka tik mažoms programoms, turinčioms nedaug pradinių duomenų ir nedaug rezultatų. Kai operuojama su didesniais duomenų kiekiais, būtų patogiau pradinius duomenis imti iš disko, o rezultatus rašyti į diską. Kaip tai padaryti?

Į Paskalio kalbą yra integruotas tekstinės bylos duomenų tipas. Jis žymimas vardu `text`. Aprašysime du tekstinės bylos tipo kintamuosius:

```
var duomenys, rezultatai;
```

Tekstinė byla yra sudaryta iš simbolių. Šis tekstas, kurį skaitome, taip pat gali būti vaizduojamas teksto byla. Jame gali būti visko: žodžių, skaičių, skyrybos ir kitokių ženklų. Tačiau viskas išreikšta simboliais. Taigi, teksto byla yra ilga simbolių seka. Kaip ir įprastame tekste, ji suskirstyta į eilutes.

Taigi bylos kintamojo reikšmė yra struktūrinė – sudaryta iš komponentų – simbolių.Tai struktūrinis duomenų tipas.

Bylos (t.y. bylos tipo kintamųjų reikšmės) saugomos diskuose. Tuo tarpu kompiuteris atlieka operacijas tik su tais duomenimis, kurie saugomi jo operatyvioje atmintinėje. Todėl norint atlikti operacijas su byloje saugomais duomenimis, pirmiau juos reikia priskirti įprastiems kintamiesiems. O tai ne kas kitas, kaip pradinių duomenų skaitymas.

Paskalio programa „žino“ tik tuos vardus, kurie yra aprašyti programoje. Vardas ir juo pažymėtas objektas „gimsta“ kai jis aprašomas ir „miršta“ kai programa baigia darbą. Bylų, saugomų diskuose, vardus tvarko operacinė sistema. Tokios bylos ir jų vardai egzistuoja ilgai, nepriklausomai nuo Paskalio programų. Operacinė sistema nežino vardų, kurie yra aprašyti Paskalio programoje. Taigi, norint, kad Paskalio programa operuotų su bylomis, saugomomis diskuose, reikia susieti bylas, aprašytas Paskalio programoje su operacinės sistemos bylomis. Turbo Paskalyje ir daugelyje kitų transliatorių tokia sąsaja programoje nurodoma sakiniu `assign`, turinčiu tokį pavidalą:

`assign(`*bv*, *eil*`)`

čia *bv* – Paskalio bylos vardas,

eil – simbolių eilutė, kurios reikšmė – diske saugomos bylos vardas.

Pavyzdžiui, sakinys

```
assign(b, 'C:\DUOMENYS.TXT')
```

aukščiau aprašytą tekstinę bylą `b` susieja su diske saugoma byla `DUOMENYS.TXT`. Vadinasi, nuo šiol visi veiksmai, užrašyti su byla `b`, faktiškai bus atliekami su disko `C:` pagrindiniame kataloge saugoma byla `DUOMENYS.TXT`.

Kaip minėjome, tekstinėse bylose simboliai suskirstyti į eilutes. Kiekvienos eilutės pabaigą žymi specialus simbolis, vadinamas *eilutės pabaigos simboliu*. Kai byla peržiūrima kokiu nors žiūrikliu, toje vietoje, kur yra eilutės pabaigos simbolis, tekstas keliamas į naują eilutę. Ekrane nematome eilutės pabaigos simbolių, bet matome tekstą, gražiai suskirstytą į eilutes.

Kokios operacijos atliekamos su bylomis?

Tai mums jau pažįstamos duomenų skaitymo ir rašymo operacijos. Panagrinėsime jų specifiką.

Skaitymo operacija `read` skaito tekstą, užrašytą byloje, lygiai taip, kaip surinktą klaviatūra. Tiktai skliaustuose po operacijos vardo pirmiausia rašomas vardas bylos, iš kurios bus skaitoma, pavyzdžiui,

```
read(b, n)
```

Duomens reikšmė, perskaityta iš bylos `b`, bus priskirta kintamajam `n`.

Jeigu sakinyje `read` pirmasis vardas yra ne bylos vardas, tai tada skaitoma iš klaviatūros (ką jau žinome).

Iš klaviatūros perskaitoma viskas, kas buvo surinkta. Iš bylos skaitoma tik tiek, kiek simbolių sudaro reikšmę kintamojo, užrašyto sakinyje `read`, t.y. skaitoma iki pirmojo simbolio, nebepriklausančio to kintamojo reikšmei. Pavyzdžiui, jeigu bylos `b` pradžia būtų tokia:

```
25425   5842   -458
```

tai pagal ankstesnį sakinį būtų perskaitytas tik pirmasis skaičius `25425`, o tolesnis tekstas lauktų savo eilės – kitų skaitymo sakinių. Galima įsivaizduoti, kad byla slenka žymeklis, kurio kairėje yra perskaitytas tekstas.

Tarpai tarp skaičių nesvarbūs – jie praleidžiami. Taip pat nekreipiama dėmesio ir į teksto suskirstymą į eilutes: jeigu skaičiaus nerandama vienoje eilutėje (ar likusioje dar neperskaitytoje jos dalyje), einama į kitą eilutę ir t.t. praleidžiami visi tarpai kol surandamas skaičius.

Yra ir kita skaitymo sakinio forma `readln`. Šiuo atveju turi įtakos ir teksto skirstymas į eilutes: perskaitoma visa eilutė iki pabaigos. Į skliaustuose išvardytus kintamuosius „netilpusi“ eilutės dalis praleidžiama. Jeigu, pavyzdžiui, pirmosios dvi bylos `b` eilutės būtų tokios:

```
25425   5842   -458
2222 3333 4444
```

tai atlikus sakinius

```
readln(b, n); read(b, m)
```

kintamieji įgytų tokias reikšmes:

```
n = 25425, m = 2222
```

Tuo tarpu sakiniai

```
read(b, n); read(b, m)
```

jiems būtų priskyrę tokias reikšmes:

```
n = 25425, m = 5842
```

Sakinį `readln` patogu vartoti tada, kai programoje reikia panaudoti tik dalį duomenų, surašytų tekstų bylos eilučių pradžiose.

Duomens skaitymo iš klaviatūros pakartoti nebegalima, nebent iš naujo jį surinktume. Tuo tarpu iš bylos perskaitytas duomuo nedingsta. Norint pakartotinai skaityti bylą reikia grįžti į jos pradžią – atlikti sakinį

```
reset(b)
```

Byla `b` vėl bus skaitoma iš pradžių – skaitymo laukiantį duomenį nurodantis žymeklis bus nustatytas į bylos pradžią. Bylos žymeklio vietą galima traktuoti kaip bylos reikšmės dalį. O tik ką aprašyto kintamojo reikšmė būna neapibrėžta. Taigi, ir žymeklis gali būti bet kur. Todėl prieš bylos skaitymo veiksmus turi eiti sakinys `reset`.

Rašymo sakinys `write` rašo rezultatus (duomenis) į bylą, kurios vardas yra parašytas pirmuoju skliaustuose. Jeigu pirmasis vardas nėra bylos vardas, tai rašoma į ekraną (ką jau žinome).

Sakinys `writeln(b)` formuoja naujos eilutės rašymą į bylą `b`.

Rašoma į bylos pabaigą. Prirašomi nauji simboliai neištrinant esamų. Rašoma byla didėja. Todėl rezultatams skirtą bylą pradžioje reikia išvalyti. Tą atlieka sakinys

```
rewrite(b)
```

Kai atliekamas šis sakinys, byla `b` išvaloma – tampa tuščia ir paruošta rezultatų rašymui. O jeigu diske nėra bylos, susietos su byla `b`, tai tokia byla sukuriama.

Pateiksime pavyzdžių.

Pavyzdys. Byloje `SKAIČIAI.TXT` yra trys sveikieji skaičiai. Parašysime programą, kuri pirmąjį skaičių iš minėtos bylos perrašys į bylą `PIRMA.TXT`, o kitus du – į bylą `ANTRA.TXT`. Pradinių duomenų byla saugoma disko `A:` pagrindiniame kataloge, rezultatų bylos – disko `C:` kataloge `DUOMENYS`.

```
program TrysBylos;
  var prad,                          { pradinių duomenų byla }
      pirma, antra: text;            { rezultatų bylos }
      x, y: integer;                 { perskaityti skaičiai }
begin
  assign(prad, 'A:\SKAIČIAI.TXT'); reset(prad);
  assign(pirma, 'C:\DUOMENYS\PIRMA.TXT'); rewrite(pirma);
  assign(antra, 'C:\DUOMENYS\ANTRA.TXT'); rewrite(antra);
  read(prad, x); writeln(pirma, x);
  read(prad, x, y); writeln(antra, x, y: 10)
end.
```

Operacinės sistemos bylų vardai ir programoje aprašyti bylų vardai yra nepriklausomi vieni nuo kitų: jie gali sutapti, gali būti skirtingi. Be to, operacinės sistemos bylų vardai sudaromi pagal kitokia taisykles, negu Paskalio programose naudojami vardai.

Su bylomis yra susiję dvi integruotos funkcijos:

eoln(b) – eilutės pabaiga,

eof(b) – bylos pabaiga.

Tai funkcijos, kurių argumentai yra bylos, o rezultatai – loginės reikšmės (apie logines reikšmes kalbėsime kitame skyrelyje).

eoln(b) = *true*, jeigu bylos *b* duomenų žymeklis rodo eilutės pabaigos simbolį ir
eoln(b) = *false*, priešingu atveju;

eof(b) = *true*, jeigu bylos *b* duomenų žymeklis rodo bylos pabaigos simbolį ir
eof(b) = *false*, priešingu atveju.

Šias funkcijas naudosime vėliau, kai jų prireiks.

Uždaviniai

2.5.1. Parašykite programą, atvirkščią programai `TrysBylos`, t.y. tokią, kuri iš bylų `PIRMA.TXT` ir `ANTRA.TXT` perrašytų tris skaičius į vieną bylą `SKAIČIAI.TXT`.

2.5.2. Perrašykite 2.4 skyrelio 1 pavyzdžio programą `TiesLygtis` taip, kad pradiniai duomenys (lygties koeficientai) būtų skaitomi iš disko `C:` pagrindiniame kataloge esančios bylos `PRAD.XXX`, o rezultatas būtų rodomas ekrane.

2.6. Integruotos tekstinės bylos

Į Paskalį yra integruotos dvi tekstinės bylos, t.y. du tekstinių bylų tipo kintamuosius:

`input` – pradinių duomenų byla, susieta su klaviatūra,
`output` – rezultatų byla, susieta su vaizduoklio ekranu.

Taigi, klaviatūra ir vaizduoklio ekranas sutapatinami su bylomis. Kai kompiuteris skaito klaviatūra surinktus pradinius duomenis, jis elgiasi taip, lyg juos skaitytų iš bylos, pavadintos vardu `input`. Kai kompiuteris rašo rezultatus į ekraną, jis elgiasi taip, lyg juos rašytų į bylą, pavadintą vardu `output`. Integruotos bylos nuo įprastų bylų skiriasi tuo, kad jų vardų galima nerašyti į skaitymo ir rašymo sakinius. Jeigu pirmasis skaitymo arba rašymo procedūros parametras yra ne byla, tai tada laikoma, kad reikia skaityti iš bylos `input` arba rašyti į bylą `output`. Jeigu kreipinyje į funkciją `eof` arba `eoln` nenurodytas parametras, tai laikoma, kad ta funkcija taikoma bylai `input`. Štai todėl anksčiau ir nepastebėdavome, kad skaitydami arba rašydami duomenis naudojamės integruotomis bylomis.

Integruotoms byloms taip pat galima taikyti procedūrą `assign`. Tai yra jas galima susieti su diske saugomomis bylomis.

Pavyzdys. Programą daugianario

$a_3x^3 + a_2x^2 + a_1x + a_0$

reikšmei rasti (žr. 2.4 skyr. 2 pavyzdį) perrašysime taip, kad integruotos bylos `input` ir `output` būtų susietos su bylomis, laikomomis diske.

```
program daugianaris;
var x,                  { daugianario kintamasis }
    a0, a1, a2, a3,     { daugianario koeficientai }
    p: real;            { daugianario reikšmė }
begin
  assign(input, 'DUOM.TXT'); reset(input);
  assign(output, 'REZ.TXT'); rewrite(output);
  read(x, a0, a1, a2, a3);
  p:= a0 + x*(a1 + x*(a2 + x*a3));
  writeln(p)
end.
```

Uždaviniai

2.6.1. Pavyzdyje nenurodytas nei diskas, nei katalogas, kuriame turi būti bylos `DUOM.TXT` ir `REZ.TXT`. Kaip Jūs manote, kur šios bylos turėtų būti saugomos?

2.6.2. Perrašykite 2.4 skyrelio 3 pavyzdžio programą laikas taip, kad integruotos bylos būtų susietos su bylomis, laikomomis diske.

3. LOGINIAI DUOMENYS IR JŲ VALDOMI VEIKSMAI

Logika yra programavimo pagrindas. Sąsaja tarp matematikos ir programavimo prasideda nuo matematinės logikos.

Loginiai duomenys valdo programoje užrašytų veiksmų atlikimo tvarką.

Sakiniai, kurių atlikimo tvarką reikia valdyti (pasirinkti vieną iš kelių sakinių, kartoti) jungiami į struktūrinius sakinius, dar vadinamus valdymo struktūromis. Šiame skyriuje pateiksime pilną valdymo struktūrų rinkinį, t.y. tokį, kurio pakanka, bet kurio sudėtingumo veiksmų atlikimo tvarkai išreikšti.

3.1. Loginiai duomenys

Loginiai duomenys turi tik dvi reikšmes, kurios Paskalio kalboje žymimos vardais `true` ir `false`. Tai reikšmės teiginio, apie kurį galima pasakyti, kad jis yra *teisingas* arba *klaidingas*. Jeigu teiginys teisingas, tai sakoma, kad jo loginė reikšmė yra `true`, jei klaidingas – `false`. Pavyzdžiui, teiginio „Dabar lyja“ reikšmė yra `true`, jeigu dabar iš tikrųjų lyja ir `false` – priešingu atveju. Teiginys „skaičius 24 yra lyginis“ yra visada teisingas, nes skaičius 24 iš tikrųjų lyginis. Taigi, šio teiginio reikšmė yra `true`.

Teiginius kartais patogu vadinti sąlygomis. Sakoma, kad sąlyga gali būti *tenkinama* (jos loginė reikšmė yra `true`) arba *netenkinama* (`false`). Sąlygos, išreikštos nelygybe 5>3, reikšmė yra visada `true`, nes skaičius 5 didesnis už skaičių 3. Sąlygos 5=3 reikšmė yra `false`, nes skaičiai 3 ir 5 nelygūs. O kokia bus sąlygos *a*>5 reikšmė, iš anksto pasakyti negalima, nes ji priklauso nuo kintamojo *a* reikšmės.

Loginiai duomenys gali būti loginių uždavinių pradiniai ir galutiniai duomenys. Tokius uždavinius tenka retai programuoti. Tačiau su loginiais duomenimis susiduriame kiekvienoje programoje, kai reikia valdyti atliekamų veiksmų eilės tvarką. Dėl to juos ir nagrinėjame šio skyriaus pradžioje.

Loginiai kintamieji, kaip ir kitų tipų kintamieji, žymimi vardais. Kad būtų galima juos atskirti nuo kitų tipų kintamųjų, aprašuose jie apibūdinami žodžiu `boolean`[*], pavyzdžiui,

```
var a, b, log: boolean;
```

Šiuo aprašu pasakoma, kad kintamieji, pažymėti vardais `a`, `b` ir `log`, yra loginiai.

Loginiams kintamiesiems galima priskirti tik logines (loginių reiškinių) reikšmes. Loginės reikšmės – tai loginės konstantos `false` ir `true`. Todėl prieskyros sakiniai

```
a := true;
b := false;
log := a
```

yra teisingi, nes kintamieji *a*, *b* ir *log* yra loginio tipo. Tuo tarpu sakiniai:

```
a := 15;
b := 54.12
```

yra neteisingi, nes loginio tipo kintamieji negali įgyti skaitinių reikšmių.

[*] Vardas `boolean` yra parinktas žymaus mokslininko Džordžo Būlio (George Boole), tyrinėjusio matematinę logiką, garbei. Šio mokslininko vardu yra pavadinta atskira algebros šaka – Būlio algebra, kuri nagrinėja loginių reikšmių algebrą.

Su loginiais duomenimis atliekamos loginės operacijos:

not inversija (ne),

and konjunkcija (ir),

or disjunkcija (arba).

Visos loginės operacijos Paskalyje žymimos baziniais žodžiais, kurie angliškai reiškia tą, ką rašėme skliaustuose (*ne*, *ir*, *arba*).

Panagrinėkime kiekvieną operaciją.

Inversija (**not**). Operando reikšmė paneigiama, t.y. jo loginė reikšmė pakeičiama priešinga:

```
not false = true,
not true = false.
```

Pavyzdžiui, vietoj

```
a <> b
```

galima rašyti

```
not(a = b)
```

arba vietoj

```
a <= b
```

galima rašyti

```
not(a > b).
```

Inversija yra vienvietė operacija, t.y. ji taikoma vienam operandui – vienai loginei reikšmei. Šiuo požiūriu ji panaši į minusą, kuriuo užrašyta vienvietė atimtis aritmetiniame reiškinyje:

```
-x
not a
```

Ir vienos, ir kitos operacijos ženklas rašomas prieš operandą. Minusas keičia skaičiaus ženklą priešingu, o inversija – loginę reikšmę priešinga.

Konjunkcijos (**and**) reikšmė yra lygi `true` tiktai tuo atveju, kai abiejų operandų reikšmės yra `true`. Visais kitais atvejais konjunkcijos reikšmė yra `false`:

```
false and false = false,
false and true  = false,
true  and false = false,
true  and true  = true.
```

Konjunkcijos rezultatus galima pavaizduoti lentele.

a	b	a **and** b
false	false	false
false	true	false
true	false	false
true	true	true

Disjunkcijos (**or**) reikšmė yra lygi `true`, jei bent vieno operando reikšmė yra `true`. Kitaip sakant, disjunkcijos reikšmė, yra `true`, jei pirmojo *arba* antrojo operando reikšmė yra `true` (dėl to disjunkcija kartais vadinama operacija *arba*):

```
false or false = false,
false or true  = true,
```

```
true  or false = true,
true  or true  = true.
```

Disjunkcijos rezultatus galima pavaizduoti lentele.

a	b	a **or** b
false	false	false
false	true	true
true	false	true
true	true	true

Loginiai reiškiniai, panašiai kaip ir aritmetiniai, gali būti ir sudėtingesni, pavyzdžiui,

```
a and b and c,
a or b or c.
```

Loginių operacijų atlikimo tvarką nurodo skliaustai. O jeigu skliaustų nėra, tai operacijos atliekamos šia prioritetų eile:

not

and

or

T.y. pirmiausiai atliekamas neigimas, po to – konjunkcija ir paskiausiai – disjunkcija.

Vienodo prioriteto operacijos atliekamos iš kairės į dešinę. Pavyzdžiui, reiškinio

```
a or b or c or d and d and f
```

reikšmė skaičiuojama taip, lyg būtų šitaip surašyti skliaustai:

```
((a or b) or c) or ((d and e) and f).
```

Loginę (loginio reiškinio) reikšmę galima rašyti (parodyti ekrane), priskirti loginio tipo kintamajam, panaudoti valdymo struktūrose (žr. tolesnius skyrelius).

Programoje loginės reikšmės dažniausiai atsiranda kaip skaičių (aritmetinių reiškinių) lyginimo rezultatas. Vartojamos 6 lyginimo operacijos, kurios Paskalio kalboje žymimos šitaip:

`<` mažiau,

`<=` mažiau arba lygu ≤,

`=` lygu,

`<>` nelygu ≠,

`>` daugiau,

`>=` daugiau arba lygu ≥.

Pateiksime lyginimo operacijų ir jų rezultatų pavyzdžių.

```
5 < 6          true
5 > 6          false
5 > 5          false
5 >= 5         true
5-1 < 5        false
a-b = (c-d)-r  reikšmė priklausys nuo kintamųjų reikšmių
5.31 < 5.32    true
5.31 < 5       false
```

Lyginimo operacijų, panašiai kaip ir aritmetinių, operandai gali būti skirtingo tipo skaičiai (aritmetiniai reiškiniai). Tokiu atveju sveikasis skaičius pakeičiamas realiuoju ir lyginami du realieji skaičiai.

Loginių operacijų operandai dažnai būna lyginimo operacijų rezultatai. Jei nėra skliaustų, Paskalyje lyginimo operacijos atliekamos paskiausiai. Todėl lyginimo reiškinius, kai su jais atliekamos loginės operacijos, reikia suskliausti. Pavyzdžiui, reiškinyje

```
(x > 5) and (x < 10)
```

skliaustai reikalingi. Jei jų nebūtų, tai apskaičiuojant reiškinį

```
x > 5 and x < 10
```

pagal Paskalyje priimtus operacijų prioritetus pirmiausiai reikėtų atlikti loginę operaciją

```
5 and x
```

O tai neatitinka mūsų užmanymo. Be to, pirmasis konjunkcijos operandas yra neleistino tipo (todėl šią klaidą aptiktų kompiliatorius).

Kelias paprastesnes sąlygas loginėmis operacijomis galima sujungti į vieną sudėtingesnę. Pavyzdžiui, dviguba nelygybe $a<x<b$ matematikoje nurodomi skaičiaus x reikšmės rėžiai. Programavime toks užrašas neleistinas. Mat išeitų, kad pirmosios lyginimo operacijos $a<x$ rezultatas, kuris yra loginė reikšmė, vėliau lyginamas su skaičiumi b, o skaičiai ir loginės reikšmės yra nepalyginami duomenys. Dvigubą nelygybę programavime galima pakeisti dviejų viengubų nelygybių konjunkcija:

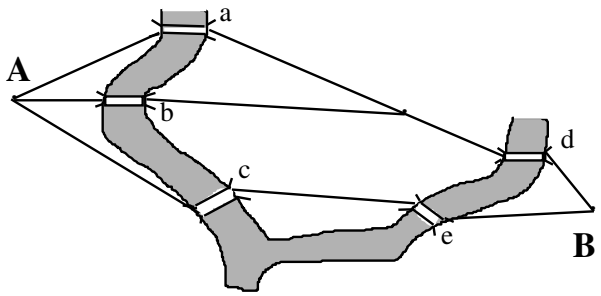
```
(a < x) and (x < b)
```

Sąlyga tenkinama, t.y. čia parašyto reiškinio reikšmė yra `true`, tik tuo atveju, jeigu tenkinamos abi sąlygos, sujungtos konjunkcijos operacija: *ir* pirmoji, *ir* antroji. Dėl to konjunkcija kartais vadinama operacija *ir*.

Lyginti galima ne tik aritmetinius reiškinius (skaičius), bet ir visų kitų paprastųjų tipų duomenis (apie juos kalbėsime vėliau), tarp jų – ir loginius duomenis. Laikoma, kad reikšmė `false` yra mažesnė už `true`, t.y. tenkinama sąlyga `false < true`.

Pateiksime programų ar jų fragmentų pavyzdžių.

1 pavyzdys. Miestus A ir B jungia keliai, pažymėti linijomis (14 pav.). Keliai eina per penkis tiltus, pažymėtus loginiais kintamaisiais `a`, `b`, `c`, `d` ir `e`. Jeigu tiltu galima važiuoti, tai jį žyminčio kintamojo reikšmė yra `true`, jeigu ne (pavyzdžiui, tiltas remontuojamas), tai `false`.



14 pav. Tiltai

Loginiam kintamajam `kelias` priskiriama reikšmė `true` jeigu miestus A ir B jungia bent vienas kelias, einantis per veikiančius tiltus:

```
kelias := ((a or b) and d) or c and e
```

2 pavyzdys. Programa, nustatanti, ar metai olimpiniai.

Pirmosios vasaros olimpinės žaidynės įvyko 1896 m. Atėnuose. Po to jos vyko arba turėjo vykti kas ketveri metai: 1900 m. – antrosios, 1904 m. – 3-iosios ir t.t. Neįvykusioms žaidynėms taip pat buvo skiriamas eilės numeris, o jų metai vis tiek laikomi olimpiniais.

Pradinis duomuo – skaičius, reiškiantis metus. Rezultatas – loginė reikšmė `true`, jeigu metai olimpiniai, arba `false`, jeigu metai neolimpiniai.

```
program olimpiada;
  var metai: integer;
      olimp: boolean;   { ar metai olimpiniai }
begin
  read(metai);
  olimp := (metai >= 1896) and (metai mod 4 = 0);
  writeln(olimp)
end.
```

Jeigu kompiuteriui pateiksime pradinį duomenį `1999`, tai jis ekrane parodys rezultatą

```
FALSE
```

nes 1999-ieji metai neolimpiniai.

Jei kompiuteriui pateiksime skaičių `2000`, tai ekrane pamatysime žodį

```
TRUE
```

Aiškiau būtų, jeigu ekrane išvystume informatyvesnį pranešimą, pavyzdžiui,

`OLIMPINIAI` arba `NEOLIMPINIAI`

Apie tokių (alternatyvių) pranešimų formavimą kalbėsime 3.3 skyrelyje.

Pateiktas pavyzdys iliustruoja, kaip į ekraną rašomos loginės reikšmės. Tačiau nei Paskalio standartas, nei Turbo Paskalis nenumato loginių reikšmių skaitymo. Mat loginės reikšmės neturi visuotinai priimtų žymenų, kaip, pavyzdžiui, skaičiai. Paskalio programų tekstuose vartojami loginių reikšmių vardai `true` ir `false` gali pasirodyti per ilgi tiems, kas ruošia pradinius duomenis. Todėl ten jie dažnai žymimi trumpiau (T ir F arba 0 ir 1). Tokiais atvejais rašoma loginių duomenų skaitymo programa, pritaikyta konkrečiam loginių reikšmių žymėjimui.

Uždaviniai

3.1.1. Duotas kintamųjų aprašas:

```
var a, b, c: integer;
    x, y, z: boolean;
```

Kurie iš šių sakinių yra neteisingi ir kodėl?

a) `b := true;`

b) `a := x;`

c) `x := a;`

d) `x := a - b;`

e) `x := c = y;`

f) `x := y + a;`

g) `c := y + 2;`

h) `a := b = c;`

i) `c := a + b.`

3.1.2. Ką parašys kompiuteris pagal šitokią programą?

```
program logika;
  var a, b: integer;
      aa, bb, cc: boolean;
```

```
begin
  a := 3; b := 5;
  aa := a < b;
  bb := a > b;
  cc := aa;
  writeln(aa);
  writeln(bb);
  writeln(cc)
end.
```

3.1.3. Sakėme, kad dviguba nelygybė $a<x<b$ su aritmetiniais duomenimis neleistina. O ar ji leistina su kurio nors kito tipo duomenimis?

3.1.4. Parašykite šešių loginių reikšmių lyginimo operacijų <, <=, = , <>, >, >= rezultatų lenteles (analogiškas konjunkcijos ir disjunkcijos lentelėms).

3.1.5. Kintamųjų reikšmės yra tokios: $a = 10$, $b = 20$, *log* = *true*, *lg* = *false*. Kokios šių loginių reiškinių reikšmės:

```
a) log or lg;
b) log and lg;
c) (a = 10) and (b = 20);
d) (a <> 10) or (b = 20);
e) (a > 5) and (b > 5) and (a < 20) and (b < 30);
f) (a > 5) and (b > 5) and (a < 10) and (b < 30);
g) (a > 5) and (b > 5) or (a < 10) and (b < 30);
h) (not (a < 15)) or (not (b < 30));
i) not (a = b);
k) not (not (a = b));
l) not (not (not (a = b)))?
```

3.1.6. Pradiniai duomenys – trys skaičiai `a`, `b` ir `c`. Parašykite loginį reiškinį, kurio reikšmė būtų `true`, tada ir tiktai tada, kai:

a) visų trijų kintamųjų `a`, `b` ir `c` reikšmės lygios;

b) visų trijų kintamųjų `a`, `b` ir `c` reikšmės skirtingos;

c) kurių nors dviejų kintamųjų reikšmės lygios;

d) visų trijų kintamųjų `a`, `b` ir `c` reikšmės yra lyginiai skaičiai;

e) visų trijų kintamųjų reikšmės yra teigiamos, bet ne didesnės kaip 100.

3.1.7. Pradiniai duomenys – keturi skaičiai `a`, `b`, `c` ir `d`. Parašykite loginį reiškinį, kurio reikšmė būtų `true`, tada ir tiktai tada, kai:

a) visų keturių kintamųjų `a`, `b`, `c` ir `d` reikšmės išdėstytos didėjančiai, t.y.

```
a < b < c < d
```

b) visų keturių kintamųjų `a`, `b`, `c` ir `d` reikšmės išdėstytos nemažėjančiai, t.y.

```
a <= b <= c <= d
```

3.1.8. Parašykite loginį reiškinį, kurio reikšmė būtų `true`, jeigu iš trijų atkarpų. kurių ilgiai duoti (yra kintamųjų `a`, `b`, ir `c` reikšmės), galima sudaryti:

a) trikampį,

b) lygiašonį trikampį,

c) lygiakraštį trikampį.

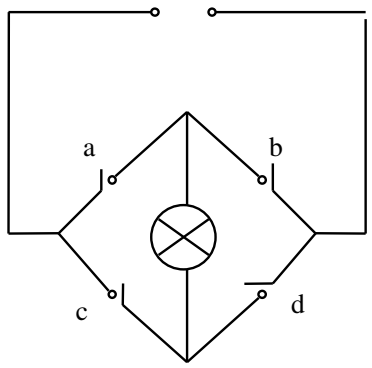
3.1.9. Apskritimo centro koordinatės yra *cx* ir *cy* bei spindulys *r*. Taško koordinatės *yra tx* ir *ty*. Parašykite loginį reiškinį, kurio reikšmė būtų `true`, jeigu taškas yra apskritimo viduje.

3.1.10. Elektros lemputė, pažymėta X, jungiama į tinklą per 15 paveiksle parodytą keturių jungiklių schemą. Jungiklių būsenos pažymėtos loginiais kintamaisiais *a*, *b*, *c* ir *d*. Jeigu jungiklis įjungtas (per jį teka elektros srovė), tai jį atitinkančio kintamojo reikšmė yra `true`, jei išjungtas (srovė neteka) – `false`. Parašykite loginį reiškinį, kurio reikšmė būtų `true`, jeigu:

a) lemputė šviečia;

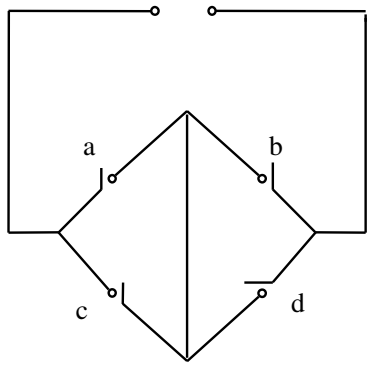
b) yra trumpas jungimas schemoje;

c) lemputė nešviečia ir srovė neteka per schemą.



15 pav.

3.1.11. Ankstesnio uždavinio schemoje lemputė pakeista laidu (16 pav.). Parašykite loginį reiškinį, kurio reikšmė būtų `true`, jeigu grandine teka elektros srovė.



16 pav.

3.1.12. Parašykite loginį reiškinį, kurio reikšmė būtų `true`, jeigu metai `m` keliamieji.

Ar metai keliamieji, nustatoma pagal tokias taisykles: 1) jeigu metai nėra šimtmečio metai, tai jie yra keliamieji, jeigu dalosi iš 4; 2) jeigu metai yra šimtmečio metai, tai jie yra keliamieji, jeigu šimtų skaičius dalosi iš 4 (pvz., 2000 metai yra keliamieji, o 2100 metai – ne keliamieji).

3.1.13. Turime sveikųjų skaičių tipo kintamąjį *a* ir šitokius loginius reiškinius:

```
a+1 > a
a+1 > 1
a*5 > 0
a*a > 0
```

`a*a >= 0`

Ties reiškiniu parašykite:

žodį `true`, jeigu jo reikšmė yra visada `true`;

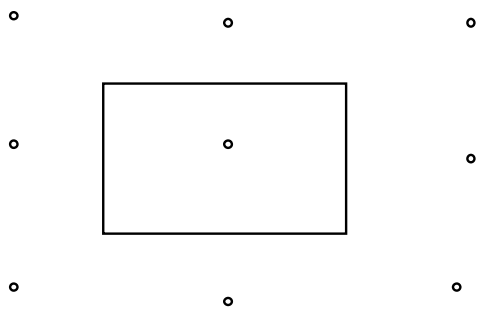
žodį `false`, jeigu jo reikšmė yra visada `false`;

žodžius `false true`, jeigu gali būti vienaip ir kitaip.

Praktikos darbas

3.1.1. Stačiakampis. Parašykite programą, kuri parodytų ekrane žodį TRUE, jeigu taškas yra viduje stačiakampio, kurio kraštinės lygiagrečios koordinačių ašims. Pradinius duomenis parinkite patys ir tokius, kad jų būtų kuo mažiau (pakanka 6 skaičių – trijų taškų koordinačių).

Parinkite keletą pradinių duomenų rinkinių, tokių, kad jais būtų galima visapusiškai patikrinti programą. Keletas taško padėčių stačiakampio atžvilgiu parodyta 17 paveiksle.



17 pav.

Taškui apibūdinti reikia dviejų koordinačių (*tx* ir *ty*). Stačiakampiui apibūdinti pakanka dviejų kurių nors priešingų jo kampų koordinačių (*ax*, *cx*, *ay*, *cy*).

3.2. Loginių reiškinių pertvarkymas

Pirmą kartą skaitant knygą šį skyrelį galima praleisti. Prie jo bus naudinga grįžti, kai pajusite, kad loginius reiškinius rašote per ilgus ir negražius ir norėsite juos suprastinti.

Programavime susiduriame bent su dviejų rūšių reiškiniais: aritmetiniais ir loginiais. Aritmetiniuose reiškiniuose vartojame aritmetines operacijas. Šių operacijų savybes gerai žinome. Jas išmokstame mokykloje. Jomis remdamiesi pertvarkome aritmetinius reiškinius – sutraukiame panašius narius, iškeliame bendrą dauginamąjį prieš skliaustus ir pan.

Panašios taisyklės taikomos ir loginiams reiškiniams. Tiktai loginių operacijų savybės šiek tiek skiriasi nuo aritmetinių. Todėl ir loginių reiškinių pertvarkymo taisyklės šiek tiek skiriasi (nors yra ir tokių pat) nuo aritmetinių reiškinių pertvarkymo taisyklių. Pateiksime svarbesnes.

1. Sukeitus vietomis konjunkcijos arba disjunkcijos operandus, rezultatas nepasikeičia:

```
a and b = b and a,
a or b = b or a.
```

2. Vienodos operacijos atliekamos bet kuria tvarka (vadinasi, jų operandus galime grupuoti kaip norime):

```
a and (b and c) = (a and b) and c,
a or  (b or c)  = (a or b)  or c.
```

3. Vienodus operandus galima iškelti už skliaustų:

```
(a and b) or  (a and c) = a and (b or c),
(a or  b) and (a or  c) = a or  (b and c).
```

Pirmosios dvi taisyklės atitinka sudėties ir daugybos dėsnius. Aritmetiniuose reiškiniuose prieš skliaustus galima įkelti tik bendrus dauginamuosius, o logikoje – bet kurios operacijos (konjunkcijos arba disjunkcijos) operandus. Apskritai, kiekviena logikos taisyklė, taikoma konjunkcijai, tinka ir disjunkcijai (ir atvirkščiai). Tačiau, taikant taisyklę kitai operacijai, reikia tapatybėje (formulėje) visas operacijas ir visas konstantas pakeisti priešingomis (**and** ⇒ **or**, **or** ⇒ **and**, `false` ⇒ `true`, `true` ⇒ `false`) ir galbūt pakeisti skliaustų išdėstymą taip, kad išliktų ankstesnė operacijų atlikimo tvarka. Šis operacijų keitimo principas vadinamas dualumo principu. Jis labai praverčia besimokančiam – pakanka žinoti vieną taisyklę, o jos „antrininkę“ galima gauti taikant dualumo principą.

4. Jei operandai sutampa, tai rezultatas lygus operandui:

```
a and a = a,
a or  a = a.
```

5. Jei vienas operandas konstanta, tai rezultatas lygus vienam iš operandų:

```
a and true  = a,
a and false = false,
a or  false = a,
a or  true  = true.
```

6. Konjunkciją galima pakeisti disjunkcija (ir atvirkščiai) įkeliant į skliaustus (iškeliant iš jų) inversiją:

```
not (a and b) = not a or  not b,
not (a or  b) = not a and not b.
```

7. Operando ir jo inversijos konjunkcijos (disjunkcijos) rezultatas yra konstanta:

```
a and not a = false,
a or  not a = true.
```

8. Dvi iš eilės einančios inversijos panaikina viena kitą:

```
not not a = a.
```

1 pavyzdys. Suprastinsime loginį reiškinį

```
a and not b
```

Pritaikę 6 taisyklę gauname:

```
not a or not not b.
```

Antrajam operandui pritaikę 8 taisyklę, gauname:

```
not a or b.
```

Daug kartų pertvarkydami ilgą reiškinį, galime suklysti. Norėdami įsitikinti, ar nepadarėme klaidų, vietoj kintamųjų įrašykime į reiškinius konkrečias reikšmes ir apskaičiuokime tų reiškinių reikšmes. Jei reiškinį pertvarkėme teisingai, tai jo reikšmės prieš pertvarkymą ir po jo sutampa esant bet kokioms kintamųjų reikšmėms. Kadangi loginis kintamasis gali įgyti tik dvi reikšmes, tai reiškinį, kuriame yra `n` kintamųjų, reikės tikrinti 2^n kartų.

2 pavyzdys. Tarkime, kad pertvarkę reiškinį

```
(a or b) and (a or not b) or b
```

gavome paprastesnį reiškinį:

```
a or b.
```

Patikrinkime, ar pertvarkydami nepadarėme klaidų, t.y. ar šių dviejų reiškinių reikšmės sutampa, esant bet kokioms jų kintamųjų `a` ir `b` reikšmėms:

a	b	(a **or** b) **and** (a **or not** b) **or** b	a **or** b
false	false	false	false
false	true	true	true
true	false	true	true
true	true	true	true

Kaip matome, visais atvejais pradinio ir pertvarkyto reiškinio reikšmės sutampa. Vadinasi, pradinis reiškinys buvo pertvarkytas teisingai.

Uždaviniai

3.2.1. Suprastinkite šiuos reiškinius:

a) **not** (a **or** b) **and not** (a **and** b)

b) a **and** b **or not** a **or not** (b **or not** a)

3.3. Vienas iš dviejų veiksmų

Gyvenime dažnai atsiduriame kryžkelėse, kai reikia pasirinkti, kuriuo keliu eiti. Tada žmogus sustoja, svarsto, pagaliau pasirenka.

Pasirinkimas – įprasta situacija programavime. Tiktai kompiuteris nesustoja ir nesvarsto – visi galimi keliai turi būti iš anksto numatyti ir į programą surašytos vienareikšmės jų parinkimo sąlygos.

Kompiuterio prigimtis dvejetainė. Todėl dažniausiai pasirenkamas vienas kelias iš dviejų. Vieno veiksmo iš dviejų parinkimas nurodomas sąlyginiu sakiniu, kurio pavidalas šitoks:

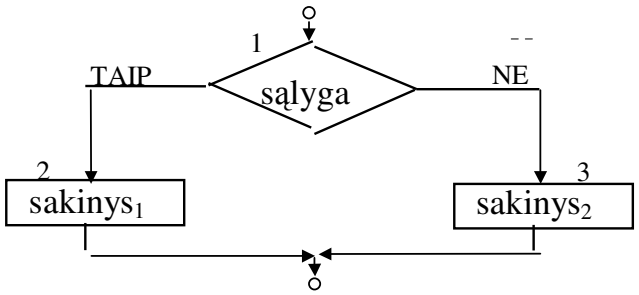
if *loginis reiškinys* **then** $sakinys_1$
else $sakinys_2$

Po žodžio **if** einantis loginis reiškinys dar vadinamas *sąlyga*.

Atliekamas tik vienas iš dviejų sakinių:

• sakinys, einantis po žodžio **then** ($sakinys_1$), jeigu loginio reiškinio reikšmė yra *true* (sąlyga tenkinama) arba

• sakinys, einantis po žodžio **else** ($sakinys_2$), jeigu loginio reiškinio reikšmė yra `false` (sąlyga netenkinama). Sakinių pasirinkimas grafiškai pavaizduotas 18 paveiksle



18 pav. Sąlyginio sakinio schema

1 pavyzdys. Didesniojo skaičiaus radimas.

```
program didesnysis;
```

```
  var a, b, max: integer;
begin
  read(a, b);
  if a >= b then max := a
            else max := b;
  writeln(max)
end.
```

Jei pradiniai duomenys būtų skaičiai 5 ir 6, tai būtų atliekami šie veiksmai:

	a	b	max	
read(a, b)	5	6	?	skaitomi pradiniai duomenys
a >= b	5	6	?	tikrinama sąlyga
max := b	5	6	6	
writeln(max)	5	6	6	rašomas rezultatas

Kad aiškiau matytųsi sąlyginio sakinio šakos, programoje jas rašome viena po kitos, t.y. žodį **else** lygiuojame su jį atitinkančiu pirmosios šakos žodžiu **then**. Kartais patogu sąlygai (su žodžiu **if**) skirti atskirą eilutę (ypač kai ilgesnė sąlyga), pavyzdžiui,

```
if (alfa*alfa) >= (beta*beta)
  then ...
  else ...
```

2 pavyzdys. Programa, nustatanti, ar skaičius dalus iš 7.

```
program dalus7;
  var x: integer;
begin
  read(x);
  write('Skaičius ', x);
  if x mod 7 = 0
    then write(' dalus')
    else write(' nedalus');
  writeln(' iš 7')
end.
```

Jei kompiuteriui pateiksime skaičių 12345, tai gausime atsakymą:

```
Skaičius 12345 nedalus iš 7
```

3 pavyzdys. Programa olimpinių žaidynių eilės numeriui nustatyti.

Panašų uždavinį jau sprendėme (žr. 3.1 skyr. 2 pavyzdį). Ten nustatėme, tik patį faktą, ar metai olimpiniai. Panaudodami sąlyginį sakinį galėsime nustatyti ir žaidynių eilės numerį. Neįvykusioms žaidynėms skiriamas eilės numeris, o jų metai vis tiek laikomi olimpiniais.

Pradinis duomuo – skaičius, reiškiantis metus. Rezultatas – olimpinių žaidynių numeris, jeigu metai olimpiniai, arba nulis, jeigu metai neolimpiniai.

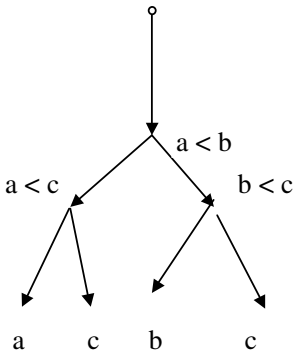
```
program olimpiadanr;
  var metai,
      nr: integer; { olimpiados eilės numeris }
begin
  read(metai);
  if (metai >= 1896) and (metai mod 4 = 0)
    then nr := (metai - 1896) div 4 + 1
    else nr := 0;
  writeln(nr)
end.
```

Sąlyginiame sakinyje (po žodžių **then** ir **else**) gali eiti bet kokie sakiniai, tarp jų ir sąlyginiai. Tada šakos šakojasi į naujas šakas ir gaunamas medis.

4 pavyzdys. Pradiniai duomenys – trys skaičiai. Programa mažiausiam iš jų rasti.

```
program minimumas;
  var a, b, c, min: integer;
begin
  read(a, b, c);
  if a < b then if a < c then min := a
                         else min := c
           else if b < c then min := b
                         else min := c;
  writeln(min)
end.
```

Priklausomai nuo to, ar tenkinama sąlyga a < b, atliekama viena kuri nors gaubiančiojo sąlyginio sakinio šaka. Kiekvieną šaką sudaro naujas sąlyginis sakinys, turintis dvi šakas (19 pav).



19 pav. Veiksmų šakojimasis 3 pavyzdžio sąlyginiame sakinyje

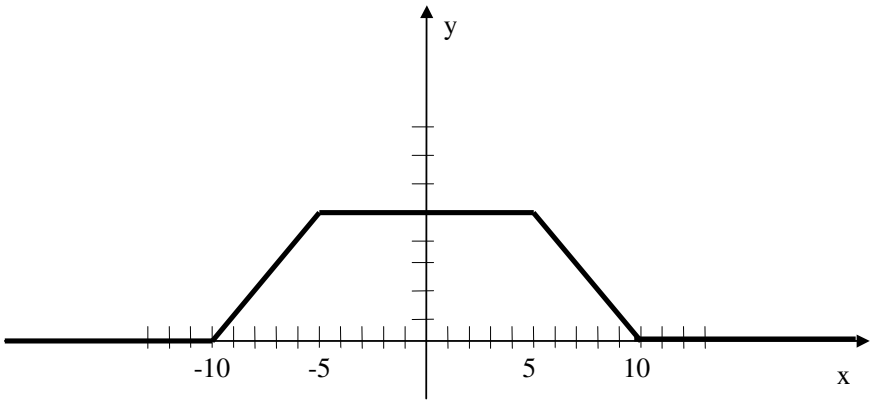
Nebūtinai kiekvienas sąlyginis turi būti simetriškas – viena sąlyga gali šakotis daugiau, kita – mažiau.

Lengviau parašyti ir suvokti tokį nesimetrišką sąlyginį sakinį, kuriame veiksmai šakojasi tik po **else.** Šitaip nuo daugelio galimų kelių (galbūt iki galo dar neišnagrinėtų) atskiriamas vienas, kurio veiksmai jau aiškūs ir tie veiksmai užrašomi po **then**, o visa kita, kas dar toliau lieka skaldyti į šakas, rašoma po **else**.

5 pavyzdys. Užrašysime sakinius kintamojo *y* reikšmei, kuri priklauso nuo kintamojo x reikšmės ir apskaičiuojama pagal šitokią formulę:

$$y = \begin{cases} 0, & \text{jei } x < -10, \\ 10 + x, & \text{jei } -10 <= x <= -5, \\ 5, & \text{jei } -5 < x < 5, \\ 10 - x, & \text{jei } 5 <= x <= 10, \\ 0, & \text{jei } x > 10. \end{cases}$$

Kaip kintamojo *y* reikšmė priklauso nuo kintamojo *x* reikšmės, rodo grafikas (20 pav.).

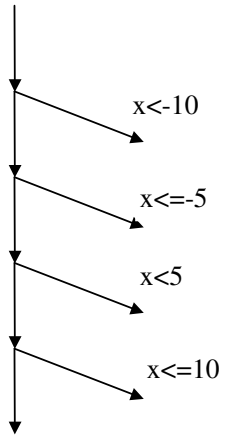


20 pav.

Sąlyginiu sakiniu ta priklausomybė išreiškiama taip:

```
if x < −10 then y := 0
  else if x <= −5 then y := 10 + x
    else if x < 5 then y := 5
      else if x <= 10 then y := 10 − x
                      else y := 0
```

Veiksmų šakojimasis grafiškai pavaizduotas 21 paveiksle.



21 pav.

Pastebėsime, kad sąlygos, einančios po žodžių **`else if`**, paprastesnė už sąlygas, esančias uždavinio formulėje. Mat kintamojo *x* reikšmės rėžį pakanka tikrinti tik iš vienos pusės – iš kitos pusės jis jau buvo patikrintas prieš patenkant į tą sakinio šaką.

Sąlyginis sakinys, kuriame po žodžių **`else`** įterpiami vis nauji sąlyginiai sakiniai, žodžius **else** lygiuojant su juos atitinkančiais **`then`**, labai išsitęstų į dešinę. Programos vaizdumas nenukenčia, jeigu sakinio dalis, prasidedančias žodžių pora **`else if`**, rašome šiek tiek patrauktas į dešinę.

Uždaviniai

3.3.1. Kokios bus kintamųjų `a` ir `b` reikšmės, atlikus šitokią sakinių seką?

```
a := 10; b := 6;
if a > b then a := a − b
         else a := a + 3;
```

```
if a > 6 then b := b + 1
         else b := b + 2
```

3.3.2. Atlikus sakinį

```
if a > b then a := a - b
         else b := b - a
```

gautos šitokios kintamųjų `a` ir `b` reikšmės: `a` = 5, `b` = 5. Kokios galėjo būti šių kintamųjų reikšmės, prieš atliekant sąlyginį sakinį?

3.3.3. Turime tokią programą mažiausiam iš trijų skaičių rasti.

```
program minim;
  var a, b, c, min: integer;
begin
  read (a, b, c);
  if (a < b) and (a < c) then min := a
    else if (b < c) and (b < a) then min := b
                                else min := c;
  writeln(min)
end.
```

Ar ji teisinga? Jei taip, ar ji visada duos tokį pat rezultatą, kaip ir programa `minimumas` (žr. 3 pavyzdį).

3.3.4. Parašykite sąlyginį sakinį rezultato f reikšmei rasti pagal formulę

$$f = \begin{cases} a + b, & \text{jei a – nelyginis,} \\ a \times b, & \text{jei a – lyginis.} \end{cases}$$

3.3.5. Duotas sąlyginis sakinys

```
if a < 3
  then c := 1
  else if a > = 5 then c := 2
                  else c := 3
```

Kokiai kintamojo `a` reikšmei esant, kintamajam `c` bus priskirta reikšmė 3?

3.3.6. Programą `olimpiadanr` pakeiskite taip, kad būtų spausdinamas šitokio pavidalo tekstas:

```
1999 metai yra neolimpiniai
2000 metai yra olimpiniai
```

3.3.7. Programoje `dalus7` į ekraną rašomo teksto eilučių pradžiose ir pabaigose yra tarpų. Kam jie reikalingi? Kaip atrodytų rašomas į ekraną tekstas, jeigu juos pašalintume?

3.3.8. Parašykite programą, kuri atliktų tokius veiksmus su pradiniu duomeniu: jeigu jis teigiamas, tai jį paverstų tokio pat modulio neigiamu skaičiumi, jeigu neigiamas – paverstų jo moduliu.

Praktikos darbas

3.3.1.Programos atlikimas pažingsniui. Programą `minimumas` (žr. 4 pavyzdį) papildykite dialogo veiksmais ir išbandykite su kompiuteriu, kai pradiniai duomenys yra šitokie:

```
 5  6 10
 5  6  2
25 17 18
25 17 16
```

Stebėkite kokie sakiniai kokiu atveju yra atliekami. Tam panaudokite pažingsninį kompiliatoriaus režimą.

Tą patį padarykite su programa `olimpiadanr` (žr. 3 pavyzdį). Pradinius duomenis programos išbandymui pasirinkite patys.

3.4. Įvairesni šakojimosi atvejai

Ne visada visi šakos veiksmai išreiškiami vienu sakiniu. Būna atvejų, kai reikia rašyti kelis sakinius, o kartais šaka būna tuščia – be jokių veiksmų. Tokius atvejus ir panagrinėsime.

Tuščias sakinys. Dažnai veiksmą reikia atlikti tik vienu atveju (pavyzdžiui, kai sąlyga tenkinama), o priešingu atveju nereikia nieko daryti. Pavyzdžiui, jeigu reikia kintamojo `a` reikšmę pakeisti jos moduliu, tai skaičiaus ženklą reikia keisti tik tada, kai jis neigiamas, o jeigu jis teigiamas, tai nereikia nieko daryti. Tokius veiksmus galima užrašyti bet kuriuo iš tokių sąlyginių sakinių:

```
if a < 0 then a := -a
         else;
```

arba

```
if a >= 0 then
          else a := -a
```

Ten, kur nereikia atlikti veiksmų, nieko ir nerašome. Tokia tuščia vieta vadinama *tuščiu sakiniu*. Pirmuoju atveju tuščias sakinys „parašytas“ po **`else`**, antruoju – po **`then`**.

Sakiniai skiriami kabliataškiais. Jeigu netyčia tarp gretimų dviejų sakinių parašysime ne vieną, o du (ar kelis) kabliataškius, tai klaidos nebus – bus laikoma, kad tenai, tarp kabliataškių yra tušti sakiniai.

Sutrumpintas sąlyginis sakinys. Kai tuščias sakinys yra po **`else`**, tai galima praleisti ir žodį **`else`**, pavyzdžiui,

```
if a < 0 then a := -a
```

Šitokia Paskalio kalbos konstrukcija vadinama *sutrumpintu sąlyginiu sakiniu*. Jo bendrasis pavidalas yra šitoks:

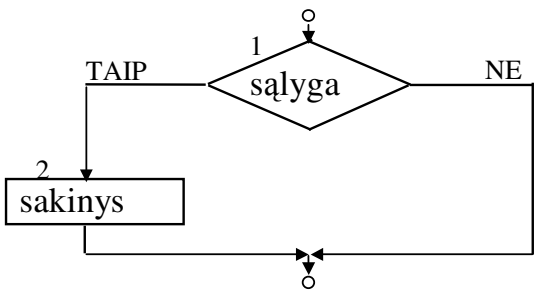
`if` *loginis reiškinys* **`then`** *sakinys*

Sutrumpinto sąlyginio sakinio schema pateikta 22 paveiksle.

Vienas į kitą įdėti sutrumpinti sąlyginiai sakiniai. Tarkime, kad turime šitaip netvarkingai į vieną eilutę surašytus vienas į kitą įdėtus sakinius:

`if` b1 **`then if`** b2 **`then`** S1 **`else`** S2

Kuriam sakiniui priklauso šaka **`else`** S2: pirmajam (**`if`** b1 **`then`**), ar antrajam (**`if`** b2 **`then`**). Paklausti galima ir kitaip: kurį **`then`** atitinka **`else`**, arba kuris iš šių dviejų vienas į kitą įdėtų sąlyginių sakinių yra sutrumpintas?



22 pav. Sutrumpinto sąlyginio sakinio schema

Yra susitarta tokį užrašą interpretuoti šitaip:

```
if b1
  then if b2
         then S1
         else S2
```

T.y. vidiniai sakiniai laikomi (jeigu tik galima) nesutrumpintais.

Sudėtingesniam sąlyginiam sakiniui perskaityti galime rekomenduoti paprastą taisyklę – trūkstamus **else** surašyti sakinio gale. Tuomet nesunkiai ir vienareikšmiškai atrandami žodžių **if**, **then** ir **else** trejetai.

Vadinasi, sakinį

```
if b1 then if b2 then if b3 then S1 else S2
```

reikia suprasti šitaip:

```
if b1
  then if b2
         then if b3 then S1
                    else S2
         else
  else
```

Kai norime parašyti sudėtingesnį sakinį, galime rekomenduoti pradžioje surašyti visus **else**, sakinius taisyklingai sulygiuoti, o po to nereikalingus **else** išbraukti.

Sudėtinis sakinys. Kalbėdami apie sakinius, rašomus į bet kurią sąlyginio sakinio šaką, vartojome vienaskaitą. O ką daryti, kai vienu ar kitu atveju reikia atlikti ne vieną, o kelis sakinius?

Tada tie keli sakiniai rašomi tarp žodžių **begin** ir **end**. Šitaip sakinių grupė paverčiama vienu sakiniu, kuris vadinamas *sudėtiniu*. Jo schema yra tokia:

```
begin
 sakinys1;
 sakinys2;
 ...
 sakinysn
end
```

Sudėtinis sakinys – tai vienas sakinys, kurį galima rašyti visur ten, kur ir bet kurį paprastą sakinį.

1 pavyzdys.

```
if a < 0 then begin
                b := b + 1; c := c + 1
              end
         else begin
                b := b - 1; c := c - 1
              end
```

Kad būtų geriau matomos sudėtinio sakinio ribos, žodžiai **begin** ir **end** rašomi vienas po kitu ir vienodai atitraukiami nuo kairiojo eilutės krašto, o jiems priklausantys sakiniai truputį patraukiami į dešinę. Trumpo sudėtinio sakinio ribos gerai matomos ir tada, kai visas jis parašytas vienoje eilutėje, pavyzdžiui,

```
if a < 0
  then begin b := b + 1; c := c + 1 end
```

2 pavyzdys. Pradiniai duomenys – du skaičiai. Rezultatas apskaičiuojamas pagal formulę

$s = x^2 + y^3$;

čia x – didesnysis, o y – mažesnysis pradinis duomuo.

Sudarome programą:

```
program p;
  var a, b, x, y: integer;
begin
  read(a, b);
  if a < b
    then begin x := b;
               y := a
         end
    else begin x := a;
               y := b
         end;
  writeln(x*x + y*y*y)
end.
```

Sudėtinį sakinį gali sudaryti daugelis kitokių sakinių, tarp jų gali būti sąlyginių ir trumpesnių sudėtinių sakinių.

Kiekvienos programos veiksmų dalį sudaro sakiniai, parašyti tarp žodžių **begin** ir **end**. Taigi būsime teisūs, jeigu sakysime, kad kiekvienos programos visi veiksmai išreiškiami vieninteliu sudėtiniu sakiniu.

Skyrybos klaidos. Atkreipiame dėmesį į tai, kad prieš žodį **else** kabliataškio rašyti negalima. Jeigu jį ten padėtume, tai reikštų, kad po kabliataškio eina jau kitas sakinys (kabliataškis skiria sakinius), pvz.,

```
if b then S1; else S2
```

Kadangi žodžiu **else** negali prasidėti joks sakinys, tai tokią klaidą pastebės Turbo Paskalio kompiliatorius.

O jeigu sakinyje

```
if b then S
```

netyčia padėtume kabliataškį po žodžio **then**:

if *b* **then;** *S*

tai programa liktų taisyklinga – būtų laikoma, kad po **then** yra tuščias sakinys, kabliataškis atskiria sakinį *S* nuo sąlyginio. Sakinys *S* nebepriklauso sąlyginiam ir atliekamas visada. Tokios klaidos kompiliatorius nerastų, o mes stebėtumės, kodėl kompiuteris duoda ne tokį rezultatą, kokio tikimės.

Uždaviniai

3.4.1. Duota programa

```
program mini;
  var a, b, c, min: integer;
begin
  read(a, b, c);
  min := a;
  if b < min then min := b;
  if c < min then min := c;
  write(min)
end.
```

Ar visada kompiuteris išspausdins vienodus rezultatus, atlikęs programą `minim` ir `minimumas` (žr. 3.3 skyr. 4 pavyzdį), kai pradiniai duomenys tie patys?

3.4.2. Parašykite sąlyginius sakinius, pagal kuriuos būtų palyginamos kintamųjų *a* ir *b* reikšmės ir, jeigu jos nelygios:

a) iš didesniosios atimama mažesnioji;

b) mažesnioji padidinama vienetu;

c) didesnioji sumažinama vienetu.

3.4.3. Duota programa

```
program skaitymas;
  var a, b, c, d: integer;
begin
  read(a, b);
  if a < b
    then read(c, d)
    else if a > b
           then read(d, c);
   write(a, ' ', b, ' ', c, ' ', d)
end.
```

Pradiniai duomenys šitokie:

a) `1  2  3  4`

b) `4  3  2  1`

Ką išspausdins kompiuteris?

Nors ir apskaičiavote rezultatą, bet programa yra neteisinga: gali būti tokių pradinių duomenų, kuriems esant bus vartojamos neapibrėžtos kintamųjų reikšmės. Kokiems pradiniams duomenims esant tai atsitiks?

3.4.4. Pavartodami sudėtinius sakinius, suprastinkite šį programos fragmentą:

```
if a > b  then c := 1;
if a > b  then d := 2;
if a <= b then c := 3;
if a <= b then d := 4
```

3.4.5. Duota programa:

```
program pp;
  var a, b, xx, yy, s: integer;
begin
  read(a, b);
  xx := a*a;
  yy := b*b;
  if a < b then xx := xx*a
           else yy := yy*b;
  s := xx + yy;
  write(s)
end.
```

Ar programa `pp` atlieka tokius pat veiksmus, kaip ir 2-ojo pavyzdžio programa `p`?

3.4.6. Duoti du sudėtiniai sakiniai:

a)
```
begin
  c := 0;
  if a < 5 then c := 1;
  if a > 5 then c := 2
end;
```

b)
```
begin
  c := 0;
  if a < 5 then c := 1
           else c := 2
end
```

Ar atlikus šiuos sakinius visada rezultatas (kintamojo `c` reikšmė) yra tas pats? Jeigu ne, tai su kuria kintamojo `a` reikšme jie skiriasi?

3.4.7. Duota programa:

```
program perdaug;
  var a, b: integer;
begin
  read(a, b);
  if a < 10 then begin a := 10;
                       b := b - 5
                 end;
  if a < 5 then begin a := 5;
                      b := b - 5
                end;
  writeln(a, b: 6)
end.
```

Sutrumpinkite ją, pašalindami vieną nereikalingą sakinį.

3.4.8. Duota programa:

```
program intervalai;
  var a, b, c: integer;
begin
  read(a, b, c);
  if a <  0 then a := 2 - a;
  if b <  0 then b := 10;
  if c > 10 then c := 10;
  writeln(a, ' ', b, ' ', c)
end.
```

Nustatykite, kokiuose intervaluose bus rezultatų reikšmės. Intervalus nurodykite nelygybėmis. (Kokie rezultatai bus išspausdinti, be abejo, priklauso nuo pradinių duomenų. Tačiau ir nežinant jų, vien iš programos teksto galima šį tą pasakyti apie rezultatus.)

3.4.9. Taikydami logines operacijas, suprastinkite šiuos sakinius:

a)
```
if a > b then
    if b < c then a := a + 1;
```

b)
```
if log then
        else a := a + 1;
```

3.4.10. Programą `dalus` (žr. 3.3 skyr. 3 pavyzdį) pakeiskite taip, kad joje būtų tik vienas suprastintas sąlyginis sakinys.

Praktikos darbas

3.4.1. Vidurinysis skaičius. Pradiniai duomenys – trys sveikieji skaičiai. Parašykite programą, kuri rastų vidurinįjį skaičių, jeigu tie skaičiai būtų surikiuoti eilės tvarka (nemažėjančiai arba nedidėjančiai). Pateikiame keletą pavyzdžių.

Pradiniai duomenys			Rezultatas
33	55	22	33
22	33	22	22
22	33	33	33
55	55	55	55

3.5. Lygčių sprendimai su pradinių duomenų tyrimu

Turėdami sąlyginį sakinį galime ištirti pradinius duomenis. Tai ypač patogu rašyti lygčių sprendimo programas, kai pirmiausia reikia ištirti kiek ir kokių sprendinių turi lygtis.

Kvadratinė lygtis

$ax^2 + bx + c = 0.$

Šaknys randamos pagal formulę

$$x = \sqrt{\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}}$$

Reiškinys

$D = b^2 - 4ac$

vadinamas kvadratinės lygties diskriminantu. Kai $D < 0$, tai lygtis neturi šaknų, kai $D = 0$, tai lygtis turi vieną šaknį, kai $D > 0$, tai lygtis turi dvi šaknis. Taigi, programoje reikia pirmiausiai apskaičiuoti diskriminantą, nustatyti, kiek lygtis turi šaknų ir rasti jas.

```
program kvlygtis;
  const paklaida = 0.00001;
  var  a, b, c,          { lygties koeficientai }
       D,                { diskriminantas }
       x1, x2: real;     { šaknys }
begin
  read(a, b, c);
  D := b*b - 4*a*c;
  if abs(d) <= paklaida
    then writeln('x = ', -b / (2*a): 10: 2)
    else if D > paklaida
           then begin
                  writeln('x1 = ', (-b + D) / (2*a));
                  writeln('x2 = ', (-b - D) / (2*a))
                end
           else writeln('Lygtis sprendinių neturi')
end.
```

Programoje tikrinama, ar diskriminantas `D` nedidesnis už labai mažą dydį `paklaida`, vietoj matematikoje įprasto patikrinimo, ar jis lygus nuliui. Mat matematikoje laikoma, kad visi dydžiai yra absoliučiai tikslūs. Tuo tarpu kompiuteryje realieji skaičiai vaizduojami apytiksliai. Dėl paklaidų vietoj nulio gali atsirasti labai mažas skaičius, artimas nuliui, bet nelygus jam. Taip atsitinka ne visada, bet gera programa turi nepriekaištingai veikti su visais pradiniais duomenimis. Dėl to panaudojome konstantą `paklaida`.

Praktikos darbai

3.5.1. Dviejų tiesinių lygčių sistema

$$\begin{cases} a_1x + b_1y = c_1 \\ a_2x + b_2y = c_2 \end{cases}$$

Rankiniu būdu tokią sistemą sprendžiame taikydami įvairius metodus: keitimo, sudėties, įvesdami naujus kintamuosius ir pan. Pasirenkame patogesnį (reikalaujantį mažiau veiksmų) metodą konkrečiai lygčių sistemai spręsti priklausomai nuo jos koeficientų. Programa turėtų būti universali

ir tikti bet kokiai lygčių sistemai spręsti. Veiksmų kiekis kompiuteriui mažiau svarbus, negu žmogui. Todėl reikėtų pasirinkti kurį nors vieną metodą, tinkamesnį programavimui. Toks metodas yra. Dviejų tiesinių lygčių sistemos šaknį galima rasti pagal formules.

Pirmiausiai apskaičiuojami lygčių sistemos determinantai (nepainiokime su kvadratinės lygties diskriminantu)

$D = a_1b_2 - a_2b_1$,

$D_x = c_1b_2 - c_2b_1$,

$D_y = a_1c_2 - a_2c_1$.

Jeigu $D = 0$ ir kuris nors iš D_x arba D_y nelygus nuliui, tai lygčių sistema nesuderinama ir sprendinių neturi.

Jeigu $D = D_x = D_y = 0$, tai lygtys priklausomos ir jų sistema turi be galo daug sprendinių.

Jeigu netenkinama nė viena iš minėtų sąlygų, tai lygties sprendiniai egzistuoja ir juos galima rasti pagal šias formules:

$x = D_x / D$;

$y = D_y / D$

Šitaip galima išspręsti ne tik dviejų lygčių sistemą su dviem nežinomaisiais, bet *n* tiesinių lygčių sistemą su *n* nežinomųjų.

Parašykite programą dviejų tiesinių lygčių sistemai spręsti.

3.5.2. Bikvadratinė lygtis

Bikvadratinė lygtis

$ax^4 + bx^2 + c = 0$

sprendžiama įvedant pagalbinį kintamąjį

$y = x^2$

Tada sprendžiama kvadratinė lygtis

$ay^2 + by + c = 0$

o iš jos sprendinių gaunami bikvadratinės lygties sprendiniai.

Parašykite programą bikvadratinei lygčiai spręsti.

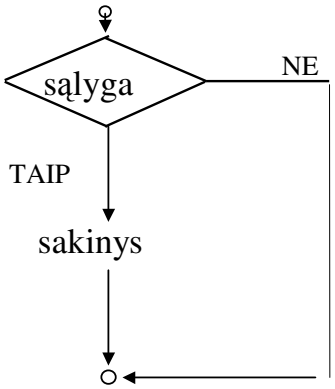
3.6. Veiksmų kartojimas. Ciklai `while` *ir* `repeat`

Programose būna veiksmų, kuriuos reikia kartoti. Dažniausiai veiksmai yra tie patys, tik juos kartoti reikia vis su kitais duomenimis.

Kartojimo veiksmai užrašomi ciklais. Paskalis turi trijų rūšių ciklus. Jų antraštės prasideda baziniais žodžiais **`while`**, **`for`** ir **`repeat`.** Todėl programuotojai kartais ciklus taip ir vadina šiais baziniais žodžiais.

Ciklas `while`. Kartojimo veiksmus taikliausiai aprašo **`while`** ciklas. Tai ir pats paprasčiausias ciklas. Jo veikimas grafiškai pavaizduotas 23 paveiksle, o programoje užrašomas šitaip:

`while` *loginis reiškinys* **`do`**
  sakinys



23 pav. Ciklo **while** schema

Pirmoji eilutė – ciklo antraštė. Po jos (t.y. po žodžio **do**) einantys sakinys kartojamas tol, kol tenkinama ciklo antaštėje užrašyta sąlyga (kol loginio reiškinio reikšmė yra `true`).

1 pavyzdys. Programa pirmajam reikšmingam (t.y. nelygiam nuliui) skaičiaus skaitmeniui rasti.

```
program pirmasis;
  var sk: integer;
begin
  read(sk);
  while sk > 9 do
    sk := sk div 10;
  writeln(sk)
end.
```

Šis uždavinys toks akivaizdus žmogui, kad jo netgi nelaikome uždaviniu – iš karto matome, koks pirmasis skaičiaus skaitmuo. Tuo tarpu kompiuteryje skaičius būna užrašytas dvejetainiais skaitmenimis ir kompiuteris dešimtainių skaičiaus skaitmenų „nemato“, lygiai taip, kaip mes nematome dvejetainių jo skaitmenų. Ir dar svarbiau – užduotis kompiuteriui gali būti išreiškiama tik tomis operacijomis, kurios yra programavimo kalboje. O Paskalis turi tik aritmetines skaičių operacijas.

Paseksime, kaip šis ciklas atliekamas su įvairiais pradiniais duomenimis.

Pradinis duomuo: 625.

	sk	Sąlyga
`read(sk)`	625	
`sk > 9`		`true`
`sk := sk div 10`	62	
`sk > 9`		`true`
`sk := sk div 10`	6	
`sk > 9`		`false`
`writeln(sk)`		

Ciklas buvo atliktas du kartus.

Jeigu pradinis duomuo būtų dviženklis skaičius, ciklas būtų atliktas vieną kartą. Jeigu pradinis duomuo būtų vienženklis skaičius, ciklas nebūtų atliktas nė vieno karto – rezultatas būtų lygus pradiniam duomeniui.

Prieš kiekvieną sakinio kartojimą, perskaičiuojama loginio reiškinio reikšmė ir pagal ją nustatoma, ar dar reikia kartoti sakinį, ar jau užbaigti ciklą. Jeigu sąlyga netenkinama (reiškinio

reikšmė `false`) tikrinant ją pirmą kartą, tai po žodžio **`do`** einantis sakinys neatliekamas nė karto. Taigi galima tvirtinti, kad ciklas **`while`** gali būti atliekamas 0, 1, 2 ir daugiau kartų.

Ciklas valdo vieno sakinio kartojimą. Jeigu reikia kartoti kelis sakinius, tai tie sakiniai sujungiami į vieną sudėtinį sakinį.

2 pavyzdys. Programa skaičiaus skaitmenų sumai rasti.

```
program SkSuma;
  var sk, suma: integer;
begin
  suma := 0;
  read(sk);
  while sk > 0 do
    begin
      suma := suma + sk mod 10;
      sk := sk div 10
    end;
  writeln(suma)
end.
```

3 pavyzdys. Programa klasės mokinių pažymių vidurkiui rasti.

```
program pvidurkis;
  var msk,                    { mokinių skaičius }
      p,                      { vieno mokinio pažymys }
      psuma: integer;         { visų mokinių pažymių suma }
      pbyla: text;            { pažymių byla }
begin
  assign(pbyla, 'PAŽYMIAI.TXT'); reset(pbyla);
  msk := 0;
  psuma := 0;
  read(pbyla, p);
  while p > 0 do
    begin
      psuma := psuma + p;
      msk := msk + 1;
      read(pbyla, p)
    end;
  writeln(psuma/msk: 8: 2)
end.
```

Paeiliui skaitomi ir sudedami mokinių pažymiai. Kartu skaičiuojamas ir mokinių (pažymių) skaičius `msk`. Ciklas kartojamas tol, kol skaitomi pažymiai didesni už nulį. Todėl čia nulis yra sutartinis ženklas. Juo užbaigiamas pažymių sąrašas. Kai perskaitomas nulis, kompiuteris supranta, kad jau perskaityti visi pradiniai duomenys (visų mokinių pažymiai), ciklą reikia baigti ir jau galima skaičiuoti vidurkį. Nulis – tai tik vienas iš daugelio būdų nurodyti sąrašo pabaigai. Jis tinka tada, kai sąraše nėra nulių. Šiam tikslui mes jį galėjome panaudoti tik todėl, kad laikėme, kad pažymių, lygių nuliui, nebūna.

Ar kiekvienas ciklas turi pabaigą? Tam, kad ciklas baigtųsi, reikia, kad jo antraštėje užrašyto loginio reiškinio reikšmė kada nors taptų `false`. Vadinasi, loginio reiškinio komponentus turi keisti sakinys. Priešingu atveju loginio reiškinio reikšmė visą laiką išliks `true` ir ciklas niekada nesibaigs. Toks ciklas vadinamas *amžinuoju*.

4 pavyzdys. Amžinasis ciklas kai nekeičiami loginio reiškinio komponentai.

```
while a < b do
  write(a)
```

5 pavyzdys. Amžinasis ciklas, kai ne ta linkme keičiami loginio reiškinio komponentai.

```
while a < b do
```

```
b := b + 1
```

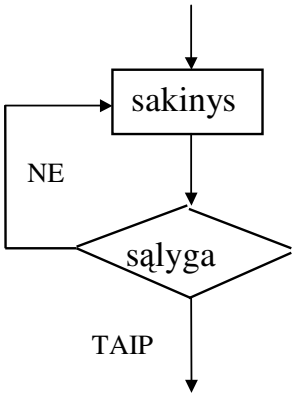
Amžinasis ciklas yra dažnai pasitaikanti klaida. Jeigu programa įtartinai ilgai veikia, tai galima tikėtis, kad kompiuteris atlieka amžinąjį ciklą. Tokiu atveju programos darbą reikia priverstinai nutraukti (daugelyje kompiuterių – paspausti klavišus Ctrl+Break) ir ieškoti klaidos – amžinojo ciklo.

Ciklas `repeat`. Kitas, rečiau vartojamas ciklas yra vadinamas **`repeat`** ciklu. Jo pavidalas yra šitoks:

`repeat`
sakinys$_1$*;*
sakinys$_2$*;*
...
sakinys$_n$
`until` *loginis reiškinys*

Sakiniai, esantys tarp žodžių **`repeat`** ir **`until`**, kartojami iki tol, kol bus patenkinta sąlyga – loginio reiškinio reikšmė taps `true`.

Ciklo veiksmai grafiškai parodyti 24 paveiksle.



24 pav. Ciklo **repeat** schema

6 pavyzdys. Programa, skaičiuojanti sumą skaičių sekos, kurios paskutinis skaičius sutampa su pirmuoju. Skaičiai renkami klaviatūra.

```
program suma;
  var sk,              { skaitomas skaičius }
      pirmas,          { pirmas perskaitytas skaičius }
      sum: integer;    { suma  }
begin
  read(pirmas);
  sum := pirmas;
  repeat
    read(sk);
    sum := sum+sk
  until sk = pirmas;
  writeln(sum)
end.
```

Cikle **repeat** pirma atliekami veiksmai, o po to tikrinama sąlyga. Dėl to ciklas visada atliekamas bent vieną kartą. Taigi galima tvirtinti, kad ciklas **repeat** gali būti atliekamas 1, 2, 3 ir daugiau kartų.

Ciklas **repeat** baigiamas, kai tenkinama jo pabaigoje (po žodžio **until**) parašyta sąlyga. Tuo tarpu ciklas **while** baigiamas, kai nebetenkinama jo pradžioje parašyta sąlyga. Šios dvi priešingybės kartais supainiojamos ir programuotojas suklysta. Galime pasiūlyti tokį lengviau įsimenamą sąlygų tikrinimo modelį. Vietoj sąlygų įsivaizduokime šviesoforus. Kai sąlyga tenkinama, šviesoforas žalias, kai netenkinama – raudonas.

Šviesoforas stovi ciklo **while** pradžioje. Vadinasi, patekti į ciklą galima kai šviesoforas žalias – sąlyga tenkinama.

Šviesoforas stovi ciklo **repeat** pabaigoje. Vadinasi, jis reguliuoja išvažiavimą iš ciklo. Taigi ciklą palikti galima kai šviesoforas žalias – sąlyga tenkinama.

Žodis **until** atlieka ir skyrybos ženklo vaidmenį – parodo, kur baigiasi ciklo sakiniai. Dėl to cikle **repeat** galima rašyti daugelį sakinių – nereikia jų „apskliausti“.

Uždaviniai

3.6.1. Kokia bus kintamojo `x` reikšmė, atlikus šitokias sakinių sekas?

```
a) k := 1;
   while k < 5 do
     k := k + 1;
   x := k
b) x := 1;
   while x <= 5 do
     x := x + 1
```

3.6.2. Duota sakinių seka:

```
a := 1; b := 1;
while a+b < 8 do
  begin
    a := a+1;
    b := b+2
  end;
s := a+b
```

Kiek kartų bus atliekamas ciklas ir kokios bus kintamųjų *a*, *b* ir *s* reikšmės, atlikus duotąją sakinių seką?

3.6.3. Kokios bus kintamųjų *a* ir *b* reikšmės, atlikus šią sakinių seką:

```
a := 1;  b := 1;
while a <= 3 do
  a := a+1; b := b+1
```

3.6.4. Parašykite programą, kuri rastų mažiausią skaičių byloje `SKAIČIAI.TEK`.

Praktikos darbas

3.6.1. Rezultatų kaita atliekant programą. 3 pavyzdžio programą `pvidurkis` modifikuokite taip, kad ji apskaičiuotų per pamoką gautų mokinių pažymių vidurkį dinamiškai, t.y. pažymių sąrašas papildomas, kai tik pažymį gauna nauja mokinys ir kiekvieną kartą ekrane parodomas naujas, pakoreguotas vidurkis. Atlikite ją kompiuteriu.

3.7. Valdymo struktūrų palyginimas

Prieskyros, duomenų skaitymo bei rašymo ir tuščias sakinys yra elementarūs, nedalomi. Į juos negali įeiti jokie kiti sakiniai.

Sakinių atlikimo tvarką nustato struktūriniai sakiniai – sudėtinis, sąlyginis ir ciklo. Jie sudaromi iš elementarių ir trumpesnių struktūrinių sakinių.

Kelių sakinių seką sudėtinis sakinys paverčia vienu sakiniu. Todėl visur ten, kur pagal programavimo kalbos taisykles gali būti rašomas tik vienas sakinys, o norisi rašyti kelis sakinius, tuos kelis sakinius reikia sujungti į vieną sudėtinį, ir problema bus išspręsta.

Visą programą sudaranti sakinių seka taip pat jungiama į vieną sudėtinį sakinį. Taigi galima tvirtinti, kad visos programos veiksmai išreiškiami vienu sudėtiniu sakiniu. Žinoma, tas sakinys gali būti sudėtingas ir ilgas – užimti daug puslapių.

Sąlyginis sakinys išrenka vieną kurią nors programos šaką. Kitos šakos sakiniai neatliekami. Vadinasi, kompiuteris atlieka mažiau sakinių negu jų yra programoje.

Ciklų sakiniai nurodo, kad reikia ciklo sakinius kartoti daug kartų. Vadinasi, kai programoje yra ciklų, kompiuteriui gali tekti atlikti daugiau veiksmų negu jų yra parašyta programoje.

Struktūriniai sakiniai dar vadinami valdymo struktūromis, nes jais išreiškiami nurodymai, kaip valdyti kitų sakinių atlikimo tvarką. Visos nagrinėtos valdymo struktūros yra lygiateisės: jos gali įeiti viena į kitą. Nuosekli sakinių seka jau buvo pakeista vienu sudėtiniu sakiniu, įeinančiu į ciklą arba sąlyginį sakinį. Patys sąlyginiai sakiniai arba ciklai buvo eiliniai nuoseklios sakinių sekos nariai. Ciklas taip pat gali būti kitame cikle arba sąlyginiame sakinyje be jokių ribojimų.

1966 m. C. Bohmas ir G. Jacopinis įrodė, kad **kiekvieną elementarių sakinių atlikimo tvarką galima aprašyti trimis valdymo struktūromis: sakinių seka, sąlyginiu sakiniu ir ciklu.** Pakanka tik vienos rūšies (nesuprastinto) sąlyginio sakinio ir vienos rūšies (while) ciklo. Kiti (alternatyvūs) struktūriniai sakiniai nėra būtini. Tačiau jie sutrumpina programą arba padaro ją vaizdesnę. Prieš rašant struktūrinį sakinį reikia pagalvoti, kurią iš galimų alternatyvių valdymo struktūrų naudoti. Pasirinkus tinkamą struktūrą, programa bus trumpesnė, ją bus lengviau skaityti.

Uždaviniai

3.7.1. Kurie iš šių teiginių teisingi?

a) ciklas, prasidedantis žodžiu **`while`**, gali būti neatliktas nė vieno karto;

b) ciklas, prasidedantis žodžiu **`repeat`**, gali būti neatliktas nė vieno karto;

c) ciklas, prasidedantis žodžiu **`while`** arba **`repeat`**, gali būti atliktas vieną kartą.

3.7.2. Palikite vieną iš skliaustuose parašytų alternatyvų:

*Ciklo sakinys, prasidedantis žodžiu **`while`** valdo (vieno sakinio / sakinių sekos) kartojimą, o ciklo sakinys, prasidedantis žodžiu **`repeat`** valdo (vieno sakinio / sakinių sekos) kartojimą.*

3.8. Begalinių eilučių sumavimas

Daugelį matematikos konstantų arba funkcijų galima išreikšti begalinių eilučių suma. Pateiksime pavyzdžių:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots + \frac{1}{n!} \ldots$$

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} \ldots$$

(e – natūrinių logaritmų pagrindas)

$$\frac{\pi}{4} = 1 - \frac{1}{3!} + \frac{1}{5!} - \frac{1}{7!} + \ldots \frac{x^n}{n!} \ldots$$

$$\text{arctg}\, x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

Kuo daugiau eilutės narių sudėsime (kompiuteriui nesunku sudėti jų labai daug), tuo tikslesnę reikšmę gausime. Kiek jų reikia sudėti, programoje galima nurodyti įvairiai. Paprasčiausia sumuoti tam tikrą iš anksto nustatytą pirmųjų narių skaičių. Tačiau programos dažniausiai sudaromos taip, kad būtų sumuojami eilutės nariai tol, kol jie pasidaro mažesni už tam tikrą, labai mažą, iš anksto pasirinktą dydį. Šis dydis ir apibūdina paklaidą.

Pavyzdys. Sudarysime programą natūrinių logaritmų pagrindui *e* rasti.

```
program logaritmas;
  const epsilon = 1E-6;      { liekamasis narys }
  var k : integer;
      e, narys: real;
begin
  e := 0.0;
  narys := 1.0;
  k := 1;
  while narys >= epsilon do
    begin
      e := e + narys;
      narys := narys/k;      { 1/1 }
                             {1/1/2 = 1/(1*2)}
                             {1/1/2/3 = 1/(1*2*3)}
                             { . . . }
      k := k + 1
    end;
  writeln(e)
end.
```

Šioje programoje skaičiaus faktorialas neskaičiuojamas išreikštiniu būdu. Eilinis eilutės narys gaunamas ankstesnį jos narį padalijus iš vieno po kito einančių natūraliųjų skaičių. Taip išvengiama didelių skaičių faktorialų reikšmių, kurios gali netilpti į sveikųjų skaičių rėžius.

Kompiuteris, atlikęs programą `logaritmas`, pateikė šitokį rezultatą:

```
2.7182815256E+00
```

Palyginimui pateikiame tikslesnę skaičiaus *e* reikšmę:

2,7182818284.

Programose, kurių rezultatai yra sveikieji skaičiai, reikia vengti realiųjų skaičių, nes operacijų su jais rezultatai yra apytiksliai skaičiai ir gali iškreipti tikrąjį rezultatą. Pavyzdžiui, jei skaičiuodami sklypo plotą vietoj 8191 m^2 gauname 8191,5 m^2 arba 8191 m^2, tai į tokias menkas paklaidas galime nekreipti dėmesio. Tuo tarpu jeigu pirminių skaičių ieškojimo programoje vietoj 8191 gautume 8192, tai toks rezultatas ir jį duodanti programa būtų klaidingi.

Uždaviniai

3.8.1. Parašykite programą funkcijos e^x reikšmei rasti, taikydami skyrelio pradžioje pateiktą eilutės sumavimo formulę.

3.8.2. Parašykite programą skaičiaus π reikšmei rasti, taikydami skyrelio pradžioje pateiktą eilutės sumavimo formulę. Sumuoti reikia tol, kol eilutės nario modulis bus mažesnis už 10^{-10}.

4. DISKRETIEJI DUOMENŲ TIPAI IR JŲ VALDOMI VEIKSMAI

Gyvenime susiduriame su dviejų tipų dydžiais: tolydžiaisiais ir diskrečiaisiais. Kompiuterio prigimtis yra diskreti arba dar vadinama kitaip – skaitmeninė. Kiekvieno jo elemento būseną galima apibūdinti diskrečiuoju dydžiu, pavyzdžiui, sveikuoju skaičiumi. Skaičiavimo technika lemia ir tai, kad gyvenime vis daugiau plinta diskretieji (skaitmeniniai) prietaisai: skaitmeninės svarstyklės, skaitmeniniai garso įrašai, skaitmeninė fotografija ir t.t. Dėl to natūralu, kad daugelio uždavinių pradiniai duomenys ir rezultatai yra diskretieji duomenys.

Programavime diskretieji duomenys naudojami dar ir veiksmams valdyti, panašiai, kaip ir loginiai duomenys.

4.1. Vardiniai duomenų tipai – diskrečiųjų duomenų tipų pagrindas

Uždaviniuose dažnai būna duomenų, kurie gali turėti nedaug galimų reikšmių. Pavyzdžiui, uždavinyje, kuriame operuojama su Mėnulio fazėmis, reikalingas duomuo, galintis turėti keturias reikšmes: *jaunatis, priešpilnis, pilnatis, delčia.* Fazes iš bėdos galima žymėti (koduoti) skaičiais (pavyzdžiui, 1, 2, 3, 4). Tačiau tai nepatogu ir nevaizdu – reikia prisiminti, kuris skaičius kurią fazę reiškia. Todėl programoje fazes būtų geriau vadinti tikraisiais jų vardais, taip, kaip ir gyvenime.

Panašių duomenų pasitaiko dažnai. Pavyzdžiui, pagrindinė spalva – *mėlyna, geltona, raudona;* mokyklos tipas – *pradinė, pagrindinė, vidurinė, gimnazija*; pasaulio šalis – *šiaurė, rytai, pietūs, vakarai;* trumpas atsakymas anketoje – *taip, ne, nežinau;* septynios savaitės dienos ir t.t. Būtų patogu, kad programavimo kalboje kiekvienam aptartam atvejui rastume tinkamą duomenų rūšį, programavime vadinamą duomenų tipu. Tačiau, kad ir kiek būtų duomenų tipų, vis tiek jų neužtektų, nes įvairių uždavinių programose beveik visada yra tik tiems uždaviniams būdingų duomenų. Todėl užuot gausinus tipų skaičių, einama kitu keliu – leidžiama pačiam programuotojui aprašyti naujus jam reikalingus tipus.

Aprašysime paminėtus duomenų tipus.

```
type fazė = (jaunatis, priešpilnis, pilnatis, delčia);
     spalva = (mėlyna, geltona, raudona);
     mokykla = (pradinė, pagrindinė, vidurinė, gimnazija);
     pasšalis = (šiaurė, rytai, pietūs, vakarai);
     atsakymas = (taip, ne, nežinau);
     savd = (pirm, antr, treč, ketv, penk, šešt, sekm);
```

Visų tipų aprašas pradedamos žodžiu `type`, o kiekvieno tipo aprašas – naujai aprašomo duomenų tipo vardu.

Skliaustuose išvardijamos aprašomiems duomenų tipams priklausančių reikšmių (konstantų) vardai. Todėl tokie tipai vadinami *vardiniais*.

Aprašysime keletą kintamųjų, vartodami ką tik aprašytus vardinius duomenų tipus.

```
var pieštukas, rašalas: spalva;
    rodyklė, kryptis: pasšalis;
    a, b, c: mokykla;
    šiandien, rytoj: savd;
```

Vardinis tipas iš tikrųjų yra ne vienas duomenų tipas, o daug skirtingų tipų – tiek, kiek jų aprašyta programoje. (Čia aprašėme net šešis tipus). Tai vardinių duomenų tipų šeima.

Programuotojo aprašyti duomenų tipų vardai (pvz., `pasšalis, mokykla, savd`) panaudojami analogiškai, kaip ir integruoti (pvz., `integer, boolean`).

Kintamajam galima priskirti tik jo apraše nurodyto tipo reikšmes, pavyzdžiui:

```
pieštukas := geltona;
rašalas   := mėlyna;
rašalas   := pieštukas;
rodyklė   := pietūs;
a         := treč
```

Tačiau neleistina priskirti kito tipo reikšmes:

```
rodyklė := raudona;
a := pieštukas
```

Daugeliui duomenų tipų būdinga jų reikšmių tvarka. Surikiuotos yra, pavyzdžiui, savaitės dienos: trečiadienis eina po antradienio ir prieš ketvirtadienį. Dėl to vardiniai duomenų tipai laikomi sutvarkytais: laikoma, kad jų reikšmės surikiuotos tokia tvarka, kaip jų vardai pateikti duomenų tipo apraše. Todėl visų vardinių duomenų tipų reikšmes galima palyginti (atlikti operacijas <, <=, <>, > ir >=). Be to, galima atlikti dvi integruotas funkcijas `succ` ir `pred`.

Funkcijos `succ` reikšmė yra to paties vardinio tipo reikšmė, einanti po duotosios reikšmės, pavyzdžiui:

```
succ(ne) = nežinau,
succ(šeš) = sekm
```

Funkcijos `pred` reikšmė yra to paties tipo reikšmė, einanti prieš duotąją reikšmę, pavyzdžiui:

```
pred(rytai) = šiaurė,
pred(šešt) = penk
```

Atliekant lyginimo operacijas, laikoma, kad reikšmių eilė sutampa su jų konstantų vardų eile tipo apraše. Pavyzdžiui:

```
pirm < antr → true,
pirm <= antr → true,
pirm > antr → false,
šiaurė < rytai → true,
priešpilnis < pilnatis → true.
```

Vardinių duomenų tipų reikšmių Paskalio kalboje negalima nei skaityti, nei rašyti. Tai galima laikyti Paskalio kalbos arba vardinių duomenų tipų trūkumu. Tačiau šis trūkumas nėra programavimo kalbų projektuotojų klaida. Tai savybė, išplaukianti iš pačių vardinių duomenų tipų aprašymo būdo. Jie aprašomi programos viduje. O viskas, kas aprašyta programoje, nėra žinoma už jos ribų, t.y. prieš pradedant programai darbą arba jį baigus. Ta pati reikšmė vienoje programoje gali būti pavadinta vienu vardu, kitoje – kitu. Kas kita yra visuotinai priimtus žymenis turinčios integruotų duomenų tipų, pavyzdžiui, skaičių, reikšmės. Skaičiai visose programose žymimi vienodai. Dėl to vardiniai duomenų tipai dažniau vartojami didesnėse programose, kuriose apsimoka turėti daug įvairių vidinių duomenų tipų.

Diskretieji duomenų tipai ir vardiniai duomenų tipai. Programavime duomenų tipai klasifikuojami įvairiai. Viena iš galimų klasifikacijų: *diskretieji* ir *tolydieji*.

Diskretieji duomenų tipai yra tokie, kurie turi baigtinį reikšmių skaičių ir tas reikšmes galima surikiuoti. Su jais atliekamos tik tos operacijos, kurias nusako reikšmių rikiavimas: lyginimo operacijos ir funkcijos `succ` ir `pred`. Ir viskas.

Paskalio kalboje vardinis duomenų tipas laikomas visų kitų diskrečiųjų duomenų tipų pagrindu. Visi kiti diskretieji duomenų tipai kildinami iš vardinio duomenų tipo. Laikoma, kad jie visi iš vardinio duomenų tipo paveldi lyginimo operacijos ir funkcijos `succ` ir `pred` ir turi dar naujų savų operacijų

Diskrečiųjų duomenų tipų grupei priklauso sveikieji skaičiai, loginis bei simbolinis duomenų tipas. Loginio duomenų tipo reikšmės žymimos vardais `false` ir `true`, visai taip kaip ir vardinio duomenų tipo reikšmės. Laikoma, kad reikšmės surikiuotos tokia pat tvarka kaip ir ką tik jas išvardinome: `false`, `true`.

Sveikieji skaičiai turi savus konstantų žymenis. Galima laikyti, kad skaičiai (pvz., -123, 4545) yra tikriniai, t.y. neaprašomi sveikojo duomenų tipo konstantų vardai.

Simbolinio tipo reikšmės – simboliai taip pat yra šio duomenų tipo konstantų tikriniai vardai.

Tolydžiųjų duomenų tipų grupei priklauso tik realieji skaičiai.

Programavime diskretieji duomenų tipai vaidina svarbų vaidmenį dėl to, kad jų reikšmės gali valdyti veiksmų atlikimo tvarką. Apie tai kalbėsime 4.3 ir 4.4 skyreliuose.

Uždaviniai

4.1.1. Duoti šie vardinių duomenų tipų aprašai:

```
type pirštas = (nykštys, smilius, didysis, bevardis, mažylis);
     planeta = (Merkurijus, Venera, Žemė, Marsas, Jupiteris,
                Saturnas, Uranas, Neptūnas, Plutonas);
     Zodiakas = (Vandenis, Žuvis, Avinas, Tauras, Dvyniai,
                 Vėžys, Liūtas, Mergelė, Svarstyklės,
                 Skorpionas, Šaulys, Ožiaragis);
```

Nurodykite, kurie reiškiniai netaisyklingi, ir apskaičiuokite taisyklingų reiškinių reikšmes:

a) `succ(succ(smilius));`

b) `pred(pred(Venera));`

c) `succ(pred(pred(Venera)));`

d) `Tauras = Avinas;`

e) `Jupiteris > Tauras;`

f) `pred(Mergelė);`

g) `(Saturnas > Neptūnas) or (Žuvis > Vėžys);`

h) `planeta = Žemė.`

4.1.2. Duota programa:

```
program šeima;
  type asmuo = (aš, tu, jis, ji);
       giminė = (tėvas, motina, sūnus, duktė, dėdė, teta);
  var a, b: asmuo;
      x: giminė;
begin
  { 1 } a := aš;
  { 2 } b := a;
  { 3 } a := asmuo;
  { 4 } giminė := x;
  { 5 } if ji = asmuo then write('TAIP');
  { 6 } if dėdė <> teta then write('GERAI');
  { 7 } if ji = teta then write('DĖDIENĖ');
  { 8 } if jis = ji then write('NEAIŠKU');
  { 9 } x := tėvas;
  { 10} if pred(x) = tėvas then write('XXX')
end.
```

Kurie sakiniai yra netaisyklingi ir kodėl?

4.1.3. Parašykite sąlyginį sakinį, kuris patikrintų, ar iš keturių atkarpų, kurių ilgiai duoti, galima sudaryti kvadratą arba stačiakampį. Rezultato duomenų tipas aprašytas taip:

```
var forma: (kv, st, ne);
      { kv – kvadratas (tuo pačiu ir stačiakampis) }
      { st – stačiakampis bet ne kvadratas }
      { ne – ne stačiakampis }
```

4.1.4. Parašykite sąlyginį sakinį, kuris nustatytų, ar dvi duotos savaitės dienos *a* ir *b* yra gretimos. Pavyzdžiui,

šeštadienis ir *sekmadienis* yra gretimos,

šeštadienis ir *penktadienis* yra gretimos,
šeštadienis ir *ketvirtadienis* nėra gretimos.

4.2. Atkarpos tipai

Dalį iš eilės einančių tipo reikšmių galima pavadinti nauju tipu – atkarpos tipu.

```
type darbodiena = pirm..penk;
     ma = 6..18; { moksleivio amžius }
```

Duomenų tipas, iš kurio imamos reikšmės atkarpos tipui, vadinamas *baziniu duomenų tipu*. Ką tik aprašyto `darbodiena` tipo reikšmės yra 4.1 skyrelyje aprašyto `savd` tipo poaibis. Tipui `darbodiena` priklauso pirmosios penkios tipo `savd` reikšmės. Tipas `savd` yra tipo `darbodiena` bazinis tipas. Atkarpos tipo apraše nurodomos pirma (ji dar vadinama *apatiniu rėžiu*) ir paskutinė tipo reikšmė (dar vadinama *viršutiniu rėžiu*). Tipo `darbodiena` pirmoji reikšmė yra `pirm`, paskutinė – `penk`.

Tipo `ma` reikšmės yra sveikųjų skaičių poaibis.

Su atkarpos tipo reikšmėmis atliekamos tos pačios operacijos, kaip ir su jo bazinio tipo reikšmėmis, t.y. atkarpos tipas neturi nei savų reikšmių, nei savų operacijų. Tuo jis skiriasi nuo kitų naujų tipų, pavyzdžiui, vardinių.

Taigi atkarpos tipus logiškiau būtų laikyti ne naujais, o išvestiniais tipais, gautais iš bazinio. Naujesnėse programavimo kalbose taip ir daroma. Pavyzdžiui, kalboje Ada atkarpos tipai vadinami bazinių tipų potipiais. Paskalio kalboje potipio sąvokos nėra, todėl atkarpos tipai laikomi lyg ir naujais tipais – jie turi savus vardus, savus kintamuosius. Tačiau atkarpos tipo konstantų ir atkarpos tipo reikšmių nėra. Kai reiškinyje yra atkarpos tipo kintamųjų, tai laikoma, kad tokie kintamieji reiškiniui pateikia bazinio tipo reikšmes. Pavyzdžiui, jeigu yra aprašytas tipas ir kintamieji

```
type pažymys = 1..5;
var a, b: pažymys;
    i: integer;
```

tai taisyklingi tokie sakiniai

```
a: = 1;
b := 5;
i := a + b;
a := (b*i + a) div 3;
b := i div b.
```

Tačiau neleistini (netaisyklingi) yra sakiniai, kuriuose bandoma atkarpos tipo kintamajam priskirti reikšmę, nepatenkančią į nurodytus rėžius, pavyzdžiui:

```
a := 6;
b := 0;
a := b*3.
```

Atkarpos tipai, panašiai kaip ir vardiniai, sudaro tipų šeimą. Programuotojas gali aprašyti kiek nori savų atkarpos tipų.

Atkarpos tipo aprašo dešinėje pusėje gali būti tik konstantos, nors jos būtų pažymėtos ir vardais, pavyzdžiui:

```
const senas  = 1901;
      jaunas = 1987;
type gimimomet = senas..jaunas
```

Atkarpos tipo reikšmės gali būti tik viename, vientisame intervale. Kitaip aprašyti atkarpos tipo reikšmes (pavyzdžiui, nelygybėmis) nepriimtina, nes tada daug sunkiau būtų kontroliuoti tipus kompiuteryje.

Atkarpos tipai iš esmės naujų galimybių neduoda, tačiau su jais programa dažnai tampa aiškesnė – jau iš tipo aprašo matyti kintamųjų reikšmių rėžiai. Be to, kompiuteris gali patikrinti, ar atliekant programą

gautos kintamųjų reikšmės iš tikrųjų yra leistiname intervale. Tais atvejais, kai intervalas nedidelis, reikšmei saugoti gali būti skiriama mažiau atmintinės.

Pakalbėkime, kaip atkarpos tipą galima panaudoti pradiniams duomenims patikrinti.

Daugelio programų rezultatai turi prasmę tik tada, kai pradiniai duomenys yra tam tikrame reikiamo tipo reikšmių intervale. Todėl nepriekaištingai sudarytose programose pirmiausia patikrinama, ar geri pradiniai duomenys, o funkcijų ir procedūrų – ar tinkamos parametrų, kuriais perduodami pradiniai duomenys, reikšmės. Vartojant atkarpos tipus, tokių tikrinimų dažnai galima išvengti. Pavyzdžiui, programoje, nustatančioje, koks bilieto numeris – laimingas ar ne – galime netikrinti, ar pradinis duomuo tikrai yra teigiamas šešiaženklis skaičius, jeigu aprašysime atkarpos tipą

```
type numeris = 100000..999999
```

ir pradinį duomenį priskirsime šio tipo kintamajam:

```
program laimingas;
  type numeris = 100000..999999;
  var x: numeris;
begin
  read(x);
  ...
```

Tada kompiuteris, atlikdamas sakinį `read(x)`, pats patikrins, ar pradinis duomuo yra nurodytame intervale. Jeigu jis nepateks į atkarpos intervalą, tai kompiuteris praneš apie klaidą. Koks bus pranešimas, kaip kompiuteris reaguos į klaidą (nutrauks skaičiavimą, klaidą taisys ar ignoruos), priklausys nuo konkretaus kompiuterio ir programavimo kalbos. Jeigu norime, kad kompiuteris atliktų reikiamus veiksmus, pavyzdžiui, rašytų pranešimą

```
PER DIDELIS SKAIČIUS
```

jeigu $x > 999999$, arba pranešimą

```
PER MAŽAS SKAIČIUS
```

jeigu $x < 100000$, tai tokiu atveju atkarpos tipas negelbės, nes visus veiksmus reikia užrašyti programoje. Jis taip pat negelbės, jeigu pradinio duomens ribojimo negalima išreikšti intervalu. Pavyzdžiui, atkarpos tipu negalima išreikšti reikalavimo, kad pradinis duomuo būtų lyginis skaičius ir pan.

Matematikoje sveikųjų skaičių aibė begalinė. Programavime duomenų tipas `integer` yra matematinių skaičių poaibis, ribojamas mažiausio ir didžiausio skaičiaus (`maxint`). Todėl duomenų tipą `integer` galima laikyti matematinių sveikųjų skaičių atkarpos tipu.

Uždaviniai

4.2.1. Programoje yra tokie kintamųjų aprašai:

```
var a:10..30;
    b: 0..20;
    c: 0..10;
```

Kurie iš išvardytų prieskyros sakinių dėl atkarpos tipų rėžių suderinamumo yra: a) visada teisingi; b) visada neteisingi; c) ne visada teisingi (priklauso nuo reiškinio kintamųjų reikšmių):

1) `b := a;`
2) `c := a;`
3) `b := c;`
4) `b := (c + 30) div 10;`
5) `c := a + 1;`
6) `a := -b.`

4.2.2. Duota programa:

```
program x;
  var a: -10..10;
      b: 1..5;
      x1, x2, x3, x4, x5, x6, x7: integer;
```

```
begin
  read(a, b);
  x1 := a + b;
  x2 := a - b;
  x3 := -a;
  x4 := -b;
  x5 := a*b;
  x6 := x5 + 1;
  x7 := a*3 + b*2;
  x6 := x1 + x6;
  write(x1, x2, x3, x4, x5, x6, x7)
end.
```

Kintamuosius `x1` – `x7` aprašykite tokiais sveikųjų skaičių atkarpos tipais, kad į jos rėžius visada tilptų rezultatas, o reikšmių skaičius būtų mažiausias.

4.3. Variantinis sakinys

Paskalis turi sakinį, specialiai pritaikytą vienam veiksmui iš daugelio veiksmų parinkti. Šis sakinys vadinamas *variantiniu*. Juo parenkamas vienas iš daugelio sakinių priklausomai nuo reiškinio reikšmės. Sakinio pavidalas yra šitoks:

case *reiškinys* **of**
c1: S1;
c2: S2;
...
cn: Sn
end

c1, c2, ..., cn – konstantų sąrašai;

S1, S2, ..., Sn – sakiniai.

Išrenkamas ir vykdomas vienas iš sakinių *Si* – tas prieš kurį parašytame konstantų sąraše *ci* yra konstanta, sutampanti su variantinio sakinio antraštėje esančio reiškinio reikšme.

1 pavyzdys. Perskaitomas vienženklis skaičius, o spausdinamas šis skaičius žodžiu.

```
program skaitmuo;
  var s: integer;
begin
  read(s);
  case s of
    0: write('nulis');      5: write('penki');
    1: write('vienas');     6: write('šeši');
    2: write('du');         7: write('septyni');
    3: write('trys');       8: write('aštuoni');
    4: write('keturi');     9: write('devyni')
  end;
  writeln
end.
```

Reiškinys ir konstantos turi būti to paties diskrečiojo duomenų tipo. Mums jau žinomi sveikieji skaičiai ir loginės reikšmės priklauso diskretiesiems duomenų tipams, todėl jie gali būti panaudoti variantiniame sakinyje. Tuo tarpu realieji skaičiai šiam tikslui netinka.

Turbo Paskalyje vietoj iš eilės einančių konstantų sąrašo galima rašyti konstantų atkarpą: pirmąją ir paskutinę konstantų reikšmes, atskirtas dviem taškais. Pavyzdžiui, vietoj

```
2, 3, 4, 5
```

galima rašyti

```
2..5
```

Be to, paskutinėje šakoje vietoj konstantų sąrašo ir po jo einančio dvitaškio gali būti žodis **else**. Sakinys, parašytas po **else**, atliekamas tuo atveju, kai reiškinio reikšmė nesutampa su jokia konstanta ir nepatenka į jokį konstantų intervalą.

2 pavyzdys. Tarkime, kad kintamasis `aa` yra sveikojo tipo.

```
case aa of
  0, 2, 4, 6, 8: write('lyginis vienženklis');
  1, 3, 5, 7, 9: write('nelyginis vienženklis');
  10..99       : write('dviženklis');
  else           write('neigiamas arba didesnis už 99')
end
```

Kiekvienoje varianto šakoje gali būti rašomas tik vienas sakinys. Tačiau jis gali būti bet koks: prieskyros, sąlyginis, variantinis, sudėtinis ir kt. Kai reikia atlikti kelis sakinius, tai jie apjungiami į vieną sudėtinį.

Šakojimąsi į daugelį krypčių galima išreikšti ir vienu variantiniu sakiniu ir daugeliu vienas į kitą įdėtų sąlyginių sakinių. Vieno variantinio sakinio atvejis paprastesnis, tačiau juo galima išreikšti tik tokį šakojimąsi, kurį valdo vieno reiškinio reikšmė. Kai šakojimasis priklauso nuo kitokių (sudėtingesnių) sąlygų, tenka naudoti sąlyginius sakinius.

Uždaviniai

4.3.1. Pradinis duomuo – mėnesio numeris. Parašykite programą, kuri spausdintų mėnesio pavadinimą.

4.3.2. Kintamasis `log` yra loginio tipo. Variantinį sakinį

```
case log of
  false: a := a+1;
  true : b := b+1
end
```

pakeiskite jam ekvivalenčiu, t.y. atliekančiu tokius pačius veiksmus, sąlyginiu sakiniu.

4.3.3. Pradiniai duomenys – du sveikieji skaičiai: *metai* ir *mėnuo*. Parašykite programą, kuri rastų, kiek dienų turi *mėnuo*.

Apie keliamųjų metų nustatymą žr. 3.1.12 uždavinį.

Praktikos darbas

4.3.1. Pinigų suma žodžiu. Sudarykite programą, kuri duotą pinigų sumą (0..9999) litais parašytų žodžiu, o pabaigoje – žodžio „litas“ reikiamą linksnį. Pavyzdžiui, jeigu pradinis duomuo 127, programa turi parašyti

```
vienas šimtas dvidešimt septyni litai
```

jeigu pradinis duomuo 5819, programa turi rašyti

```
penki tūkstančiai aštuoni šimtai devyniolika litų
```

4.4. Žinomo kartojimų skaičiaus ciklas

Kai kartojimų skaičius žinomas prieš atliekant ciklą, patogu vartoti šitokio pavidalo ciklą:

for *ciklo kintamasis* := *reiškinys*$_1$ **to** *reiškinys*$_2$ **do**
  sakinys

Ciklo antraštė prasideda žodžiu **for**. Po jo rašomas ciklo kintamojo vardas. Toliau nurodoma, su kokiomis ciklo kintamojo reikšmėmis turi būti atliekamas ciklas.

Sakinys atliekamas su skirtingomis ciklo kintamojo reikšmėmis. Pirmoji jo reikšmė yra *reiškinys*$_1$. Antroji – reikšmė, einanti po reiškinio *reiškinys*$_1$ reikšmės, t.y. *succ(reiškinys*$_1$*)*, trečioji – *succ(succ(reiškinys*$_1$*)* ir t.t. Paskutinį kartą sakinys atliekamas kai ciklo kintamojo reikšmė yra *reiškinys*$_2$.

1 pavyzdys. Sakinių seka skaičiui *a* pakelti laipsniu *n* (n ≥ 0).

```
p := 1;
for i := 1 to n do
  p := p*a
```

Ciklas atliekamas *n* kartų, t.y. tiek kartų, kokiu laipsniu keliamas skaičius *a*. Kiekvieną kartą atliekant ciklą, rezultatui *p* priskiriama vis nauja reikšmė, kuri lygi ankstesnei jo reikšmei, padaugintai iš kintamojo *a* reikšmės. Pavyzdžiui, kai *a* = 3 ir *n* = 2, ciklas atliekamas du kartus ir gaunama *p* = 9, t.y. rezultatas lygus skaičiaus 3 kvadratui. Kai *a* = 5 ir *n* = 3, ciklas atliekamas tris kartus ir gaunama *p* = 125 (skaičiaus 5 kubas).

Kai n = 0, ciklas neatliekamas nė karto. Tada rezultatas p = 1, kas atitinka kėlimo laipsniu operacijos apibrėžtį.

2 pavyzdys. Programa, pagal kurią randama nurodyto intervalo visų skaičių kvadratų suma, lyginių skaičių kvadratų suma ir nelyginių skaičių kvadratų suma atskirai.

```
program sumos;
  var m, n,       { intervalas }
      slyg,       { lyginių suma }
      snelyg,     { nelyginių suma }
      k: integer;
begin
  read(m, n);
  slyg := 0; snelyg := 0;
  for k := m to n do
    if k mod 2 = 0
       then slyg := slyg + k*k
       else snelyg := snelyg + k*k;
  writeln(slyg + snelyg: 10, slyg: 10, snelyg: 10)
end.
```

Šiame pavyzdyje ciklui priklauso tik vienas sakinys, parašytas po jo antraštės. Tai sąlyginis sakinys. Vadinasi, sąlyginio sakinio veiksmai bus kartojami tiek kartų, kiek kartų bus atliekamas ciklas. Rašymo sakinys ciklui nepriklauso. Todėl atliekamas tik vieną kartą – pasibaigus ciklui. Kai *m* = 2 ir *n* = 4, ciklas atliekamas tris kartus ir ekrane matysime šitokius rezultatus:

```
29   20   9
```

Kai reikia kartoti kelių sakinių veiksmus, tie sakiniai sujungiami į vieną sudėtinį sakinį. Į ciklą gali įeiti kitas ciklas, o tame, mažesniame, cikle gali būti naujų ciklų ir t.t.

3 pavyzdys. Programa skaičių nuo 1 iki *n* *m*-tiesiems laipsniams sumuoti:

```
program LaipsniųSuma;
  var n,          { intervalo pabaiga }
      r,          { laipsnio rodiklis }
      p,          { vieno skaičiaus laipsnis }
      s,          { laipsnių suma }
      i, j: integer;
begin
  read(n, r);
  s := 0;
  for i := 1 to n do
    begin
      p := 1;
      for j := 1 to r do
        p := p*i;
        s := s+p
      end;
   writeln(s)
end.
```

Išorinis ciklas (jo antraštė **for** i := 1 **to** n **do**) atliekamas *n* kartų – su kiekvienu skaičiumi vieną kartą. Šio ciklo kartojamas sakinys yra sudėtinis. Jame yra kitas ciklas *i*-tojo skaičiaus *r*-tajam laipsniui rasti. Šis ciklas su kiekvienu skaičiumi atliekamas tiek kartų, kokiu laipsniu reikia jį kelti. Pavyzdžiui, kai $r = 3$, ciklas su kiekvienu skaičiumi atliekamas tris kartus ($j = 1, 2, 3$). Jeigu $n = 20$, tai išorinis ciklas atliekamas 20 kartų, o vidinis $20 \cdot 3 = 60$ kartų.

Abiejų reiškinių ($reiškinys_1$ ir $reiškinys_2$) reikšmės yra apskaičiuojamos tik vieną kartą – prieš atliekant ciklą, įsimenamos ir toliau naudojamos. Todėl komponentų keitimai ciklo sakinyje nebeturi įtakos ciklo kartojimų skaičiui.

Jeigu $reiškinys_1 = reiškinys_2$, tai ciklas atliekamas vieną kartą.

Jeigu $reiškinys_1 > reiškinys_2$, tai ciklas neatliekamas nė karto.

Jei ciklo veiksmus reikia atlikti perbėgant ciklo kintamojo reikšmes atvirkščia tvarka – nuo didžiausios iki mažiausios, tai tada ciklo antraštėje vietoj žodžio **to** rašomas žodis **downto**.

for *ciklo kintamasis* := $reiškinys_1$ **downto** $reiškinys_2$ **do**

Šis ciklas atliekamas atvirkščiai, negu anksčiau nagrinėtas. Pirmoji ciklo kintamojo reikšmė yra $reiškinys_1$, antroji – $pred(reiškinys_1)$ trečioji – $pred(pred(reiškinys_1))$ ir t.t. Paskutinį kartą sakinys atliekamas kai ciklo kintamojo reikšmė yra $reiškinys_2$.

5 pavyzdys. Programa, rašanti visus vienženklius skaičius atvirkščiai (mažėjančiai).

```
program atv;
  var sk: integer;
begin
  for sk := 9 downto 0 do
    write(sk: 2);
  writeln
end.
```

Ciklas su žodžiu **downto** atliekamas vieną kartą, jeigu $reiškinys_1 = reiškinys_2$, t.y. taip pat, kaip ir ankstesnis ciklas. Tačiau jis neatliekamas nė karto esant priešingai sąlygai, t.y. jeigu $reiškinys_1 < reiškinys_2$.

Ciklo **for** kintamasis ir abu jo reiškiniai turi būti tuo paties duomenų tipo. Dažniausiai jie būna sveikojo tipo. Tačiau gali būti ir bet kurio kito paprastojo diskrečiojo duomenų tipo, pavyzdžiui, vardinio, loginio, simbolinio (žr. 4.1 skyr.). Dėl to apibrėždami ciklo veikimą vengėme sakyti, kad kai ciklas kartojamas, tai jo kintamojo reikšmė padidinama arba sumažinama vienetu, o sakėme, kad imama tolesnė arba prieš ją einanti reikšmė. Mat sudėties ir atimties operacijas turi tik skaičiai. Kiti diskretieji duomenų tipai (pvz., vardinis) aritmetinių operacijų neturi. Tačiau visi diskretieji duomenų tipai turi tolesnės arba prieš einančios reikšmės gavimo operacijas (*succ* ir *pred*).

Realieji skaičiai nepriklauso diskrečiųjų duomenų tipų grupei, todėl nei ciklo kintamojo, nei kurio nors reiškinio tipas negali būti realusis.

Ciklo kintamasis duoda nemažai naudos, bet su juo atsiranda ir problemų. Kokia bus ciklo kintamojo reikšmė, atlikus ciklą? Į šį klausimą ne taip lengva atsakyti. Logiškai galvojant, kintamojo reikšmė turėtų būti lygi reiškinio, esančio po žodžio **to**, reikšmei. Tačiau kompiuteris dažniausiai ciklo kintamojo reikšmę pirmiau padidina (sumažina) ir tik po to tikrina, ar ji dar patenka į leistinus rėžius. Tokiu atveju ciklo kintamojo reikšmė būtų lygi *succ*(*reiškinys2*) arba *pred*(*reiškinys1*) – **downto** atveju. Be to, tokiu atveju nebūtų galima parašyti ciklo, perrenkančio visas kurio nors diskrečiojo duomenų tipo reikšmes. Dėl to, susitarta ciklo kintamojo reikšmės nenaudoti už ciklo ribų. (Laikoma, kad užbaigto ciklo kintamojo reikšmė tampa neapibrėžta.).

Ciklas **for** yra skirtas iš anksto žinomam veiksmų kartojimų skaičiui užrašyti, todėl nedera jo paskirties iškreipti keičiant ciklo viduje jo kintamojo reikšmę.

Visus Paskalio kalbos ciklus skirstėme į dvi rūšis: žinomo kartojimo skaičiaus (**for**) ir nežinomo kartojimų skaičiaus (**while** ir **repeat**).

Kada katrą ciklą naudoti? Jeigu dar prieš atliekant ciklą žinoma, kiek kartų reikės jį kartoti, tai labiau tinka ciklas, kurio antraštė prasideda žodžiu **for**. Tada programa būna trumpesnė ir vaizdesnė. Jeigu kartojimų skaičiaus prieš atliekant ciklą negalima nustatyti, reikia naudoti ciklą su žodžiu **while** arba **repeat**.

Uždaviniai

4.4.1. Kiek kartų bus atliekamas ciklas, kurio antraštė yra šitokia:

```
a) for k := 10 to 20 do
b) for k := 20 to 10 do
c) for k := -1 to 1 do
d) for k := a to a do
e) for k := a to a+10 do
f) for k := -a to a do
g) for k := 10 downto 20 do
h) for k := 20 downto 10 do
```

4.4.2. Kokius skaičius matysime ekrane atlikę šią programą.

```
program bandymas;
  var a, b, i: integer;
begin
  a := 0;
  b := 5;
  for i := a to b do
    begin
      write(i: 2);
      b := b-1
    end;
  writeln
end.
```

4.4.3. Sudarykite programą visų teigiamų nelyginių dviženklių skaičių kvadratų sumai rasti.

4.4.4. Pradiniai duomenys – geometrinės progresijos pirmasis narys ir jos vardiklis. Sudarykite programą, kuri apskaičiuotų ir išspausdintų 10 pirmųjų progresijos narių.

4.4.5. Pradiniai duomenys – aritmetinės progresijos pirmasis narys, jos skirtumas ir natūralusis skaičius *n*. Sudarykite programą, kuri apskaičiuotų ir išspausdintų 10 iš eilės einančių progresijos narių, pradedant *n*-tuoju.

4.4.6. Turime vieną ciklą įdėtą į kitą ciklą:

```
for i := 1 to n do
  for j := 1 to m do
```

Raskite (jei galima) tokias *n* ir *m* reikšmes (konstantas), kad ciklai būtų atliekami šitiek kartų:

a) išorinis ciklas 5 kartus, vidinis – 10 kartų;

b) išorinis ciklas 10 kartų, vidinis – 10 kartų;

c) išorinis ciklas 10 kartų, vidinis – 0 kartų;

d) vidinis ciklas 12 kartų;

e) vidinis ciklas 13 kartų;

f) išorinis ciklas 0 kartų, vidinis – 10 kartų.

4.4.7. Kiek kartų bus atliekamas ciklas

```
n := 10;
for k := 1 to n do
  n := n-1
```

4.4.8. Ar ciklas **for** gali būti nebaigtinis?

4.4.9. Kokie nekorektiškumai yra šioje programoje?

```
program GalimosKlaidos;
  var k: integer;
begin
  for k := 1 to 10 do
```

```
    begin
      k := k+1;
      writeln(k: 5);
    end;
  writeln(k)
end.
```

4.4.10. Turime dvi programas, skirtas tam pačiam darbui: dešimčiai skaitmenų rašyti. Kuri iš jų neteisinga ir kodėl?

```
program pirma;
  var k: 0..9;
begin
  while k <= 9 do
    begin
      write(k: 2);
      k := k + 1
    end
end.
```

```
program antra;
  var k: 0..9;
begin
  for k := 0 to 9 do
    write(k: 2)
end.
```

4.4.11. Programuotojas, išbandęs 4.4.9 uždavinio programą Turbo Paskaliu, nepastebėjo jokių klaidų. Tuo tarpu uždavinio sprendime pateikti net du nekorektiškumai. Kodėl teorija nesiderina su praktika?

Praktikos darbas

4.4.1. Daugianario reikšmės skaičiavimas. Parašysime programą, kuri apskaičiuotų daugianario

$a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_1 x + a_0$

reikšmę.

Analogišką programą trečiojo laipsnio daugianariui jau rašėme (žr. 3.4 skyr., 2 pavyzdį). Turėdami ciklą galime lengvai parašyti programą bet kurio laipsnio daugianariui skaičiuoti.

Daugianarį pertvarkysime – užrašysime jį Hornerio schema

$(... (((a_n)x + a_{n-1})x + a_{n-2})x + ... + a_1)x + a_0$

Šitaip užrašyto daugianario reikšmei skaičiuoti reikės mažiau daugybos operacijų. Be to, pradėjus jį skaičiuoti nuo koeficiento su didžiausiu indeksu, galima nežinoti, kiek dar bus koeficientų, t. y. koks daugianario laipsnis.

Rašome programą.

```
program daugianaris;
  var x,
      a,                { kuris nors koeficientas }
      px: real;         { daugianario reikšmė }
      koef: text;       { daugianario koeficientai }

begin
  assign(koef, 'KOEF.TXT'); reset(koef);
  px := 0;
  write('x = ');
  read(x);
  write(' Surinkite daugianario koeficientus ');
  writeln('a(n), a(n-1), ... a(1), a(0)');
```

```
  writeln (' Nepraleiskite nulių! ');
  while not eof(koef) do
    begin
      px := px*x;
      read(koef, a);
      px := px+a
    end;
  writeln;
  writeln('x =  ', x: 10, px: 15)
end.
```

Kintamasis *a* atstovauja visiems daugianario koeficientams. Kai perskaitoma kurio nors koeficiento reikšmė, ji pridedama prie jau turimos daugianario reikšmės ir pasidaro nebereikalinga. Todėl tą patį kintamąjį galima panaudoti naujai perskaitytai daugianario reikšmei įsiminti.

Naudodamiesi kompiuteriu ir šia programa, apskaičiuokite šitokių daugianarių reikšmes:

a) $5x^3 + 2x^2 + 3x + 2$, kai x = 1, 100;

b) $2x^8 + x^6 + 4x^4 - 5x^2 + x$, kai x = 0, 1, 2, 3.

4.5. Skaičių perrinkimo uždaviniai

Kompiuteris veikia greitai. Todėl uždavinius galima spręsti bandymų keliu. Paeiliui išbandomi visi variantai ir atrenkami tie, kurie tenkina uždavinio sąlygas.

Matematikoje yra suformuluota nemažai uždavinių, kuriuose reikia rasti skaičius, tenkinančias tam tikras savybes. Kol nebuvo kompiuterių, tokių skaičių paieška buvo savotiškas sportas. Dabar kompiuteriu galima greitai patikrinti daugybę skaičių ir sportu tampa programų tokiems uždaviniams rašymas.

Skaičių, tenkinančių tam tikras savybes, paieškos algoritmus galima suskirstyti į tris grupes:

1. Pilnas perrinkimas. Tikrinami visi skaičiai iš eilės. Aišku, kad šitaip galima patikrinti tik baigtinio intervalo skaičius.

2. Ribotas perrinkimas. Įrodoma, kad ieškomą savybę gali turėti tik skaičiai, patenkantys į tam tikrus intervalus. Tada pakanka patikrinti tik tuos intervalus.

3. Skaičių generavimas. Randamas būdas, kaip gauti visus skaičius, tenkinančius reikiamą sąlygą, be perrinkimo. Tada rašoma programa, generuojanti ieškomą skaičių aibę.

Pilnas perrinkimas. Tai paprasčiausi algoritmai, kai tikrinami visi skaičius iš eilės.

1 pavyzdys. Reikia rasti visus intervalo [m; n] skaičius, kurių skaitmenų sumos kubas lygus pačiam skaičiui.

```
program kubai;
  var m, n,              { intervalo rėžiai }
      k,                 { kandidatas į ieškomus skaičius }
      s,                 { skaičiaus k skaitmenų suma }
      kk: integer;
begin
  read(m, n);
  for k := m to n do
    begin
      s := 0;
      kk := k;
      while kk > 0 do
        begin
          s := s + kk mod 10;
          kk := kk div 10
        end;
      if k = s*s*s then writeln(k)
```

```
    end
end.
```

2 pavyzdys. Visų skaičiaus daliklių radimas.

```
program dalikliai;
  var n,                { duotas skaičius }
      dal: integer;     { kandidatas į duoto skaičiaus daliklius }
begin
  read(n);
  for dal := 1 to n do
    if n mod dal = 0 then writeln(dal)
end.
```

Šioje programoje jau galima įžiūrėti tam tikrą perrinkimo ribojimą – tikrinami tik skaičiai, nedidesni už duotą skaičių. Mat laikome, kad savaime aišku, jog skaičiaus daliklis negali būti didesnis už jį patį.

Ribotas perrinkimas.

3 pavyzdys. Skaičius *n* neturi daliklių iš intervalo [*n* ***div*** *2 + 1; n - 1*]. Atmetę šį intervalą beveik dvigubai sumažinsime ciklo kartojimų skaičių.

```
program dalikliai2;
  var n,                { duotas skaičius }
      dal: integer;     { kandidatas į duoto skaičiaus daliklius }
begin
  read(n);
  for dal := 1 to n div 2 do
    if n mod dal = 0 then writeln(dal);
  writeln(n)             { n yra skaičiaus n daliklis }
end.
```

Paieškos intervalą galima dar daugiau apriboti (žr. 4.5.1 užd.).

Paminėsime keletą įdomiomis savybėmis pasižyminčių skaičių.

Draugiškaisiais skaičiais vadinami du natūralieji skaičiai, kurių kiekvienas lygus antrojo skaičiaus visų daliklių, išskyrus jį patį, sumai.

Tobuluoju skaičiumi vadinamas natūralusis skaičius, lygus visų savo daliklių, išskyrus jį patį, sumai.

Automorfiniu skaičiumi vadinamas skaičius, lygus savo kvadrato paskutiniams skaitmenims.

Šių ir daugelio kitų įdomių skaičių paieškos algoritmai yra aprašyti knygoje [3].

4 pavyzdys. Tarkime, kad reikia parašyti programą, kuri rastų *visus* skaičius, kurie lygūs savo skaitmenų kubų sumai. Iš pradžių tai atrodo neįveikiamas uždavinys, kadangi neįmanoma patikrinti visos begalinės skaičių aibės. Tačiau galima įrodyti, kad šią savybę gali turėti tik skaičiai, patenkantys į baigtinį intervalą, nedidesni kaip keturženkliai. Įrodymui reikalingus duomenis apie įvairiaženklius skaičius surašysime į lentelę.

Keliaženkliai skaičiai	Mažiausias skaičius	Didžiausia skaitmenų kubų suma
Vienaženkliai	1	729
Dviženkliai	10	1458
Triženkliai	100	2187
Keturženkliai	1000	2916

Penkiaženkliai	10000	3645

Skaitmenų kubų sumos auga lėčiau, negu skaičiai. Didžiausia penkiaženklio skaičiaus skaitmenų suma yra mažesnė už patį mažiausią penkiaženklį skaičių. Vadinasi penkiaženkliai skaičiai ieškomos savybės nebegali turėti. Juo labiau šešiaženkliai, septynženkliai ir t.t. Todėl pakanka apriboti paiešką iki keturženklių skaičių. Bet ir keturženklius skaičius galima tikrinti ne visus, o tik iki 2916, nes didesnės keturženlių skaitmenų kubų sumos už šį skaičių nebėra.

```
program kubai2;
  var m, n,             { intervalo rėžiai }
      k,                { kandidatas į ieškomus skaičius }
      s,                { skaičiaus k skaitmenų suma }
      sss,              { skaičiaus k skaitmenų kubų suma }
      kk: integer;
begin
  for k := 1 to 2916 do
    begin
      sss := 0;
      kk := k;
      while kk > 0 do
        begin
          s := kk mod 10;
          sss := sss + s*s*s;
          kk := kk div 10
        end;
      if k = sss then writeln(k)
    end
end.
```

Skaičių, kurie lygūs savo skaitmenų sumos kubui, intervalas taip pat ribotas. Toks uždavinys buvo pateiktas pirmojo Lietuvos jaunųjų programuotojų konkurso dalyviams ir yra aprašytas knygoje [4].

Skaičių generavimas.

5 pavyzdys. Tarkime, kad reikia rasti visus duoto intervalo skaičius, kurie dalosi iš 7. Paprasčiausias sprendimas – patikrinti visus intervalo skaičius. Tačiau galima perrinkimo išvengti: rasti pirmąjį (mažiausią) skaičių, o kitus gauti prie jo pridedant po 7.

```
program septynetas;
  var m, n,             { intervalo rėžiai }
      k: integer;       { ieškomas skaičius }
begin
  read(m, n);
  k := m;
  while k mod 7 <> 0 do
    k := k + 1;
  while k <= n do
    begin
      writeln(k);
      k := k + 7
    end
end.
```

Uždaviniai

4.5.1. Suradus vieną skaičiaus daliklį ir iš jo padalijus skaičių galima gauti kitą daliklį – rasto daliklio „porininką“. Pavyzdžiui, suradę skaičiaus 28 daliklį 2 galima gauti kitą skaičiaus daliklį 28 **div** 2 = 14. Tuo pasinaudodami patobulinkite 3 pavyzdžio programą sumažindami perrinkimų skaičių.

4.5.2. Parašykite programą skaičiaus skaidymui į pirminius daugiklius.

4.5.3. Parašykite programą, kuri rašytų pirmąjį dešimtuką skaičių, dalių iš 2, 3 ir 5.

4.6. Diskretieji ir tolydieji duomenys

Kompiuterio prigimtis yra *diskreti*, dar kitaip vadinama *skaitmeninė*. Visi duomenys jame saugomi diskrečiu pavidalu: nulių ir vienetų kodais. Tačiau matematikoje ir realiame gyvenime yra dar kita duomenų rūšis – tolydieji duomenys. Tai realieji skaičiai. Realiųjų skaičių aibė yra begalinė. Skaičių begalybė pasireiškia ne tik skaičiams be galo didėjant arba be galo mažėjant, bet ir į gylį. Tarp bet kurių dviejų realiųjų skaičių galima rasti be galo daug kitų realiųjų skaičių. Pavyzdžiui, tarp skaičių

1,123456788 ir
1,123456789 yra skaičiai:

1,1234567881

1,12345678812

1,12345678820001 ir t.t.

Kompiuteris ne tik diskretus, bet ir baigtinis: skaičiui saugoti skiriamas baigtinis atmintinės kiekis. Taigi kompiuteryje realieji skaičiai vaizduojami apytiksliai, baigtinėmis sveikųjų skaičių sekomis. Dėl to atsiranda paklaidos.

Daugumos praktinių uždavinių skaičiavimo paklaida būna maža. Tačiau kartais pravartu rezultatus apskaičiuoti su kuo mažesne paklaida. Ypač tai svarbu, kai paklaida kaupiasi, atliekant skaičiavimus cikluose. Žinant, kaip realieji skaičiai vaizduojami kompiuteryje, galima suvokti, kokias paklaidas jis daro. Apie tai ir kalbėsime.

Beveik visuose kompiuteriuose realieji skaičiai vaizduojami vadinamuoju *slankaus kablelio* pavidalu. Paaiškinsime jį. Vienam realiajam skaičiui skirtas atmintinės laukas padalijamas į dvi dalis. Vienoje jų saugomas pats skaičius, kitoje – daugiklio laipsnis. Sakykime, kad turime kompiuterį, kuriame skaičiui skiriamos 6 skiltys o daugiklio laipsniui – 2 skiltys. Tada laukas būtų padalytas taip:

±	X	X	X	X	X	X	±	X	X
skaičius							daugiklio laipsnis		

Čia ženklu `X` žymimas skaitmuo.

Pavyzdžiui, skaičius `0.25E-8` tokiame kompiuteryje būtų užrašytas šitaip:

+	2	5	0	0	0	0	–	0	8

Raidė `E` į atmintį neįrašoma – ir taip aišku skaičiaus ir jo daugiklio laipsnio riba. Nerašomas ir trupmeninę dalį nuo sveikosios atskiriantis taškas (kablelis). Susitariama, kad jis bus visada toje pačioje vietoje, pavyzdžiui, prieš pirmąją skaičiaus skiltį, t. y. visi skaičiai neturės sveikosios dalies. Na, o tų skaičių, kurie iš tikrųjų turi sveikąją dalį, taškas perkeliamas į kairę, atitinkamai padidinus daugiklio laipsnį. Pavyzdžiui, skaičius

```
67.5R-5
```

būtų užrašomas taip

```
0.675E-3
```

Siekiant kuo didesnio skaičių tikslumo, jie „sukoreguojami“ taip, kad pirmoji trupmeninė skiltis būtų reikšminga t. y. nebūtų lygi nuliui. Pavyzdžiui, skaičius

```
0.00125E-5
```

užrašomas taip

`0.125E-7`

Pateikiame daugiau pavyzdžių.

Skaičius programoje	Skaičius kompiuteryje								
`125E50`	+	1	2	5	0	0	0	+	5	3
`0.0000000137`	+	1	3	7	0	0	0	−	0	7
`3.14`	+	3	1	4	0	0	0	+	0	1
`-3E8`	−	3	0	0	0	0	0	+	0	9
`12345.12345`	+	1	2	3	4	5	1	+	0	5
`9999999999E89`	+	9	9	9	9	9	9	+	9	9

Pats didžiausias skaičius, kurį galima įrašyti į čia pavaizduoto kompiuterio atmintį, būtų `0.999999E99`, t. y. $10^{99} - 10^{93}$. Pats mažiausias (pagal modulį) šešių skilčių tikslumo skaičius būtų `0.100000E-99`, t. y. 10^{-100}. Taigi leistinas skaičių intervalas labai platus. Neįmanoma net įsivaizduoti tokių didelių arba tokių mažų skaičių. Aptariamo kompiuterio atmintyje saugomos tik šešios reikšminės skaičiaus skiltys, nepaisant, kokie jie – dideli ar maži. Todėl visame leistiname skaičių intervale santykinė paklaida yra ta pati, o absoliučioji keičiasi – didėja, didėjant skaičiams. Pavyzdžiui, skaičius

100000000,0

nagrinėjamame kompiuteryje būtų užrašytas taip:

`0.100000E09`

Jam artimiausias didesnis skaičius, kurį galima įrašyti į kompiuterio atmintinę būtų

`0.100001E09`

t. y.

100001000.0

Taigi du gretimi skaičiai skiriasi 1000, t. y. kompiuteris skaičiuoja tūkstančių tikslumu. Todėl galime būti beveik tikri: kai `a = 1E8`, skaičiuojant nagrinėjamu kompiuteriu, lygybė

```
a = a+1
```

bus tenkinama!

Kai skaičiai labai dideli, pavyzdžiui, `a = 1E50`, lygybė

```
a = a+1
```

gali būti tenkinama skaičiuojant kiekvienu kompiuteriu!

Taigi realiaisiais skaičiais galima apytiksliai pavaizduoti ir labai didelius sveikuosius skaičius. Todėl posakis, kad realiaisiais skaičiais programavime vadinami trupmeniniai skaičiai, nėra tikslus. Geriau būtų sakyti, kad realiaisiais skaičiais vadinami apytiksliai skaičiai.

Daugiau medžiagos apie realiuosius skaičius galima rasti straipsnyje [5].

Realieji skaičiai kompiuteriuose vaizduojami nevienodai – gali būti kitoks skilčių skaičius skaičiui arba laipsnio rodikliui saugoti, skaičiai koduojami dvejetaine sistema. Tačiau bendri vaizdavimo principai yra tie patys.

Uždaviniai

4.6.1. Ką parašys kompiuteris, atlikęs šitokią programą

```
program paklaida;
  const da = 654321E5;
        db = 654320E5;       { dideli skaičiai }
        mc = 654321;         { mažas skaičius }
  var a, b: real;
```

```
begin
  a := da+mc-db;
  b := da-db+mc;
  writeln(a);
  writeln(b)
end.
```

4.6.2. Kurios iš šių reiškinių reikšmės yra nepriimtinos, t.y. rodančios, kad jas pateikęs kompiuteris dirba blogai (yra sugedęs)?

a) `a/5.0*5.0 = a*5.0/5.0` `false`

b) `a/5.0*5.0 = a*5.0/5.0` `true`

c) `5/2.0 = 2.5` `true`

d) `5/2.0 = 10.0` `true`

e) `25000 = 25E3` `false`

f) `25000 <> 25E3` `true`

g) `2500 = 30E3` `true`

h) `12E50 = 12E50+1.0` `true`

i) `12E50 = 12E50+1.0` `false`

5. PROGRAMAVIMO TECHNOLOGIJOS ELEMENTAI

Jau įpusėjome knygą. Jau sudarėme keliolikos paprastų uždavinių programas ir matėme, kad kiekviena jų yra savita. Dažnai ne iš karto pavyksta sudaryti gerą programą, tokią, kad ją galima būtų laikyti tobulybės viršūne. Taigi, atėjo laikas pamąstyti, kaip rašome programas, paieškoti būdų, kaip programuoti greitai ir gerai.

Tam pačiam uždaviniui galima sudaryti daug skirtingų programų. Programavimas yra kūrybinis procesas, ir sunku rasti bendrus receptus, kaip sudaryti kiekvieno uždavinio programą. Tačiau galima suformuoti bendras taisykles, kurios padėtų šį darbą paspartinti, jį geriau atlikti. Tai ypač svarbu sudarant didesnių uždavinių programas. Bet kokį didesnį darbą lengviau įveikti, suskirsčius jį į dalis – mažesnius darbus. Tas pats tinka ir programavimui.

Uždavinio programavimą galima suskirstyti į šitokias dalis – etapus:

1) uždavinio formulavimas;

2) sprendimo metodo parinkimas ar sudarymas;

3) programos rašymas;

4) programos tikrinimas;

5) programos tobulinimas;

6) programos derinimas (išbandymas kompiuteriu).

Programa yra produktas, daiktas, kuriuo naudosis daugelis žmonių. Mus supa daugybė žmogaus sukurtų daiktų. Juos vertiname dėl to, kad jie atlieka jiems skirtas funkcijas. Jais gėrimės, jeigu jie gražūs ir juose matome juos sukūrusių žmonių mintį ir pagarbą mums, t.y. tiems, kas tais daiktais naudojasi. Deja gražių daiktų ne tiek daug, o netikusių mėtosi visur.

„Programavimas yra ir mokslas ir menas. Šie du požiūriai puikiai papildo vienas kitą. Aš jaučiu, kad programavimas panašus į poeziją arba muziką. Yra elegantiškų programų, yra grakščių programų, yra spindinčių programų. Aš teigiu, kad galima parašyti puikias programas, kilnias programas ir išties didingas programas“ (Donaldas Knutas).

5.1. Uždavinio formulavimas

Norint išspręsti bet kokį uždavinį, reikia turėti jo sąlygą. Programavimo uždavinio sąlyga vadinama uždavinio formuluote. Programavimo uždavinio sprendinys (rezultatas) yra programa. Taigi, norint sudaryti programą, reikia turėti užduotį programavimui – uždavinio formuluotę.

Formuluojant uždavinį, reikia nuodugniai išsiaiškinti, kokie reikalavimai keliami būsimai programai. Priešingu atveju galima ko nors nenumatyti ir neatlikti veiksmų, kurie galbūt buvo turimi galvoje, bet nebuvo aiškiai suformuluoti. Uždavinio formuluotėje reikia aiškiai nurodyti, kokie turi būti pradiniai duomenys, ką atlikti pagal programą ir kokių rezultatų norima. (Kaip gauti rezultatus, t. y. kokius veiksmus atlikti, nustatoma vėliau, kai sudaroma programa).

1 pavyzdys. Tarkime, kad gavome užduotį:

Reikia sudaryti programą duotųjų skaičių sumai rasti.

Turime patikslinti, kiek yra duota skaičių, arba (jeigu nežinoma, kiek jų yra) apibrėžti, kaip bus nustatoma tų skaičių sekos pabaiga. Pateikiame kelias šio uždavinio formuluotes.

1. *Pradiniai duomenys – du sveikieji skaičiai. Reikia juos perskaityti, apskaičiuoti ir išspausdinti jų sumą.*

2. *Pradiniai duomenys – dešimt sveikųjų skaičių. Reikia juos perskaityti, apskaičiuoti ir išspausdinti jų sumą.*

3. *Pradiniai duomenys – sveikųjų skaičių, nelygių nuliui, seka. Sekos pabaigoje – nulis. Reikia perskaityti pradinius duomenis, rasti ir išspausdinti jų sumą.*

Kuris šių variantų geriausias, turi nuspręsti būsimasis programos naudotojas, nes jis geriausiai žino savo uždavinį. Kai programos naudotojas yra kartu ir programuotojas (o taip būna mokantis programavimo), jis pats turi nuspręsti, kuris variantas tinkamiausias.

2 pavyzdys. *Pradinis duomuo yra skaičius, reiškiantis metus. Reikia sudaryti programą, pagal kurią galima būtų nustatyti, ar metai yra keliamieji.*

Ar tai, kas pasakyta, galima laikyti tikslia uždavinio formuluote. Be abejo, ne.

Pirma, nenurodyta, apie kokius metus kalbama: šio šimtmečio, šio tūkstantmečio, ar bet kuriuos metus po Kristaus gimimo. O gal tai ir metai prieš Kristų?

Antra, nepasakyta, kokia forma kompiuteris turi pateikti rezultatą.

Patiksliname uždavinio formuluotę:

Pradinis duomuo yra natūralusis skaičius, reiškiantis mūsų eros metus. Reikia sudaryti programą, pagal kurią kompiuteris išspausdintų duotuosius metus ir žodį METAI, o po to – žodį KELIAMIEJI, jeigu metai keliamieji, arba žodį PAPRASTIEJI, jeigu metai paprastieji, arba žodį NETEISINGI, jeigu pradinis duomuo neteisingas.

Tokia uždavinio formuluotė jau pakankamai aiški ir tiksli.

3 pavyzdys. Reikia parašyti programą, kuri patikrintų, ar iš trijų atkarpų, kurių ilgiai duoti, galima sudaryti trikampį. Reikia spausdinti duotuosius atkarpų ilgius ir žodį TAIP, jei galima, arba NE, jei negalima sudaryti trikampio. Išanalizuokime šią formuluotę.

Apie nulinius ir neigiamus atkarpų ilgius neverta ir kalbėti, nes iš tokių atkarpų trikampio sudaryti negalima. Todėl savaime aiškus atsakymas – NE.

Iš pradžių galima pasigesti atkarpų ilgio matavimo vienetų. Bet, gerai pagalvojus, pasidaro aišku, kad jų nurodyti ir nereikia. Svarbu, kad visos atkarpos būtų išmatuotos tais pačiais ilgio vienetais.

Taigi prie šios uždavinio formuluotės nieko pridurti nereikia.

Uždaviniai

5.1.1. Duotos dvi uždavinio formuluotės:

1) *Pradinis duomuo – sveikasis skaičius n. Reikia sudaryti programą visų intervalo (0; n) lyginių skaičių sumai rasti.*

2) *Pradinis duomuo – sveikasis skaičius n. Reikia sudaryti programą visų lyginių skaičių nuo 0 iki n (jei n nelyginis, tai iki n – 1) sumai rasti. Jeigu n < 0, tai sumuoti neigiamus skaičius. Jų suma neigiama.*

Ar šios formuluotės ekvivalenčios, t. y. ar galima sudaryti programą, atitinkančią abi formuluotes?

5.2.2. Uždavinio šitokia formuluotė.

Pradiniai duomenys – šimto sveikųjų skaičių seka. Sudarykite programą, kuri patikrintų, ar duotoje skaičių sekoje yra bent vienas nulis.

Ko šioje formuluotėje trūksta?

5.2. Kaip kuriamas arba pasirenkamas uždavinio sprendimo metodas

Programuotojas turi mokėti išspręsti uždavinį. Tik po to, kai bus sudaryta programa, kompiuteris ją galės atlikti ir išvaduoti žmogų nuo daugelio ilgų duomenų apdorojimo veiksmų.Tačiau juos žmogus turi gerai mokėti atlikti ir sugebėti užrašyti programoje. Taigi programuotojui visų pirma reikia išsiaiškinti, kaip sprendžiamas uždavinys.

Daugelio paprastų uždavinių sprendimo būdas akivaizdus. Pavyzdžiui, jeigu reikia atlikti aritmetinius veiksmus su keletu skaičių, pakanka parašyti aritmetinį reiškinį; o atliekant vienodus veiksmus su daugeliu skaičių, rašomas ciklas ir pan.

Programuojant sudėtingesnius uždavinius, reikia remtis žinomais matematikos dėsniais, taisyklėmis, teoremomis.

1 pavyzdys. Reikia rasti visų nelyginių natūraliųjų skaičių, ne didesnių už n, sumą. Paprasčiausias šio uždavinio sprendimo būdas – parašyti ciklą, pagal kurį būtų sumuojami visi iš eilės einantys nelyginiai skaičiai: 1 + 3 + 5 ... . Tačiau neskubėkime. Ši seka – aritmetinė progresija. O progresijos narių sumą s_n apskaičiuojame pagal formulę:

$$s_n = n \times (a_1 + a_n)/2$$

čia a_1 – pirmas narys,

a_n – paskutinis narys,

n – narių skaičius.

Parašę šią formulę vietoj ciklo, daug kartų sumažinsime kompiuterio darbą.

2 pavyzdys. Grįžkime prie ankstesnio skyrelio 3 pavyzdžio uždavinio. Čia reikia patikrinti, ar iš trijų atkarpų, kurių ilgiai duoti, galima sudaryti trikampį. Norint išspręsti šį uždavinį, pakanka žinoti trikampio savybę, teigiančią, kad kiekviena trikampio kraštinė yra mažesnė už kitų dviejų kraštinių sumą. Pažymėję atkarpų ilgius raidėmis a, b ir c, galime šią savybę išreikšti loginiu reiškiniu

$(a+b > c)$ **and** $(a+c > b)$ **and** $(b+c > a)$,

kurio reikšmė yra *true*, jeigu iš atkarpų galima sudaryti trikampį, ir *false* – priešingu atveju.

3 pavyzdys. Reikia sudaryti programą, kurį nustatytų į kokį pirminių daugiklių skaičių išskaidomas duoto skaičiaus faktorialas. Pavyzdžiui, jeigu pradinis duomuo yra skaičius 5, tai rezultatas turi būti 15, nes

$$10! = 3628800 = 2\times2\times2\times2\times2\times2\times2\times2\times3\times3\times3\times3\times5\times5\times7.$$

Paprasčiausias sprendimo būdas – rasti skaičiaus faktorialą, o po to jį skaidyti į pirminius daugiklius ir suskaičiuoti tų daugiklių skaičių. Tačiau ir nedidelio skaičiaus faktorialas yra labai didelis skaičius. Todėl greitai gausime perpildymą. Betgi kai ir pradinis duomuo, ir rezultatas yra nedideli skaičiai, nelogiška, kad būtų perpildymas. Todėl reikia paieškoti kitokių sprendimo būdų.

Skaičiaus n faktorialas yra skaičių nuo 1 iki n sandauga. Taigi galima skaidyti į pirminius daugiklius atskirus skaičius jų nesudauginus.

Sprendžiant uždavinį, visada reikia nuodugniai išanalizuoti jo formuluotę, neužmiršti visų galimų atvejų, visų galimų pradinių duomenų reikšmių. Kartais sprendžiant uždavinį aptinkama, kad formuluotėje ne viskas pasakyta arba joje yra prieštaravimų. Tada ji tikslinama.

Daugelį uždavinių galima išreikšti lygtimis. Programuotojas, be abejo, turi pats mokėti tas lygtis spręsti ir jų sprendimo algoritmą pateikti programoje. Atskira matematikos šaka (skaičiavimo matematika ir skaičiavimo metodai) nagrinėja matematikos uždavinių, daugiausia lygčių, sprendimo metodus, tinkančius programavimui.

Iš kelių uždavinio sprendimo metodų pasirenkamas vienas, atsižvelgiant į jo universalumą, programavimo paprastumą, kompiuterio laiko ir atminties ekonomiją bei kitas charakteristikas, turinčias įtakos programavimui ir programos atlikimui.

Tenka programuoti įvairių žmogaus veiklos sričių (fizikos, biologijos, muzikos, ekonomikos ir pan.) uždavinius ir atlikti veiksmus su jiems būdingais objektais (gyvūnais, muzikos garsais, ekonominiais rodikliais ir pan.). Uždavinių sprendimo metodus nagrinėja matematika, o veiksmai atliekami su matematiniais objektais: skaičiais, loginėmis reikšmėmis ir pan.Todėl nematematiniai uždaviniai pakeičiami jiems ekvivalenčiais matematikos uždaviniais, kitaip sakant, sudaromas realaus pasaulio uždavinio matematinis modelis. Toks keitimas vadinamas modeliavimu.

Su realaus pasaulio matematiniu modeliu susidūrėme, nagrinėdami 3.1 skyrelio 1 pavyzdį. Tenai tiltus pakeitėme loginiais kintamaisiais ir atlikome su jų reikšmėmis logines operacijas, atspindinčias realų pasaulį.

5.3. Programos rašymas

Kai uždavinys nesudėtingas ir iš karto aišku, kaip jį spręsti, tai nesunku ir jo programą parašyti. Sudėtingesnį uždavinį aprėpti ir programuoti visą iš karto sunku, o labai sudėtingą – išvis neįmanoma. Todėl stambesnis uždavinys skaidomas į smulkesnes dalis ir mėginama rašyti kiekvienos dalies programą. Jeigu kuri nors dalis yra dar per stambi, ji skaidoma į smulkesnes dalis tol, kol tos jų programos pasidaro pakankamai trumpos ir vaizdžios. Šitaip dalimis galima parašyti ir labai stambių uždavinių programas.

Pavyzdys. Parašykime programą 5.2 skyrelio 3 pavyzdžio uždaviniui: į kokį pirminių daugiklių skaičių išskaidomas duoto skaičiaus faktorialas.

Pirmiausia sudarome programos eskizą,

```
program p;
  var n,                    { n! }
      k,                    { faktorialo daugiklis }
      d,                    {vieno k pirminių daugiklių skaičius }
      daugsk: integer;
begin
  read(n);
  daugsk := 0;
  for k := 2 to n do
    begin
      Rasti, į kokį pirminių daugiklių
      skaičių d išskaidomas skaičius k
      daugsk := daugsk + d
    end;
  writeln(daugsk)
end.
```

Tuos veiksmus, kuriuos mokame programuoti, užrašėme Paskalio kalbos žymenimis, o tuos, kurių dar neprogramavome, pavaizdavome stačiakampiu su jame įrašytu žodiniu veiksmų formulavimu. Taigi, suprogramavome tik dalį uždavinio. Tai, kas įrašyta į stačiakampį, dar reikės programuoti. Taigi, programos rašymą suskaidėme į dalis.

Dabar reikia suprogramuoti, t.y. užrašyti Paskalio kalbos žymenimis stačiakampio veiksmus. Juos galima paimti iš 4.5.2 uždavinio sprendimo ir pritaikyti šio uždavinio programai.

```
program p;
  var n, nn,                { n! }
      k, kk,                { faktorialo daugiklis }
```

```
      d,                        { vieno k pirminių daugiklių skaičius }
      daugsk: integer;
begin
  read(n);
  daugsk := 0;
  nn := n;
  for k := 2 to n do
    begin
      kk := k;
      d := 0;
      while kk <= nn do
        begin
          while nn mod kk = 0 do
            begin
              d := d + 1;
              nn := nn div kk
            end;
          kk := kk + 1
        end;
      daugsk := daugsk + d
    end;
  writeln(daugsk)
end.
```

Štai ir visa programa. Perrašydami programos fragmentą iš 4.5.2 uždavinio sprendimo, juose turėjome kai ką pakeisti, ypač kintamųjų vardus. Teko programą papildyti naujų kintamųjų aprašais. Tai nepatogu ir keičiant galima padaryti klaidų. Šių nepatogumų galima išvengti naudojantis autonominėmis programų dalimis – funkcijomis ir procedūromis. Apie jas kalbėsime 6 skyriuje.

5.4. Programos tikrinimas ir derinimas

Jau spėjome patirti, kad, sudarant programą, suklysti labai lengva. Todėl ją reikia kruopščiai ir nuodugniai patikrinti dar prieš pateikiant kompiuteriui.

Tikrinant kiekvieną programą, rekomenduojama įsitikinti, ar:

1) nėra sintaksės klaidų;

2) aprašyti visi (kintamųjų) vardai;

3) apibrėžiamos visų kintamųjų reikšmės prieš jas vartojant;

4) programos veiksmai baigtiniai;

5) teisingi rezultatai.

Sintaksės (arba gramatikos) klaidos atsiranda, kai, rašydami programą, nusižengiame programavimo kalbos (mūsų atveju – Paskalio) gramatikai – pavartojame ne tuos arba ne ten, kur reikia, skyrybos ženklus, netaisyklingai parašome kokią nors kalbos konstrukciją ir pan. Sintaksės klaidos didesnių problemų nekelia, nes jas atranda kompiliatorius ir čia pat informuoja apie jas.

Beveik visi kompiliatoriai neaptinka neapibrėžtų reikšmių vartojimo. Primename, kad visuose reiškiniuose (prieskyros sakinio dešinėje pusėje, po žodžių **`if`**, **`while`**, **`to`**, rašymo sakiniuose ir kitur) galima vartoti tik apibrėžtas (prieš tai priskirtas) kintamųjų reikšmes. Kai programą sudaro vien paprastieji (prieskyros, duomenų skaitymo arba rašymo) sakiniai, reikšmių apibrėžtumą patikrinti labai paprasta: prieš kiekvieną sakinį, kuriame vartojama kintamojo reikšmė, bet kurioje programos vietoje turi būti sakinys, suteikiantis reikšmę tam kintamajam.

Kai kintamajam priskiriantis reikšmę sakinys yra sąlyginiame sakinyje arba cikle, tai juo pasitikėti ne visada galima. Mat toks sakinys gali būti kartais neatliekamas, o tada kintamojo reikšmė gali likti neapibrėžta.

1 pavyzdys. Turime programą skaičiaus absoliutiniam didumui (moduliui) rasti:

```
program abs;
  var  a, b: integer;
begin
  read(a);
  if a > 0 then b := a
           else if a < 0 then b := -a;
  writeln(b)
end.
```

Kai $a = 0$, neatliekama nė viena sąlyginio sakinio šaka ir kintamojo b reikšmė lieka neapibrėžta. Vadinasi, ši programa neteisinga vien dėl to, kad jos rezultatas yra neapibrėžtas tik su viena pradinio duomens reikšme.

Jeigu kintamojo reikšmė apibrėžiama tik sąlyginiame sakinyje, tai reikia įsitikinti, ar, esant bet kokiems pradiniams duomenims, bus atliekama bent viena sakinio šaka, kurioje yra apibrėžiantis reikšmę sakinys.

Jeigu kintamasis gauna reikšmę tik cikle, tai reikia įsitikinti, ar tas ciklas visada bus bent kartą atliekamas.

Kompiuteris gali atlikti tik tokią programą, kurioje reikalaujama baigtinio veiksmų skaičiaus. Kai programoje yra ciklų, kurių antraštė prasideda žodžiu **while**, tai atsiranda pavojus ciklui niekada nesibaigti.

Tam, kad ciklas būtų baigtinis, būtina (bet nepakankama) sąlyga, jog ciklo viduje būtų keičiama bent viena reikšmė, esanti ciklo antraštės sąlygoje. Pavyzdžiui, net ir nesigilindami į atliekamų veiksmų prasmę, galime pasakyti, kad nebaigtinis yra ciklas

```
while a > 0 do
  write(a)
```

Jeigu ciklo antraštės sąlyga $a > 0$ yra tenkinama prieš atliekant ciklą, tai ji bus ir toliau tenkinama, nes cikle nekeičiama kintamojo a reikšmė.

Kad minėta sąlyga yra tik būtina, bet nepakankama, matyti iš šitokio pavyzdžio:

```
while a > 0 do
  a := a + 1
```

Čia ciklo antraštės sąlygos kintamojo a reikšmė keičiama didėjimo kryptimi. Jeigu sąlyga $a > 0$ buvo tenkinama prieš atliekant ciklą pirmą kartą, ji bus tenkinama ir atlikus ciklą kiek norima kartų.

Išnagrinėkime kitą ciklą:

```
while a >= 0 do
  a := a - 1
```

Šis ciklas yra baigtinis, kad ir kokia didelė būtų kintamojo a reikšmė prieš atliekant ciklą: atiminėjant iš šios reikšmės vienetą, ji vis tiek kada nors pasidarys neigiama, sąlyga ciklo antraštėje nebus tenkinama ir ciklas baigsis.

Kaip patikrinti, ar programos rezultatai teisingi?

Pats paprasčiausias būdas – išbandyti kompiuteriu. Labai svarbu parinkti tinkamus kontrolinius pradinius duomenis. Jeigu programoje yra ciklų, pravartu kontrolinius duomenis parinkti tokius, kad ciklas nebūtų nė karto atliekamas, kad būtų atliekamas vieną kartą ir kad būtų atliekamas kelis kartus. Kai programoje yra sąlyginių sakinių, reikia parinkti tokius kontrolinius duomenis, kad bent vieną kartą būtų atlikta kiekvieno sąlyginio sakinio dalis (šaka).

2 pavyzdys. Tarkime, turime programą skaičiui 2 pakelti laipsniu n:

```
program laipsnis;
  var n,          { laipsnio rodiklis }
      p,          { rezultatas }
      k: integer;
begin
  read(n);
  p := 1;
  for k := 1 to n do
    p := p*2;
  writeln(p)
end.
```

Jai patikrinti parenkame keletą pradinių duomenų ir pateikę juos kompiuteriui gauname tokius rezultatus:

```
-5   1
 0   1
 1   2
 2   4
 5  32
```

Rezultatai teisingi, kai $n \geq 0$. Kai n < 0, tai visada gauname vienetą. Todėl galima tvirtinti, kad ši programa tinka dvejetui kelti teigiamu laipsniu.

Ar galima teigti, kad programos rezultatai yra teisingi, esant bet kurioms pradinių duomenų reikšmėms, jei jie teisingi su keliomis pasirinktomis pradinių duomenų reikšmėmis? Deja, ne. Išnagrinėkime dar vieną pavyzdį.

3 pavyzdys. Tarkime, yra sudaryta kita programa skaičiui 2 kelti laipsniu *n*.

```
program laip;
  var k, n, p: integer;
begin
  read(n);
  p := 1;
  for k := 1 to n do
    p := k*2;
  writeln(p)
end.
```

Apskaičiavę šios programos rezultatus, esant pradiniams duomenims 0, 1, 2, gauname 1, 2, 4. Jie teisingi. Tačiau daryti išvadą, kad programa `laip` teisinga, dar per anksti. Paėmę bet kurį didesnį pradinį duomenį, įsitikinsime, kad programos rezultatas neteisingas. Pavyzdžiui, kai $n = 3$, rezultatas yra 6, o turi būti 8, nes $2^3 = 8$.

Taigi, apsiribojus keliais pradiniais duomenimis arba keliais jų variantais, galima praleisti kaip tik tuos duomenis, su kuriais gaunami klaidingi rezultatai. Kitaip tariant, su atskirais pradiniais duomenimis galima įrodyti tik tai, kad programa klaidinga (jei ji iš tikrųjų klaidinga), o ne tai, kad teisinga (nors iš tikrųjų ji ir yra teisinga). Todėl labai svarbu parinkti tinkamus kontrolinius pradinius duomenis, kuriems esant rezultatai galėtų būti neteisingi.

Apie kontrolinių duomenų parinkimą galima paskaityti straipsnyje [6].

Klaidų ieškojimas ir taisymas kompiuteriu vadinamas programos derinimu.

Ne visas klaidas pavyksta greitai ir lengvai rasti. Sudėtingose programose pasitaiko giliai „pasislėpusių“ klaidų. Kaip jų ieškoti ir jas rasti, bendrų receptų nėra. Ieškant klaidų programose, praverčia logika, įžvalgumas ir, žinoma, geras dalyko – programavimo – išmanymas.

Atlikdami programą kompiuteriu, ne tik aptinkame klaidas, bet pabūname jos naudotoju: paruošiame pradinius duomenis, skaitome kompiuterio išspausdintus rezultatus, t.y. įsitikiname, ar patogu naudotis sudarytąja programa. Praktiškai išbandydami programą, galime patobulinti ir

uždavinio formuluotę. Tada reikia vėl grįžti prie ankstesnių programavimo darbų ir pakoreguoti jau sudarytą programą.

Kad programa teisinga su visomis galimomis pradinių duomenų reikšmėmis, reikia įrodyti matematiškai. Tai padaryti kur kas sunkiau, negu patikrinti programos rezultatą su atskiromis pradinių duomenų reikšmėmis.

4 pavyzdys. Pabandykime įsitikinti, kad programos `laipsnis` (žr. 2 pavyzdį) rezultatas teisingas, ne skaičiuodami, o išskleisdami ciklą formule, esant įvairioms pradinio duomens reikšmėms.

```
p := 1;
for k := 1 to n do
  p := p*2
```

Kai ciklas $n < 1$, ciklas neatliekamas nė karto. Vadinasi, ciklas rezultato reikšmės nekeičia ir išlieka ankstesnė jo reikšmė $p = 1$.

Kai $n > 1$, ciklas kartojamas n kartų. Vadinasi,

$p = 1 \times 2$, kai $n = 1$,

$p = 1 \times 2 \times 2$, kai $n = 2$,

$p = 1 \times 2 \times 2 \times 2$ kai $n = 3$.

Nesunku padaryti išvadą, kad

$$p = \underbrace{1 \times 2 \times 2 \times \ldots \times 2}_{\text{n kartų}}$$

su kiekvienu n. Tai reiškia, kad $p = 2^n$.

Uždaviniai

5.4.1. Duoti du sąlyginiai sakiniai

1)
```
if a > 0 then b :=  5
         else b := 10;
```

2)
```
if a < 0 then b := 10
         else b :=  5
```

Kokiai kintamojo `a` reikšmei esant jie neekvivalentūs (t.y. jų rezultatai skirtingi)?

5.4.2. Kintamojo a reikšmė iš anksto nežinoma. Nurodykite sąlygas, kurias tenkins kintamojo reikšmė atlikus šiuos sakinius:

a)
```
if a > 10 then a := 10
          else a := a + 5;
```

b)
```
if a > 10 then a := 1
          else a := 0
```

5.4.3. Kintamojo *a* reikšmė iš anksto nežinoma. Nurodykite sąlygas, kurias tenkins kintamojo reikšmė atlikus šiuos sakinius:

a)
```
while a > 10 do
  a := a-1;
```

b)
```
while a > 10 do
  a := a-10
```

5.4.4. Kokias sąlygas turi tenkinti kintamojo `k` reikšmė, kad būtų baigtiniai šie ciklai:

a)
```
while c < 0 do
   c := c+k;
```

b)
```
while k <> 0 do
   k := k+1
```

c)
```
while k <> 0 do
```

```
k := k-2
```

5.5. Programos tobulinimas

Ne iš karto pavyksta sudaryti grakščią ir ekonomišką programą. Kai programa sudaryta, netgi patikrinta, dažnai gimsta naujų idėjų programai patobulinti – padaryti ją trumpesnę, lakoniškesnę, aiškesnę arba ekonomiškesnę (pavyzdžiui, greičiau atliekamą).

1 pavyzdys. Tarkime, skaičiaus moduliui rasti buvo parašytas šitoks sakinys:

```
if a >= 0 then b := a
  else if a < 0 then b := -a
```

Nesunku pastebėti, kad antroji sąlyginio sakinio dalis apima visus tuos atvejus, kurie netenkina pirmosios sąlygos ($a >= 0$). Todėl sakinį galima užrašyti trumpiau (ir aiškiau):

```
if a >= 0 then b := a
          else b := -a
```

2 pavyzdys. Turime ciklą:

```
for k := 1 to 1000 do
  s := s + p + k*k
```

Kiekvieną kartą atliekant jį, prie kintamojo *s* reikšmės pridedama ta pati kintamojo *p* reikšmė. Todėl sudėtį galima iškelti iš ciklo, pakeitus ją daugyba:

```
s := s + 1000*p;
for k := 1 to 1000 do
  s := s + k*k
```

Dabar 1000 sudėčių bus pakeista viena daugyba, todėl kompiuteris programą atliks greičiau.

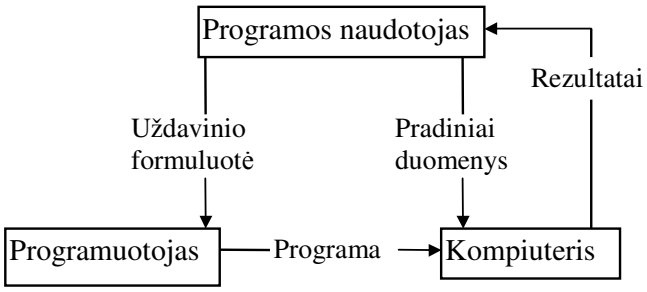
Programos rašymas, tikrinimas ir tobulinimas yra tarpusavyje glaudžiai susiję. Patobulinę patikrintą programą, vėl turime ją tikrinti, nes tobulindami galėjome padaryti naujų klaidų. Jas pataisius gali vėl atsirasti tobulintinų vietų. Kartais tobulinant ir taisant klaidas, programoje atsiranda tiek daug pakeitimų ir „užlopytų“ vietų, kad ją geriau parašyti iš naujo.

Programuotojas, rašydamas programą, pripranta prie jos ir nepastebi trūkumų. Dažnai programą pavyksta patobulinti, kai ją skaitome praėjus ilgesniam laikui nuo jos rašymo. Tada būname ją primiršę ir geriau sekasi viską pergalvoti iš naujo. Trūkumus lengviau pastebėti ir kito programuotojo sudarytoje programoje.

5.6. Programavimo stilius

Naudotojas naudoja programą savo uždaviniams spręsti. Jam svarbu, kokį uždavinį sprendžia ta programa, kaip pateikti pradinius duomenis kompiuteriui ir kaip kompiuteris išspausdins rezultatus. Kadangi duomenų skaitymo ir rašymo veiksmai nurodomi programoje, tai naudotojui geresnė ta programa, kuriai paprasčiau paruošti pradinius duomenis ir kuri patogiau (vaizdžiau) išspausdins rezultatus.

Programą sudaro programuotojas, ją atlieka kompiuteris, o ja naudojasi žmogus – naudotojas (25 pav.). Vadinasi, programą galima vertinti programuotojo, kompiuterio ir naudotojo požiūriu.



25 pav. Programuotojo, naudotojo ir kompiuterio sąryšis

Naudotojas naudoja programą savo uždaviniams spręsti. Jam svarbu, kokį uždavinį sprendžia ta programa, kaip pateikti pradinius duomenis kompiuteriui ir kaip kompiuteris pateiks rezultatus. Kadangi duomenų skaitymo ir rašymo veiksmai nurodomi programoje, tai naudotojui geresnė ta programa, kuriai paprasčiau paruošti pradinius duomenis ir kuri patogiau (vaizdžiau) parodys (išspausdins) rezultatus.

1 pavyzdys. Pateikiame dvi programas tam pačiam uždaviniui spręsti – dviem laiko intervalams, išreikštiems valandomis ir minutėmis, sudėti.

```
program laikas1;
  var ah, amin,              { pirmas intervalas, val., min. }
      bh, bmin,              { antras intervalas, val., min. }
      h, min: integer;       { intervalų suma, val., min. }
begin
  read(ah, amin, bh, bmin);
  min := ah*60 + amin + bh*60 + bmin;
  h := min div 60;
  min := min mod 60;
  writeln(h, min: 3)
end.
```

```
program laikas2;
var ah, amin,                { pirmas intervalas, val., min. }
    bh, bmin,                { antras intervalas, val., min. }
    h, min: integer;         { intervalų suma, val., min. }
begin
  read(ah, bh, amin, bmin);
  min := ah*60 + amin + bh*60 + bmin;
  h := min div 60;
  min := min mod 60;
  writeln(h, min: 3)
end.
```

Programos skiriasi tik pradinių duomenų pateikimo tvarka. Be abejo, racionaliau išdėstyti pradiniai duomenys programoje `laikas1` – čia pirmiau eina pirmojo intervalo valandos ir minutės, po to antrojo intervalo valandos ir minutės. Todėl naudotojas geriau pasirinks programą `laikas1`.

Programos tekstas naudotojo nedomina – jis gali net nemokėti jo perskaityti. Jam reikalinga tik trumpa informacija, kaip pateikti pradinius duomenis ir kokius rezultatus jis gaus. Tokia informacija vadinama naudojimosi programa instrukcija. Programos `laikas1` ji būtų maždaug tokia:

Programa `laikas1` *skirta dviem laiko intervalams sumuoti. Pradiniai duomenys – 4 sveikieji skaičiai, pateikiami šitokia tvarka: pirmojo intervalo valandos ir minutės, antrojo intervalo valandos ir minutės. Rezultatai – laiko intervalų suma: pirmiau valandos, po to – minutės.*

Kompiuteris atlieka veiksmus, surašytus programoje, formaliai, paraidžiui. Todėl jo požiūriu programos vaizdumas ar duomenų paruošimo patogumas neturi prasmės. Tačiau kompiuteriui svarbus programos ekonomiškumas. Iš tikrųjų programos ekonomiškumas svarbus naudotojui (ne kompiuteriui). Tik šis parametras siejasi su kompiuterio parametrais (skaičiavimų greičiu ir atmintinės kiekiu). Todėl kompiuterio požiūriu geresnė programa yra ta, kuriai mažiau reikia atmintinės (mažiau kintamųjų) ir mažiau veiksmų (ypač – mažesnis ciklų kartojimų skaičius).

2 pavyzdys. Pateikiame dvi programas aritmetinės progresijos pirmųjų 100 narių sumai rasti. Abiejų programų pradiniai duomenys tie patys – pirmasis narys ir skirtumas.

```
program prog;
  const n = 100;        { sumuojamų narių skaičius }
  var a1,               { pirmasis narys }
      aj,               { narys }
      d,                { skirtumas }
      suma,             { narių suma }
      j: integer;
begin
  read(a1, d);
  suma := a1; aj := a1;
  for j := 2 to n do
    begin
      aj := aj + a1;
      suma := suma + aj
    end;
  writeln(suma)
end.
```

```
program progresija;
  const n = 100;        { sumuojamų narių skaičius }
  var a1,               { pirmasis narys }
      an,               { paskutinis narys }
      d,                { skirtumas }
      suma: integer;    { narių suma }
begin
  read(a1, d);
  an := a1 + (n-1)*d;
  suma := n * (a1 + an) div 2;
  writeln(suma)
end.
```

Aišku, ekonomiškesnė yra programa `progresija`, kurioje progresijos narių suma skaičiuojama pagal formules, be ciklo.

Programuotojas rašo programą kompiuteriui. Atrodytų, kad mes esame programų kūrėjai, o kompiuteris – jų skaitytojas ir atlikėjas. Kompiuteris tikrai yra programų atlikėjas. Tačiau programas skaito ir žmogus, norėdamas sužinoti, kokie veiksmai užrašyti joje, pasisemti patirties iš jau sudarytų programų, kai mokosi programuoti, kai sudaro naujas programas naudodamasis jau sudarytų programų tekstais. Programas tenka skaityti, kai jos tobulinamos, kai taisomos klaidos, kai pritaikomos naujiems uždaviniams. Taigi ir pačiam programos autoriui dažnai tenka pabūti jos skaitytoju – nuolat grįžti prie anksčiau sudarytų ir jau pamirštų programos dalių.

Taigi rašant programą, reikia nuolat galvoti, kad ją nagrinės žmogus ir įsivaizduoti esant jos skaitytoju. Todėl reikia stengtis, kad programa būtų aiški, lengvai suprantama. Kai programa aiški, joje mažiau būna ir klaidų, mažiau „tamsių vietų“ klaidoms pasislėpti.

Norint parašyti gerai suprantamą ir lengvai skaitomą programą, reikia daugiau pastangų. Tačiau sugaištas laikas beveik visada atsiperka: programa rašoma vieną kartą, o skaitoma daug kartų, programą rašo vienas žmogus, o skaito daugelis.

Taigi programuotojo požiūriu vertingesnė yra aiškesnė ir vaizdesnė programa. Šis vertinimo kriterijus gali prieštarauti kompiuteriniam: tam, kad programa būtų aiškesnė, kartais reikia daugiau kintamųjų arba suprantamesnių, bet lėčiau atliekamų veiksmų. Bet kompiuterių greitis ir atmintinė nuolat didėja, todėl kuriant programas vis dažniau atsižvelgiama į žmogaus poreikius. Pagaliau kompiuteris tarnauja žmogui, o ne atvirkščiai.

Taigi programos kokybę reikia vertinti trimis požiūriais:

1) programuotojo (programos teksto skaitytojo),

2) programos naudotojo ir

3) kompiuterio.

Bet kokiu atveju programa turi būti teisinga – negalima kalbėti apie programos kokybę, jeigu ji neteisinga, arba švelniau pasakius – sprendžia ne tą uždavinį, kurio buvo tikėtasi.

5.7. Rezultatų apipavidalinimas

Savaime aišku, kad turi būti patogu naudotis programa. Iš programos teikiamų pranešimų turi būti aišku, ką kiekvienu momentu reikia daryti, kokius pradinius duomenis pateikti. Rezultatai turi būti išduodami aiškiu ir suprantamu pavidalu.

Jeigu programą numatoma dažniau ir ilgiau naudoti, tai reikia stengtis jos rezultatus pateikti kuo vaizdžiau, su paaiškinimais. Rašydami programą sugaišime daugiau laiko, tačiau ja bus lengviau naudotis. Kompiuteris gali ne tik apskaičiuoti rezultatus, bet ir spausdinti visiškai parengtus dokumentus – algalapius, žiniaraščius, tvarkaraščius ir pan. Kaip tai padaryti pailiustruosime pavyzdžiu.

Pavyzdys. Programa duoto intervalo sveikųjų skaičių kvadratų ir kubų lentelei spausdinti.

```
program KvadrataiKubai;
  var nuo, iki,        { skaičių intervalas }
      n: integer;
begin
  read(nuo, iki);
  for n := nuo to iki do
    writeln(n: 5, n*n: 5, n*n*n: 5)
end.
```

Jeigu šiai programai pateiktume pradinius duomenis

```
20 30
```

tai kompiuterio ekrane pamatytume šitokius rezultatus:

```
   20   400  8000
   21   441  9261
   22   484 10648
   23   529 12167
   24   576 13824
   25   625 15625
   26   676 17576
   27   729 19683
   28   784 21952
   29   841 24389
   30   900 27000
```

Kai domimės tik algoritmu ir eksperimentuojame su kompiuteriu, tai toks rezultatų pavaizdavimas mus tenkina. Tuo tarpu jeigu programą planuotume platinti naudotojams, reikėtų, kad naudotojas matytų, ką ši programa skaičiuoja, kokių pradinių duomenų jai reikia, o rezultatai turėtų būti pateikti vaizdžiau, pavyzdžiui lentele. Papildysime programą visais šiais naudotojui reikalingais komponentais.

```
program KvadrataiKubai;
  var nuo, iki,        { skaičių intervalas }
      n: integer;
begin
  writeln('Programa duoto intervalo skaičių');
  writeln('kvadratams ir kubams skaičiuoti');
  writeln;
  write('Intervalo pradžia: ');
  read(nuo);
  write('Intervalo pabaiga: ');
  read(iki);
  writeln;
  writeln('   Kvadratai ir kubai ');
  writeln;
  writeln('+-------+-------+-------+');
  writeln(':       :       :       :');
  writeln(':     n :   n*n : n*n*n :');
  writeln(':       :       :       :');
  writeln('+-------+-------+-------+');
  for n := nuo to iki do
    writeln(': ', n: 5, ' : ', n*n: 5, ' : ', n*n*n: 5,' :');
  writeln('+-------+-------+-------+')
end.
```

Dabar kompiuteris informuos apie tai ką ši programa skaičiuoja, paprašys pradinių duomenų: intervalo pradžios ir pabaigos. Pasibaigus programai ekrane matysime štai tokį vaizdą:

```
Programa duoto intervalo skaičių
kvadratams ir kubams skaičiuoti

Intervalo pradžia: 20
Intervalo pabaiga: 30
```

```
   Kvadratai ir kubai

+-------+-------+-------+
:       :       :       :
:     n :   n*n : n*n*n :
:       :       :       :
+-------+-------+-------+
:    20 :   400 :  8000 :
:    21 :   441 :  9261 :
:    22 :   484 : 10648 :
:    23 :   529 : 12167 :
:    24 :   576 : 13824 :
:    25 :   625 : 15625 :
:    26 :   676 : 17576 :
:    27 :   729 : 19683 :
:    28 :   784 : 21952 :
:    29 :   841 : 24389 :
:    30 :   900 : 27000 :
+-------+-------+-------+
```

Lentelės rėmeliai dar nelabai gražūs – jie sudaryti iš simbolių, kuriuos galima surinkti kompiuterio klaviatūra. Gražesnę lentelę galima padaryti iš pseudografikos ženklų. Tą pačią programą pakeisime taip, kad kompiuteris lentelės rėmelius sudarytų iš pseudografikos ženklų.

```
program KvadrataiKubai;
  var nuo, iki, n: integer;
begin
  writeln('Programa duoto intervalo skaičių');
  writeln('kvadratams ir kubams skaičiuoti');
  writeln;
  write('Intervalo pradžia: ');
  read(nuo);
  write('Intervalo pabaiga: ');
  read(iki);
  writeln('   Kvadratai ir kubai ');
  writeln;
  writeln('╔═══════╤═══════╤═══════╗');
  writeln('│       │       │       │');
  writeln('│     n │   n*n │ n*n*n │');
  writeln('│       │       │       │');
  writeln('╟───────┼───────┼───────╢');
  for n := nuo to iki do
    writeln('╟ ', n:5, ' │ ', n*n:5, ' │ ', n*n*n:5,' ╢');
  writeln('╚═══════╧═══════╧═══════╝')
end.
```

Dabar kompiuteris rezultatus pateiks surašytus į štai tokią lentelę:

```
Kvadratai ir kubai
```

n	n*n	n*n*n
20	400	8000
21	441	9261
22	484	10648
23	529	12167
24	576	13824
25	625	15625
26	676	17576
27	729	19683
28	784	21952
29	841	24389
30	900	27000

Atkreipiame dėmesį į tai, kad programos tekste lentelę formuojantys sakiniai ženklai išdėstyti taip, kad būtų galima matyti, koks „paveikslas“ bus piešiamas kompiuterio ekrane.

Apipavidalinant rezultatus svarbus kiekvienas rašomas simbolis ir kiekviena tuščia vieta (tarpas). Todėl būsimą ekrano vaizdą geriau pirma suprojektuoti ant languoto popieriaus ir po to rašyti programą. Programoje padarytos klaidos lengvai pastebimos ją atlikus kompiuteriu.

Dar didesnes rezultatų vaizdavimo galimybes duoda tikroji grafika (ne pseudo grafika!). Tačiau mes pagrindinį dėmesį skirsime algoritmams ir grafikos nenagrinėsime.

Praktikos darbas

5.7.1. Daugybos lentelė. Suprojektuokite daugybos lentelę ir sudarykite programą, kuri apskaičiuotų lentelės elementus ir juos įrašytų į ekrane rodomą lentelę.

5.8. Programos teksto apipavidalinimas

Kompiuteriui nesvarbu, kaip išdėstytas programos tekstas. Tačiau žmogui nagrinėti programą kur kas lengviau, kai aiškios sakinių ribos, lengvai suvokiama programos struktūra.

Programą lengviau skaityti, kai programos teksto išdėstymas atspindi programos struktūrą, kai iš teksto vaizdžiai matosi kiekvienos valdymo struktūros pradžia ir pabaiga bei struktūrų hierarchija – kaip struktūros įdėtos viena į kitą. Tai išreiškiama programos teksto eilučių lygiavimu ir atitraukimu nuo kairiojo puslapio krašto. Lygiavimo taisyklės labai paprastos: lygiaverčiai sakiniai, t.y. priklausantys tai pačiai sakinių sekai vienodai atitraukiami nuo kairiojo eilutės krašto – lygiuojami. Į sakinį įdėti kiti sakiniai (jie sudaro jau kitą, žemesnį, hierarchijos lygį) daugiau patraukiami į dešinę. Per kiek ženklų patraukti į dešinę, kaip išdėstyti sakinių pradžios ir pabaigos bazinius žodžius (pvz., **`begin`** – **`end`**, **`case`** – **`end`**, **`repeat`** – **`until`**), yra stiliaus dalykas. Tačiau gero stiliaus taisyklės reikalauja, kad visos programos tekstas būtų išdėstytas pagal vienodas taisykles. Šioje knygoje kiekvieną naują sakinių hierarchijos lygį pradedame su nemažesniu kaip dviejų pozicijų poslinkiu į dešinę, kiekvieną žodį **`end`** rašome po jį atitinkančiu žodžiu **`begin`**.

Pateiksime lygiavimo schemos pavyzdį

```
read ...
a := ...
for ...
  begin
    a :=
    while ...
       case ...
         5: repeat
              if ...
                then
                else
              b := ...
            until ...
         6: for ...
        end
  end;

  while...
    if ...

     write ...
```

Šitoks programos teksto išdėstymas primena knygos turinį. Smulkesnių skyrelių pavadinimai daugiau patraukti į dešinę, negu juos gaubiančių stambių skyrių.

Įvairiau lygiuojami sąlyginiai sakiniai. Kai veiksmai šakojasi į dvi trumpai užrašomas kryptis, žodį **else** rašomas po jį atitinkančiu žodžiu **then**, pavyzdžiui:

```
if a = 0 then write(a)
         else write(b)
```

Šakojimąsi į dvi sudėtingesnės šakas rekomenduojama užrašyti šitaip:

```
if a = 0
  then ...
  else ...
```

Kai veiksmai šakojasi į daugelį krypčių, kiekviena nauja žodžių **else** ir **if** pora šiek tiek patraukiama į dešinę, pavyzdžiui:

```
if a = b then write(a+1)
  else if a = c then write(a+2)
    else if a = d then write(a+3)
      else write(a)
```

Šitaip rašant tekstas mažiau nuslenka į dešinę.

Programą lengviau skaityti, kai vienoje eilutėje yra vienas sakinys. Programos vaizdumas nenukenčia ir kai į vieną eilutę sudedami keli labai trumpi ir tarpusavyje susiję sakiniai, pavyzdžiui:

```
a := 0; b := 0
```

Į vieną eilutę galima parašyti ir neilgą sudėtinį sakinį, pavyzdžiui:

```
begin read(a); write(a) end
```

Vienos eilutės tekstas taip pat turi struktūrą: vieni žodžiai ar simboliai yra stipriau susiję, negu kiti. Sąsajų stiprumas išreiškiamas tarpais. Tarp logiškai išsiskiriančių simbolių, žodžių ar jų grupių, esančių vienoje eilutėje, reikia palikti didesnį tarpą, negu tų grupių viduje. Pavyzdžiui, po vieną tarpą rekomenduojama palikti abipus prieskyros simbolio :=. Jeigu reiškinyje (be skliaustų) yra santykio ir aritmetinių operacijų, tai, palikę po tarpą abipus santykio operacijų, pabrėšime, kad jos atliekamos paskiausiai. Ta pačia taisykle reikia vadovautis ir tada, kai tenka dalį sakinio kelti į naują eilutę: geriau sakinį skaldyti į mažiau susijusias dalis. Pavyzdžiui:

```
if a - 12 = b*2
  then alfa := alfa*ilgis*(plotis + sigma) +
```

```
                (a*b + c*c)
  else write('Galutiniai rezultatai ',
             alfa*alfa - 128: 8,
             ilgis + plotis)
```

Daugybos ir dalybos operacijų prioritetas aukštesnis (jos pirmiau atliekamos), negu sudėties ir atimties (jos paskiau atliekamos). Todėl abipus sudėties ir atimties operacijų simbolių reikėtų palikti po tarpą, o abipus daugybos ir dalybos – nepalikti. Tačiau padėti komplikuoja sveikųjų skaičių dalybos operacijos **div** ir **mod**, žymimos žodžiais. Nepalikus tarpų jie susilietų su gretimais kintamųjų (operandų) vardais. Todėl kai reiškinyje yra operacijos **div** arba **mod** tenka palikti tarpus abipus visų dviviečių operacijų simbolių.

Kitais atvejais tekstą išdėstant eilutėje reikėtų laikytis lietuvių kalbos rašybos taisyklių. Priimta tarpą palikti po dvitaškio, kabliataškio, kablelio, bet ne prieš šiuos simbolius. Programose ypač svarbu palikti tarpą po kabliataškio, nes jis skiria vieną sakinį nuo kito.

Programose kiek savitą paskirtį turi skliaustai. Paprastaisiais skliaustais suskliaudžiami funkcijos parametrai. Tarp funkcijos vardo atidarančio parametrų skliaustų tarpo palikti nereikia. Mat sąsaja tarp funkcijos vardo ir jos parametrų yra stipriausia (turi aukščiausią prioritetą). Paliktas tarpas, pavyzdžiui,

```
abs (a+b)*c
```

sudarytų klaidingą įspūdį, kad funkcijos parametras yra visas reiškinys ir reikia pirmiau padauginti ir po to skaičiuoti funkcijos reikšmę. Todėl šį pavyzdį reikia perrašyti šitaip:

```
abs(a+b) * c
```

Tarp suskliausto teksto ir skliaustų vidinių pusių lietuvių kalboje tarpai nepaliekami. Tačiau tarpai programose tarp komentarų ir juos gaubiančių figūrinių skliaustų padeda atskirti komentarus nuo skliaustų. Skliaustai atlieka lyg ir komentarų rėmelių vaidmenį, panašiai kaip ir pseudografikos simboliai.

Kartais rėmelių vaidmenį atlieka ir dvitaškiai aprašuose, pavyzdžiui,

```
var realusis : real;
    seikasis : integer;
    s        : char;
    log      : boolean;
```

Plačiau apie programos teksto išdėstymą rašoma straipsnyje [7].

Uždaviniai

5.8.1. Pašalinkite nereikalingus simbolius iš programos

```
program simboliai;
  var a, b, i, s: integer;
begin
  read(a);
  i := 0; s := 0;
  for i := 1 to a do
    begin
      s := s+(i*i)*(i+1)
    end;
  writeln(s);
end.
```

5.8.2. Su žemiau pateikta programa atlikite šiuos veiksmus: 1) nustatykite, ką matysite ekrane atlikę šią programą, 2) sulygiuokite programos eilutes ir įterpkite reikalingus tarpus, 3) vėl nustatykite, ką matysite ekrane atlikę šią programą.

Kurį programos tekstą: čia pateiktą ar sutvarkytą, buvo lengviau suvokti?

```
program tvarka;
var i,s:integer;
begin s:=1;i:=10;
while i mod 7=0 do
i:=i - 1; s:= s+i * i;
if s > 25 then if s = 99 then s:=s +1 else
s:=s div i-2 else s:=s- 1;writeln(s)end.
```

5.9. Programos ekonomiškumas

Programa yra ekonomiška jeigu ji taupiai naudoja kompiuterio išteklius: laiką ir atmintinę**.**

Laikas. Sakinys į programą rašomas vieną kartą. Kai programa vykdoma, jis gali būti atliekamas vieną kartą, neatliekamas nė karto (jeigu, pavyzdžiui jis yra sąlyginiame sakinyje ir pakliuvo į nevykdomą šaką) arba atliekamas daug kartų (jeigu jis yra cikle). Aišku, kad apčiuopiamą naudą gali duoti daug kartų atliekamų sakinių vykdymo laiko ekonomija. Tarkime, kad sakinys *S* yra cikle, o tas ciklas dar kitame cikle:

```
for i := 1 to 1000 do
  for j := 1 to 1000 do
    S;
```

Sakinys *S* bus atliekamas milijoną kartų. Taigi, kiekviena sutaupyta mikrosekundė virs visa sekunde. Todėl reikia gerai apžiūrėti visus ciklus: ar tikrai jie reikalingi, ar cikluose nėra tokių veiksmų, kuriuos būtų galima iškelti iš ciklų.

1 pavyzdys. Programa nelyginių skaičių nuo 1 iki *n* sumai rasti.

```
program NelygSuma;
  var n,                { intervalo pabaiga }
      suma,             { nelyginių skaičių suma }
      i: integer;
begin
  suma := 0;
  read(n);
  for i := 1 to n do
    if i mod 2 <> 0 then suma := suma + i;
  writeln(suma)
end.
```

Betgi iš eilės einantys nelyginiai skaičiai sudaro aritmetinę progresiją. O narių sumą galima apskaičiuoti pagal formulę

$$s_n = \frac{n(a_1+a_n)}{2}$$

čia a_1 – pirmas narys,

a_n – paskutinis narys,

s_n – narių skaičius.

Pakeitę ciklą formule, gerokai sumažinsime programos darbo laiką.

2 pavyzdys. Tarkime kad turime šitokį programos fragmentą:

```
read(a);
s := 0;
for i := 1 to n do
  for j := 1 to m do
    s := s + a + sqrt(1/i + sqrt(1/j))
```

Kintamojo `a` reikšmė cikle nekeičiama. Todėl netikslinga jo reikšmę n·m kartų (tiek kartų atliekamas prieskyros sakinys) pridėti prie sumos `s`. Geriau pridėti vieną kartą, bet padaugintą iš n·m.

Reiškinio `1/i` reikšmė priklauso nuo išorinio ciklo kintamojo `i`. Vidiniame cikle ji išlieka ta pati. Todėl jos skaičiavimą galima iškelti į išorinį ciklą.

Perrašome programos fragmentą.

```
read(a);
s := a*n*m;
for i := 1 to n do
  begin
    t := 1/i;
    for j := 1 to m do
      s := s + sqrt(t + sqrt(1/j))
  end
```

Programos tekstas pailgėjo, bet programos vykdymo laikas gerokai sutrumpėjo.

Tobulinant programą ekonomiškumo link, nereikia persistengti. Tuos sakinius, kurie atliekami nedaug kartų, geriau parašyti taip, kad jie būtų aiškesni skaitytojui, nors ir keliskart lėčiau atliekami. Kompiuteris to nepajus, o skaitytojas bus laimingas greičiau supratęs programą.

Atmintinė. Kiekvieno kintamojo reikšmei saugoti skiriama vieta kompiuterio atmintinėje. Tačiau vienam mums jau žinomo paprastojo tipo (pvz., sveikųjų skaičių, realiųjų skaičių) kintamajam vietos reikia tiek mažai (kelių baitų), kad jų skaičiaus mažinimas pastebimos ekonomijos neduos. Kas kita, kai programoje naudojamos didelės duomenų struktūros (žr. 7 skyr.) arba rekursija (žr. 8 skyr.). Todėl apie atmintinės ekonomiją kol kas neverta kalbėti.

Užduotys

5.9.1. Parašykite programą nelyginių natūraliųjų skaičių sumai nuo 1 iki *n* rasti naudodamiesi aritmetinės progresijos formule.

Praktikos darbas

5.9.1. Eksperimentas. 2 pavyzdyje pateiktus abu programos fragmentus įdėkite į programas, išbandykite su kompiuteriu, kai $n = 1000$ ir $m = 1000$, nustatykite jų atlikimo laikus.

6. PROGRAMOS SKAIDYMAS Į DALIS

Didelių darbų neįveiktume, jeigu nemokėtume padaryti mažų darbų. Kiekvienas didelis darbas susideda iš daugelio mažų darbų. Ir tūkstančio mylių kelionė prasideda vienu žingsniu – sako kinų patarlė. Reikia tik mokėti darbą suskaidyti į dalis, o padarius mažesnes dalis – jas sujungti į darnią visumą – didelio darbo rezultatą.

Uždavinio dalių programos išreiškiamos funkcijomis ir procedūromis. Po to jos sujungiamos į programą ir gaunama darni viso uždavinio programa.

Ta pati funkcija ar procedūra gali būti panaudota įvairiose programose, panašiai, kaip bet kurio mechanizmo (pvz., laikrodžio, televizoriaus) keičiamas mazgas arba detalė.

6.1. Funkcijos ir procedūros

Paprastesnių uždavinių programos būna trumpos, vaizdžios. Turinčiam įgūdžių jas parašyti nesunku. Jau esame sudarę keliolikos paprastų uždavinių programas ir matėme, kad kiekviena jų yra savita. Be to, tam pačiam uždaviniui galima sudaryti daug skirtingų programų. Todėl ne visada iš karto pavyksta sudaryti gražią, gero stiliaus programą.

Programavimas yra kūrybinis procesas, ir nėra bendrų receptų, kaip sudaryti kiekvieno uždavinio programą. Tačiau galima suformuoti bendras taisykles, kurios padėtų šį darbą paspartinti, jį geriau atlikti.

Didesnį uždavinį visą iš karto išspręsti sunku. Dažnai pavyksta lengviau įveikti jį suskaidžius į dalis. Mažesnes dalis būna lengviau išspręsti. Išsprendę atskirai kiekvieną dalį ir visų dalių sprendimus sujungę į vieną, gauname viso, didelio, uždavinio sprendimą.

Ar galima suskaidyti į dalis didelio uždavinio programavimą?

Be abejo, galima. Tiktai darbas sklandžiau sektųsi, jeigu uždavinio dalis būtų galima programuoti atskirai, nepriklausomai vieną nuo kitos, o po to lengvai, jų nepertvarkant, sujungti į vieną didelę programą. Tokios autonominės programų dalys yra *funkcijos* ir *procedūros.* Jos turi savus autonomiškus aprašus, savus pradinius duomenis ir rezultatus, per kuriuos palaiko ryšius su kitomis programos dalimis.

Skaityti programą, sudarytą iš funkcijų ir procedūrų, taip pat lengviau. Mat aiškiai matosi savarankiškų dalių ribos ir ryšiai tarp jų. Kiekvieną funkciją ir procedūrą galima nagrinėti atskirai.

Su funkcijomis ir procedūromis jau susidūrėme. Tai buvo į Paskalio kalbą integruotos funkcijos (pvz., `abs, sqrt, succ, pred`) bei procedūros (pvz., `read, write`). Jas jau mokame ir panaudoti.

O dabar pateiksime pavyzdį, iš kurio matysime, kad mums reikalinga nauja, funkcija, kurios neturi Paskalis.

Pavyzdys. Pradiniai duomenys – du natūralieji skaičiai *a* ir *b*. Rezultatas apskaičiuojamas pagal formulę

$a^b - b^a$.

Formulė paprasta. Veiksmai taip pat paprasti. Kėlimo laipsniu operacijos Paskalio kalboje nėra. Tačiau kėlimo laipsniu programą jau esame nagrinėję (žr. 4.4 skyr. 1 pavyzdį). Čia kelti laipsniu reikia du kartus, vadinasi, du kartus rašyti tokį pat ciklą tik su kitais kintamaisiais. Programa bus paprastesnė, jeigu kėlimo laipsniu veiksmus aprašysime funkcija.

Sudarome šitokią programos schemą:

```
program formulė;
  var a, b, c: integer;
```

funkcijos `laipsnis` aprašas

```
begin
  read(a, b);
  c := laipsnis(a, b) - laipsnis(b, a);
  write(c)
end.
```

Vardu `laipsnis` pavadinome kėlimo laipsniu funkciją. Jos aprašyti kol kas nemokame (apie tai – kitame skyrelyje). Todėl vietoj būsimo aprašo piešiame stačiakampį. Kai išmoksime sudaryti funkcijos aprašą, stačiakampį pakeisime tokiu aprašu.

Kai pradiniai duomenys perskaityti, reikia apskaičiuoti rezultatą. Tam naudojame funkciją `laipsnis`. Po vardo skliaustuose rašome funkcijos parametrus. Tai pradiniai duomenys funkcijos rezultatui gauti. Laikome, kad pirmasis parametras yra laipsnio pagrindas, o antrasis – laipsnio rodiklis. Dėl to, pirmą kartą kreipdamiesi į funkciją, rašome `laipsnis(a, b)`, o antrą kartą – `laipsnis(b, a)`. Šie du užrašai vadinami *kreipiniais* į funkciją. Kaskart kreipiantis apskaičiuojama funkcijos reikšmė, kuri, atlikus funkcijos veiksmus, atsiranda kreipinio vietoje.

6.2. Funkcijų aprašai

Funkcijos aprašas panašus į programą. Jis turi savą aprašų dalį ir veiksmų dalį.

1 pavyzdys. Aprašykime ankstesnio skyrelio funkciją `laipsnis`:

```
function laipsnis(p, n: integer): integer;
                  { p – laipsnio pagrindas }
                  { n – laipsnio rodiklis }
  var pn, i: integer;
begin
  pn := 1;
  for i := 1 to n do
    pn := pn*p;
  laipsnis := pn
end.
```

Pirmoji eilutė yra funkcijos antraštė. Žodis **`function`** pasako, kad vardu `laipsnis` pavadinta funkcija. Kaip ir kreipinyje, po funkcijos vardo skliaustuose rašomi parametrai. Apraše nurodomi dar ir jų tipai. Funkcija `laipsnis` turi du parametrus (laipsnio pagrindą ir laipsnio rodiklį). Abu parametrai yra sveikieji skaičiai. Jų vardai `p` ir `n`. Už skliaustų – dvitaškis, o po jo nurodomas funkcijos reikšmės (rezultato) tipas. Kėlimo laipsniu rezultatas yra sveikasis skaičius, todėl rašomas žodis `integer`.

Funkcijos antraštė panaši į kintamojo aprašą, tik vietoj žodžio **`var`** rašomas žodis **`function`** ir skliaustuose nurodomi parametrai. Be to, funkcijos aprašas dar nurodo, kokius veiksmus reikia atlikti, kad būtų gauta funkcijos reikšmė. Tie veiksmai ir sudaro minėtą savarankišką programos dalį, turinčią savus kintamuosius ir jų aprašus, savus sakinius. Šita programos dalis vadinama funkcijos programa, visa kita – pagrindine programos dalimi.

Rezultatas (funkcijos reikšmė) priskiriamas funkcijos vardui. Šiame pavyzdyje tai užrašyta sakiniu

```
laipsnis := pn
```

Funkcijų aprašai eina po kintamųjų aprašų. Visos programoje vartojamos funkcijos turi būti aprašytos (žinoma, išskyrus standartines, kurių aprašyti nereikia).

2 pavyzdys. Funkciją `laipsnis` įtraukime į ankstesnio skyrelio programą ir išnagrinėkime, kaip ir kada atliekami jos veiksmai.

```
program formulė;
  var a, b, c: integer;
  function laipsnis (p, n: integer): integer;
                                { p – laipsnio pagrindas }
                                { n – laipsnio rodiklis }
    var pn, i: integer;
  begin
    pn := 1;
    for i := 1 to n do
      pn := pn*p;
    laipsnis := pn
  end;                  { funkcijos laipsnis aprašo pabaiga }
begin                   { čia prasideda pagrindinė programos dalis }
  read(a, b);
  c := laipsnis(a, b) - laipsnis(b, a);
  write(c)
end.
```

Funkcijos aprašo veiksmus kompiuteris tik įsimena, bet kol kas jų neatlieka. Veiksmai pradedami nuo pagrindinės programos dalies (žr. programą `formulė`), t. y. pirmasis sakinys, kurį atlieka kompiuteris, yra

```
read(a, b)
```

Funkcijos programos veiksmai atliekami tada, kai jų prireikia – kai, atliekant pagrindinę programos dalį, randamas kreipinys į funkciją. Programoje `formulė` tai įvyks, kai bus atliekamas antrasis pagrindinės programos dalies sakinys

```
c := laipsnis(a, b) - laipsnis(b, a)
```

Jo dešinėje pusėje yra du kreipiniai į funkciją `laipsnis`. Vadinasi, atliekant šį sakinį į funkciją bus kreipiamasi du kartus ir du kartus atliekami funkcijos aprašo veiksmai.

Kreipinys yra panašus į funkcijos užrašą matematikoje. Jį galima rašyti reiškinyje. Po funkcijos vardo skliaustuose užrašyti parametrai priklauso funkcijai. Kitaip sakant funkcijos vardą ir jos parametrus sieja aukščiausias operacijų atlikimo prioritetas. Todėl kreipinyje tarp funkcijos vardo ir parametrų (skliaustų) tarpo nerašome. Tuo tarpu funkcijos aprašo parametrų sąrašas būna ilgesnis, negu kreipinyje, nes jame nurodomi ir parametrų tipai. Todėl čia natūralu palikti tarpą tarp funkcijos vardo ir suskliaustų parametrų.

Kreipinyje užrašyti parametrai vadinami faktiniais, o funkcijos aprašo parametrai – formaliaisiais. Šiame pavyzdyje formalieji funkcijos `laipsnis` parametrai yra `p` ir `n`, o faktiniai parametrai kiekviename kreipinyje gali būti vis kitokie. Faktinių parametrų turi būti tiek pat, kiek ir formaliųjų. Prieš atliekant funkciją, formaliesiems parametrams priskiriamos faktinių parametrų reikšmės. Pirmajam formaliajam parametrui priskiriama pirmo faktinio parametro reikšmė, antram – antro ir t. t.

Faktiniai parametrai gali būti ne tik konstantos arba kintamieji, bet ir reiškiniai. Parametrus, kurie keičiami faktinių parametrų reikšmėmis, dar vadiname parametrais-reikšmėmis.

Grįžkime prie pavyzdžio. Vietoj kreipinio

```
laipsnis(a, b)
```

bus atliekami šitokie veiksmai:

1) visiems funkcijos `laipsnis` kintamiesiems ir jos programai paskiriama atmintis;

2) formaliesiems (funkcijos) parametrams priskiriamos faktinių (kreipinio) parametrų reikšmės:

```
p := a; n := b
```

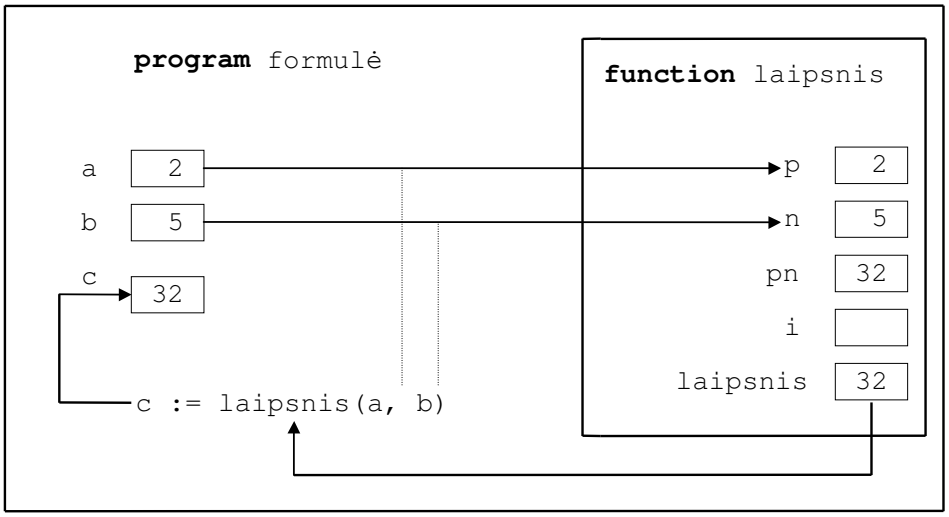
Jeigu, pavyzdžiui, programos pradiniai duomenys būtų skaičiai 2 ir 5 (t. y. a = 2, b = 5), tai formalieji (funkcijos) parametrai įgytų tokias reikšmes:

```
p := 2; n := 5;
```

3) atliekama funkcijos programa, t. y. skaičius 2 pakeliamas laipsniu 5;

4) gautoji funkcijos reikšmė (`laipsnis`) įrašoma į programą vietoj kreipinio ir tęsiami programos pagrindinės dalies veiksmai, t. y. vėl kreipiamasi į funkciją `laipsnis`, tik jau su kitais parametrais (tiksliau – su tais pačiais parametrais, tik sukeistais vietomis).

26 paveiksle schemiškai parodyta, kaip atmintis paskirstoma kintamiesiems. Stačiakampiais pavaizduoti atminties laukai, skirti kintamųjų reikšmėms saugoti, o rodyklėmis – reikšmių persiuntimas iš vieno lauko į kitą. Stačiakampiuose surašytos tokios kintamųjų reikšmės, kurias jie įgyja, kai atliekamas kreipinys `laipsnis(a, b)`, o parametrų reikšmės yra: `a = 2` ir `b = 5`.



26 pav.

Tame pačiame prieskyros sakinyje antrąkart kreipiamasi į funkciją `laipsnis` su kitais parametrais. Dabar skaičius 5 bus keliamas kvadratu.

Po antro kreipinio apskaičiuojama reiškinio reikšmė: 32–25 = 7. Po to kitu sakiniu ji rašoma. Taigi šioje programoje funkcija `laipsnis` buvo panaudota du kartus. Tiek pat kartų buvo atlikti ir joje aprašyti veiksmai.

3 pavyzdys. Funkciją `laipsnis` įjunkime į kitą programą, pagal kurią randama skaičių nuo 1 iki 10 *n*-tųjų laipsnių suma (*n* – pradinis duomuo).

```
program suma;
  const iki = 10;        { iki kiek sumuojama }
  var n,                 { laipsnio rodiklis }
      s: integer;        { laipsnių suma }
      j: 1..iki;
```

```
  function laipsnis (p, n: integer): integer;
    var pn, i: integer;
  begin
    pn := 1;
    for i := 1 to n do
      pn := pn * p;
    laipsnis := pn
  end;
begin
  read(n);
  s := 0;
  for j := 1 to iki  do
    s := s + laipsnis(j, n);
  writeln(s)
end.
```

Kreipinys į funkciją `laipsnis` yra cikle. Į ją bus kreipiamasi tiek kartų, kiek kartų atliekamas ciklas, t. y. 10 kartų.

Kadangi funkcijos aprašas yra savarankiška programos dalis, turinti savus kintamuosius, tai visi funkcijos viduje aprašyti kintamieji skiriasi nuo pagrindinėje programos dalyje aprašytų kintamųjų, netgi jei jie turi tuos pačius vardus. Šiame pavyzdyje vardu `n` buvo pavadintas programos pagrindinės dalies kintamasis ir funkcijos parametras. Tačiau tai skirtingi kintamieji. Kiekvienam jų priskiriama atskira vieta kompiuterio atmintinėje. Programos kintamųjų vardai nepainiojami, nes jų vartojimo sritys nesutampa. Pagrindinėje programos dalyje parašytas kintamasis `n` gali būti vartojamas bet kur programoje, išskyrus funkcijos `laipsnis` vidų, o funkcijoje aprašytas kintamasis `n` – tik funkcijoje.

Programuotojas, sudarydamas funkcijos aprašą (funkcijos programą), gali jos viduje parinkti vardus, neatsižvelgdamas į kitose programos dalyse vartotus vardus. Tai labai svarbi programavimo kalbos ypatybė: ji ypač praverčia sudarant dideles programas, kuriose būna daug vardų.

4 pavyzdys. Sudarysime funkciją didesniajam iš dviejų skaičių rasti. Programą šiam uždaviniui jau esame rašę. Dabar tuos pačius apiforminsime kaip funkcijos aprašą ir galėsime panaudoti bet kur programoje.

```
function max (a, b: integer): integer;
begin
  if a > b then max := a
           else max := b
end;
```

Pritaikykime šią funkciją didžiausiai iš keturių kintamųjų `a`, `b`, `c` ir `d` reikšmei rasti.

```
didž := max(max(a, b), max(c, d))
```

Šiame sakinyje funkcija `max` vartojama tris kartus:

1) `max(a, b)` – didesniajai kintamųjų `a` ir `b` reikšmei rasti;

2) `max(c, d)` – didesniajai kintamųjų `c` ir `d` reikšmei rasti;

3) didžiausiai (iš rastų dviejų didesniųjų) reikšmei rasti.

5 pavyzdys. Jeigu dažnai reikia ieškoti didžiausio iš keturių skaičių, tai pravartu sudaryti šiam tikslui skirtą funkciją. Pavadinkime ją `max4`. Programos viduje vartosime jau žinomą funkciją *max* didesniajam iš keturių skaičių rasti. Sudarome šitokią programą:

```
program didelis;
  var a, b, c, d, e, f, g, h: integer;
  function max (a, b: integer): integer;
  begin
```

```
    if a > b then max := a
             else max := b
  end;
  function max4 (a, b, c, d: integer): integer;
  begin
    max4 := max(max(a, b), max(c, d))
  end;
begin       { čia prasideda pagrindinė programos dalis }
  read(a, b, c, d, e, f, g, h);
  writeln(max4(a, b, c, d));
  writeln(max4(e, f, g, h));
  writeln(max4(a+e, b+f, c+g, d+h));
  writeln(max4(a-e, b-f, c-g, d-h))
end.
```

Iš programos `didelis` pagrindinės dalies kreipiamasi į funkciją `max4` keturis kartus. Kaskart iš funkcijos `max4` į funkciją `max` kreipiamasi tris kartus. Vadinasi, iš viso funkcijos `max4` veiksmai atliekami 4 kartus, o funkcijos `max` – 12 kartų.

Šiuo pavyzdžiu parodėme, kad į funkciją gali būti kreipiamasi ne tik pagrindinėje programos dalyje, bet ir kitoje funkcijoje (šiuo atveju – funkcijoje `max4`).

6 pavyzdys. Sudarysime loginę funkciją, kurios reikšmė yra `true`, jei duotieji metai keliamieji, ir `false` – priešingu atveju. Funkciją įjungsime į programą, nustatančią, kurie metai duotame intervale yra keliamieji.

```
program keliamieji;
  var pradžia, pabaiga, metai: integer;
  function kel (m: integer): boolean;
  begin
    if m <= 1583 then kel :=  m mod 4 = 0
                 else kel := (m mod 400 = 0) or
                             (m mod 100 <> 0) and
                             (m mod 4 = 0)
  end;
begin
  read(pradžia, pabaiga);
  for metai := pradžia to pabaiga do
    if kel(metai) then writeln(metai, ' keliamieji')
                  else writeln(metai, ' paprastieji')
end.
```

Uždaviniai

6.2.1. Duotos funkcijos `max` ir `max4`, aprašytos 5 pavyzdyje. Kurie kreipiniai netaisyklingi ir kodėl?

a) `max(10, 25);`

b) `max(0, 0);`

c) `max(-10, -25);`

d) `max(10, 20, 30);`

e) `max(true, false);`

f) `max4(a, b, c);`

g) `max4(1, 2, 3, 4);`

h) `max4(d, c, b, a).`

6.2.2. Sudarykite funkciją mažesniajam iš dviejų skaičių rasti.

6.2.3. Sudarykite funkciją skaičiaus faktorialui rasti.

6.2.4. Panaudodami funkciją `max4`, parašykite sakinį keturių duotųjų skaičių `a`, `b`, `c` ir `d` didžiausiam paskutiniam skaitmeniui rasti.

Pavyzdžiui, jeigu `a = 25`, `b = 130`, `c = 127`, `d = 1985`, tai rezultatas turi būti `7`.

6.2.5. Sudarykite funkciją natūraliojo skaičiaus skaitmenų sumai rasti.

6.2.6. Sudarykite loginę funkciją, patikrinančią, ar trys duotieji skaičiai sudaro aritmetinę progresiją.

6.2.7. Dažnai spaudoje didesni skaičiai išreiškiami tūkstančiais. Pavyzdžiui, rašoma *124 tūkst. litų* užuot rašius *124 000 litų*. Parašykite funkciją kuri sveikąjį skaičių paverstų tūkstančių skaičiumi jį apvalindama iki sveikų tūkstančių.

6.3. Daugkartinis uždavinio skaidymas į dalis ir jų išreiškimas funkcijomis

Kai didelis uždavinys suskaidomas į dalis, gali pasirodyti, kad ir dalys dar per didelės, kad iš karto būtų galima parašyti jų programas. Tada skaidymas tęsiamas: didesnės dalys vėl skaidomos į mažesnes, iš jų didesnės vėl skaidomos ir t.t. kol nebelieka sunkiai įveikiamų dalių.

Skaidymą pademonstruosime su nelabai dideliu, bet gerai besiskaidančiu į dalis uždaviniu.

Datų uždavinys. Sudarykime programą dienų skaičiui tarp dviejų duotų datų rasti.

Pradiniai duomenys – dvi datos. Kiekviena jų išreiškiama trimis skaičiais: metais, mėnesiu ir diena. Abiejų datų metai ne ankstesni kaip 1583 (jau buvo įvestas Grigaliaus kalendorius). Reikia rasti dienų skaičių tarp tų datų. Jeigu antroji data yra ankstesnė už pirmąją, tai dienų skaičius turi būti neigiamas. Jeigu abi datos tos pačios – dienų skaičius tarp jų lygus nuliui.

Uždavinį suskaidysime į šitokias dalis:

1. Pradinių duomenų (dviejų datų) skaitymas;

2. Dienų skaičiaus tarp dviejų duotųjų datų radimas;

3. Rezultato (dienų skaičiaus) rašymas.

Pirmoji ir trečioji dalys yra pakankamai paprastos. Jas jau galima užrašyti konkrečiais sakiniais. Antroji dalis būtų paprasta, jeigu turėtume funkciją dienų skaičiui tarp dviejų datų rasti. Kol kas jos nėra. Funkciją sudarysime vėliau, o dabar programos eskize rašysime tik funkcijos antraštę, o visą kitą dalį vaizduosime stačiakampiu, kuriame žodžiais paaiškinsime, ko tikimės iš būsimos funkcijos.

```
program dienos;
  var mt1, mn1, dn1,   { pirmoji data }
      mt2, mn2, dn2,   { antroji data }
      d: integer;      { dienų skaičius tarp datų }
  function dsk (mt1, mn1, dn1, mt2, mn2, dn2: integer):
                                                integer;
```

dienų skaičius tarp dviejų datų: pirmoji data: `mt1, mn1, dn1` antroji data: `mt2, mn2, dn2`

```
begin
  read(mt1, mn1, dn1);
```

```
  read(mt2, mn2, dn2);
  d := dsk (mt1, mn1, dn1, mt2, mn2, dn2);
  write(mt1, '.', mn1: 2, '.', dn1: 2, '   ',
        mt2, '.', mn2: 2, '.', dn2: 2, d: 6)
end.
```

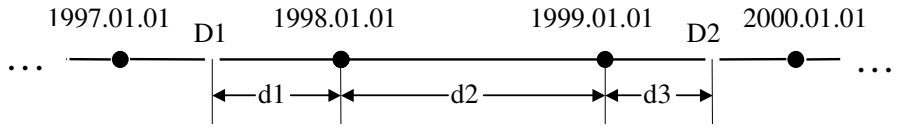
Duomenų skaitymą užrašėme dviem sakiniais, norėdami paryškinti, kad skaitomos dvi skirtingos datos. Į rašymo sakinį įtraukėme ne tik rezultatą, bet ir pradinius duomenis, kad matytųsi iš kurių buvo gautas rezultatas.

Svarbiausią uždavinio dalį – dienų skaičiaus radimą tarp dviejų datų – užrašėme funkcija `dsk`. Dabar ją programuokime.

Pradiniai funkcijos duomenys yra dvi datos, išreikštos šešiais skaičiais. Rezultatas – dienų skaičius tarp tų datų. Pasvarstykime, kaip galima rasti rezultatą.

Pirmas metodas

Uždavinį pavaizduokime grafiškai (kai *mt*1 = 1997, o *mt*2 = 2000).



Reikia rasti tris dienų skaičius:

*d*1 – nuo datos *D*1 iki metų *mt*1 pabaigos,

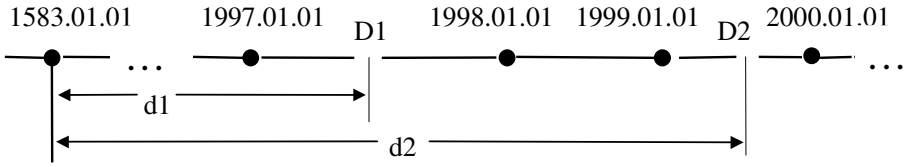
*d*2 – per visus metus (tarp metų *mt*1 pabaigos ir metų *mt*2 pradžios),

*d*3 – nuo metų *mt*2 pradžios iki datos *D*2.

Sudėję visus tris skaičius, gautume rezultatą:

dsk := *d*1 + *d*2 +*d*3.

Antras metodas



Reikia rasti du dienų skaičius:

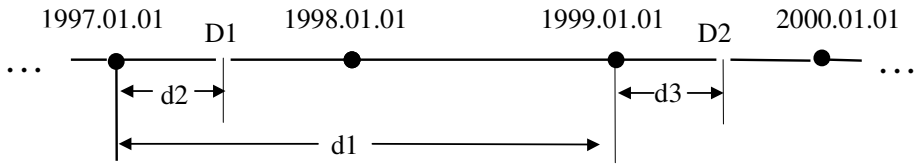
*d*1 – nuo pradinės datos *D*0 (1583 m. sausio 1 d.) iki datos *D*1,

*d*2 – nuo pradinės datos *D*0 iki datos *D*2.

Gautų skaičių skirtumas būtų rezultatas:

dsk := *d*2 – *d*1.

Trečias metodas



Reikia rasti tris dienų skaičius:

*d*1 – nuo metų mt1 pradžios iki metų *mt*2 pradžios,

*d*2 – nuo metų mt1 pradžios iki datos *D*1,

*d*3 – nuo metų mt2 pradžios iki datos *D*2.

Rezultatą gautume pagal formulę

dsk := *d*1 – *d*2 + *d*3.

Kurį metodą pasirinkti?

Jeigu pasirinktume pirmąjį, tai reikėtų sudaryti tris skirtingas programos dalis skaičiams *d*1, *d*2 ir *d*3 rasti. Be to, šis metodas netinka, kai pirmoji data vėlesnė už antrąją. Jeigu pasirinktume antrąjį metodą, tai reikėtų rasti du skaičius – *d*1 ir *d*2. Juos abu galėtume rasti pagal tą pačią funkciją. Todėl antrasis metodas yra universalesnis ir paprastesnis. Tačiau reikia vartoti pradinę datą. Sprendžiant trečiuoju metodu, trims skaičiams rasti reikia dviejų skirtingų funkcijų (skaičius *d*2 ir *d*3 galima rasti pagal tą pačią funkciją).

Pasirenkame trečiąjį metodą, dienų skaičių tarp dviejų datų išreikšdami dviem funkcijomis:

```
function dsk (mt1, mn1, dn1, mt2, mn2, dn2: integer): integer;
  function dmt (mt1, mt2: integer): integer;
```

dienų skaičius nuo metų *mt*1 pradžios iki metų *mt*2 pradžios

```
    function dmtpr (mt, mn, dn: integer): integer;
```

dienų skaičius nuo metų *mt* pradžios iki tų pačių metų mėnesio *mn* dienos *dn*

```
begin
  dsk := dmt(mt1, mt2) - dmtpr(mt1, mn1, dn1)
                       + dmtpr(mt2, mn2, dn2)
end;
```

Štai ir sudarėme funkciją `dsk`. Jos veiksmus išreiškėme kitomis dviem paprastesnėmis funkcijomis. Taigi funkciją `dsk` užbaigėme.Tačiau visa programa dar nebaigta – neaprašytos funkcijos `dmt` ir `dmtpr`. Toliau programuosime funkcijų `dmt` ir `dmtpr` vidų, o funkcijos `dsk` programos neliesime.

Dabar reikia programuoti kitas dvi funkcijas. Funkcijos `dmt` reikšmę galima rasti sudėjus visų metų, esančių duotame intervale, dienų skaičių. Kadangi dienų skaičius metuose priklauso nuo to, ar metai keliamieji, tai reikia naudoti ciklą, o jame – sąlyginį sakinį.

Čia dar reikia nepamiršti, kad antroji data gali būti vėlesnė už pirmąją. O tada, kaip reikalauja uždavinio sąlyga, rezultatas turi būti neigiamas.

```
function dmt (mt1, mt2: integer): integer;
  var m,                { metai }
      d: integer;       { dienos }
  function kel (m: integer): boolean;
```

ar metai keliamieji?

```
  begin
    d := 0;
    for m := mt1 to mt2 - 1 do
      if kel(m) then d := d + 366
                else d := d + 365;
    for m := mt2 to mt1 - 1 do
      if kel(m) then d := d - 366
```

```
                else d := d - 365;
    dmt := d
end
```

Funkciją, nustatančią, ar metai yra keliamieji, jau sudarėme 6.2. skyrelyje. Ją ir perrašome:

```
function kel (m: integer): boolean;
begin
  if m < 1583
    then kel := m mod 4 = 0
    else kel := (m mod 400 = 0) or
                (m mod 100 <> 0) and
                (m mod 4 = 0)
end.
```

Ši funkcija yra universalesnė negu reikėtų mūsų programai: ji tinka metams ir ankstesniems už 1583. Šią funkciją galima būtų suprastinti: palikti tik antrąją sąlyginio sakinio šaką. Tačiau jos neprastinsime. Programuotojai, vartodami savo programose kitų sudarytas funkcijas, stengiasi jų nekeisti, kad nepadarytų klaidų.

Taigi turime užbaigtą vieną programos šaką. Liko dar kita šaka - rasti, kiek dienų prabėgo nuo metų `mt` pradžios iki mėnesio `mn` dienos `dn`.

Funkcijos reikšmę galima rasti, sudėjus visų mėnesių dienas nuo metų pradžios iki mėnesio `mn` pradžios, o prie gautos sumos pridėjus `dn` dienų. Kadangi mėnesiai turi nevienodą dienų skaičių, tai teks naudoti ciklą, o jame – sąlyginį sakinį. Be to, dienų skaičius vasario mėnesį priklauso nuo to, ar metai `mt` keliamieji. Vadinasi, ir čia bus reikalinga funkcija `kel`:

```
function dmtpr (mt, mn, dn: integer): integer;
  var mėn, d: integer;
  function kel (m: integer): boolean;
           ...
begin      { dmtpr }
  d := 0;
  for mėn := 1 to mn - 1 do
    case mėn of
      1, 3, 5, 7, 8, 10, 12: d := d + 31;
      4, 6, 9, 11          : d := d + 30;
      2: if kel(mt) then     d := d + 29
                    else     d := d + 28
    end
end;
```

Surašę visas funkcijas, gauname šitokią programą:

```
program dienos;
  var mt1, mn1, dn1,        { pirmoji data }
      mt2, mn2, dn2,        { antroji data }
      d: integer;           { dienų skaičius }
  function dsk (mt1, mn1, dn1, mt2, mn2, dn2: integer): integer;
    function dmt (mt1, mt2: integer): integer;
    var m,                  { metai }
        d: integer;         { dienos }
    function kel (m: integer): boolean;
    begin
      if m < 1583 then kel :=  m mod 4 = 0
                  else kel := (m mod 400 = 0) or
                              (m mod 100 <> 0) and
                              (m mod 4 = 0)
```

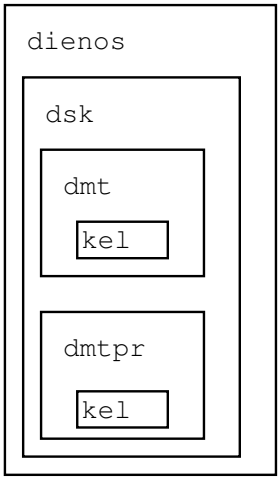
```
  end;
begin          { dmt }
  d := 0;
  for m := mt1 to mt2-1 do
    if kel(m) then d := d + 366
              else d := d + 365;
  for m := mt2 to mt1 - 1 do
    if kel(m) then d := d - 366
              else d := d - 365;
  dmt := d
end;
function dmtpr (mt, mn, dn: integer): integer;
  var mėn, d : integer;
  function kel (m: integer): boolean;
  begin
    if m < 1583
      then kel :=  m mod 4 = 0
      else kel := (m mod 400 = 0) or
                  (m mod 100 <> 0) and
                  (m mod 4 = 0)
  end;
begin    { dmtpr }
  d  := 0;
  for mėn := 1 to mn - 1 do
    case mėn of
      1, 3, 5, 7, 8, 10, 12: d := d + 31;
      4, 6, 9, 11:           d := d + 30;
      2: if kel(mt)  then    d := d + 29
                     else    d := d + 28
    end;
  dmtpr := d + dn
end;
begin    { dsk }
  dsk := dmt(mt1, mt2) - dmtpr(mt1, mn1, dn1)
                       + dmtpr(mt2, mn2, dn2)
end;
begin { dienos }
  read(mt1, mn1, dn1);
  read(mt2, mn2, dn2);
  d := dsk (mt1, mn1, dn1, mt2, mn2, dn2);
  write(mt1, '.', mn1: 2, '.', dn1: 2, '   ',
        mt2, '.', mn2: 2, '.', dn2: 2, d: 6)
end.
```

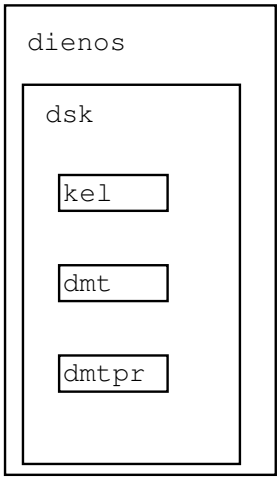
Nors programą sudarėme iš kruopščiai patikrintų funkcijų, tačiau klaidų gali atsirasti jas jungiant. Galima suklysti arba ką nors praleisti ir perrašant. Todėl programą reikia išbandyti kompiuteriu. Tikrinimui, arba kaip dažnai sakome, programos derinimui, parenkami kuo įvairesni kontroliniai pradiniai duomenys, bet tokie, kurių rezultatą nesunku ir be kompiuterio apskaičiuoti. Keli tokie duomenys ir rezultatai, kurie turėtų būti gaunami pagal tokius duomenis, pateikti lentelėje.

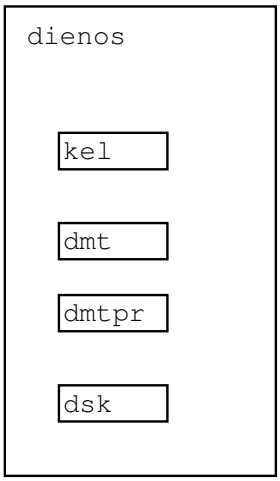
Pirmoji data	Antroji data	Dienų skaičius tarp datų
1990 01 01	1991 01 01	365
2000 01 01	2000 12 31	365
1989 05 31	1989 06 24	24
1900 02 25	1901 02 25	365
1980 02 15	1990 02 15	3653
1980 03 15	1990 03 15	3652
1989 10 20	1988 10 20	-365
1990 04 01	1990 04 01	0

Kontroliniai pradiniai duomenys buvo tik teisingos datos. Tačiau, eksploatuojant programą, gali pasitaikyti ir neteisingų pradinių duomenų – neegzistuojančių datų, pavyzdžiui, 1999 m. vasario 29 d., arba datų, nepatenkančių į uždavinio formuluotės apibrėžtą intervalą (pavyzdžiui, 1400 m. sausio 1 d.). Todėl reikėtų patikrinti, ar pradiniai duomenys teisingi. Dažnai programuotojai pradinius duomenis tikrinančias programos dalis sudaro ir įjungia į programą vėliau, po to, kai programa sudaryta ir patikrinta su teisingais pradiniais duomenimis. Mes irgi taip padarysime, atlikdami 6.3.3 uždavinį.

Sudarydami šią programą, vienas funkcijas įterpėme į kitas. Kaip jos įeina viena į kitą, pavaizduota 27 paveiksle. Funkcijos, esančios kitose funkcijose panašiai kaip ten aprašyti kintamieji, nežinomos už jų ribų. Todėl, pavyzdžiui, funkcijų `dmt` ir `dmtpr`, naudotų funkcijoje `dsk`, negalima naudoti pagrindinėje programos dalyje. Dėl to turėjome funkcijos `kel` aprašą kartoti. Jeigu funkcija `kel` būtų aprašyta tik funkcijoje `dmt`, tai ji netiktų funkcijai `dmtpr`, ir atvirkščiai. Žinoma, du kartus aprašyti tą pačią funkciją neracionalu. Vieno aprašo pakaktų, jeigu jį iškeltume į funkciją `dsk` (28 pav.).







27 pav. 28 pav. 29 pav.

Jeigu visas funkcijas aprašytume pagrindinėje programoje (29 pav.), galėtume jas naudoti visur. Tačiau funkcijas įtraukus į kitas funkcijas, savarankiškesnės būna programos dalys. Pavyzdžiui, programos pradžioje pavartodami funkciją `dsk`, dar nežinojome, ar programuojant šią funkciją prireiks kitų funkcijų. Jų ir nereikėjo. Skaidant uždavinį, pagrindinė programos dalis

nepasikeitė, neatsirado net naujų vardų. Naujos funkcijos atsispindėjo tik funkcijos `dsk` viduje. Taigi racionaliausia reikėtų laikyti 28 paveiksle parodytą programos struktūrą.

Uždaviniai

6.3.1. Duota funkcija

```
function dpr (mt, mn, dn: integer): integer;
  var mėn, d: integer;
  function kel (m: integer): boolean;
  begin
    kel := (m mod 400 = 0) or
           (m mod 100 <> 0) and
           (m mod 4 = 0)
  end;
begin       { dpr}
  d := 0;
  for mėn := 1 to mn - 1 do
    if (mėn = 2) and kel(mt) then d := d + 29
      else if (mėn = 2) and not kel(mt) then d := d + 28
        else if (mėn < 8) and (mėn mod 2 = 0)
          then d := d + 30
          else if (mėn > 8) and (mėn mod 2) <> 0
            then d := d + 30
            else d := d + 31;
  dpr := d + dn
end
```

Ar vienodos funkcijų `dpr` ir `dmtpr` reikšmės, kai parametrų reikšmės tos pačios?

6.3.2. Ką reikėtų pakeisti programoje `dienos`, kad ji tiktų visiems mūsų eros metams?

P a s t a b a: paskutinė Julijaus kalendoriaus data buvo 1582 m. spalio mėn. 4 d.; po jos ėjo pirmoji Grigaliaus kalendoriaus data – 1582 m. spalio 15 d.

6.3.3. Parašykite funkciją, patikrinančią, ar duota data nuo 1583 m. iki 3000 m. yra teisinga.

Praktikos darbas

6.3.1. Klasės mokinių gimimo datos. Sudarykite programą, kurios pradiniai duomenys būtų klasės mokinių gimimo datos, o rezultatas – visų jų pragyventų dienų bendras skaičius šiandien ir kiekvieną kitą dieną per mėnesį į priekį.

6.4. Procedūros

Funkcija gali turėti daug pradinių duomenų (parametrų) ir tik vieną rezultatą – funkcijos reikšmę. Tačiau dažnai patogu išskirti savarankiškas programos dalis, kurių rezultatai yra kelios reikšmės. Tuo tikslu vartojama kita kalbos konstrukcija – *procedūra*.

Išnagrinėkime pavyzdį. Pradiniai duomenys – du sveikieji skaičiai *a* ir *b*. Rezultatai – taip pat du sveikieji skaičiai *x* ir *y*. Pirmasis rezultatas *x* turi būti lygus mažesniam iš skaičių *a* ir *b*, antrasis – *y* – didesniam. Taigi reikia dviejų rezultatų, o jiems gauti – dviejų funkcijų, pavyzdžiui, `min` ir `max`, ir užrašyti šitokius sakinius:

```
x := min(a, b);
y := max(a, b)
```

Procedūrose ir pradiniai duomenys, ir rezultatai perduodami parametrais. Parametrų gali būti kiek reikia. Vadinasi, vartojant vieną procedūrą, galima gauti daug rezultatų.

1 pavyzdys. Sudarysime procedūrą `minmax` anksčiau minėtam uždaviniui spręsti ir įtrauksime ją į programą.

```
program procpvz;
  var a, b, c, d, x, y, z: integer;
  procedure minmax (aa, bb: integer;
                    var  xx, yy: integer);
  begin
    if aa > bb
      then
        begin xx := bb;
              yy := aa
        end
      else
        begin
          xx := aa;
          yy := bb
        end
     end;          { procedūros minmax pabaiga }
begin               { programos pradžia }
  minmax(3, 4, x, y);
  writeln(x, y);
  minmax(6, 5, x, y);
  writeln(x, y);
  read(a, b, c, d);
  minmax(a, b, x, y);
  writeln(x, y);
  minmax(c, d, x, z);
  writeln(x, z)
end.
```

Pirmoji procedūros aprašo eilutė – procedūros antraštė. Po žodžio **procedure** rašomas procedūros vardas, po jo skliaustuose išvardijami formalieji parametrai.

Procedūra `minmax` turi 4 parametrus: `aa`, `bb`, `xx` ir `yy`. Pirmieji du (`aa`, `bb`) yra pradiniai duomenys. Jie aprašomi taip, kaip ir funkcijų parametrai. Kreipiantis į procedūrą, tiems parametrams priskiriamos faktinių parametrų, nurodytų kreipinyje, reikšmės. Kiti du parametrai (`xx` ir `yy`) skirti procedūros rezultatams. Prieš juos rašomas žodis **var**, pabrėžiantis, kad kreipinyje į procedūrą šiuos parametrus atitinkantys faktiniai parametrai turi būti kintamieji (ne konstantos ir ne reiškiniai).

Po procedūros antrašte vienu sudėtiniu sakiniu užrašomi jos veiksmai (procedūros programa). Tose programos vietose, kur reikia atlikti procedūros veiksmus (procedūrą), rašomas kreipinys. Kiekvienas kreipinys į procedūrą yra atskiras sakinys. Toje programos vietoje, kur parašytas kreipinys, atliekama procedūros programa.

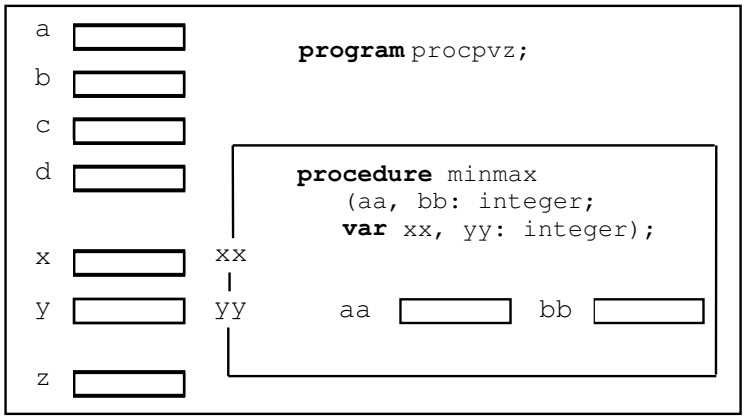
Programoje `procpvz` pirmasis kreipinys į procedūrą `minmax` yra

`minmax(3, 4, x, y).`

Vadinasi, šioje programos vietoje bus pirmą kartą atliekama procedūra `minmax`. Jos pradiniai duomenys yra skaičiai 3 ir 4. Rezultatai yra kintamųjų `x` ir `y` reikšmės.

Rezultatams skirti parametrai vadinami parametrais-kintamaisiais. Prieš juos procedūros antraštėje rašomas žodis **var**. Procedūroje `minmax` tokie parametrai yra `xx` ir `yy`. Jiems neskiriama vieta atmintyje, jų reikšmės saugomos ten, kur ir juos atitinkančių faktinių parametrų (kintamųjų) reikšmės (30 pav.). Vadinasi, tas pats atminties laukas, kuris buvo paskirtas kintamajam (faktiniam

parametrui), kreipimosi į procedūrą momentu tarsi įgyja antrą vardą. Taigi procedūroje veiksmai faktiškai atliekami su kreipinyje įrašytais kintamaisiais.



30 pav.

Grįžkime prie pavyzdžio. Kreipiniu

```
minmax(3, 4, x, y)
```

procedūros `minmax` parametrams `aa` ir `bb` bus priskirtos konstantų 3 ir 4 reikšmės (žr. 30 pav.). Atliekant procedūrą, kintamieji `x` ir `y` įgyja naujus vardus `xx` ir `yy`, parašytus procedūroje. Taigi visi veiksmai, kurie nurodyti procedūroje su kintamaisiais `xx` ir `yy`, bus faktiškai atliekami su kintamaisiais `x` ir `y`, parašytais kreipinyje ir esančiais pagrindinėje programos dalyje. Priskiriant arba kitaip keičiant šių kintamųjų reikšmes, rezultatas perduodamas iš procedūros į pagrindinę programos dalį.

Atlikus pirmąjį kreipinį į procedūrą `minmax`, kintamųjų `x` ir `y` reikšmės bus 3 ir 4. Kreipiantis į procedūrą antrą kartą (`minmax(6, 5, x, y)`), visi procedūros `minmax` veiksmai bus atliekami iš naujo ir gaunami rezultatai $x = 5$ ir $y = 6$. Kiti du kreipiniai atliekami su pradiniais duomenimis, kurių reikšmės rašant programą nežinomos.

Kai pradinių duomenų reikšmės $a = 15$, $b = 14$, $c = 13$ ir $d = 12$, kompiuteris pagal programą `procpvz` išspausdina šitokius rezultatus:

```
 3   4
 5   6
14  15
12  13
```

2 pavyzdys. Sudarysime procedūrą skaičių nuo 1 iki *n* kvadratų ir kubų sumai rasti.

```
procedure kvkub (n: integer; var kv, kub: integer);
  var k: integer;
begin
  kv := 0; kub := 0;
  for k := 1 to n do
    begin
      kv := kv + k*k;
      kub := kub + k*k*k
    end
end
```

3 pavyzdys. Sudarysime procedūrą didžiausiam *mx* ir mažiausiam *mn* skaičiui iš trijų duotųjų skaičių *a*, *b* ir *c* rasti. Procedūroje vartosime 6.2. skyr. aprašytas funkcijas `min` ir `max`.

```
procedure xxx (a, b, c: integer; var  mx, mn: integer);
begin
  mx := max(max(a, b), c);
  mn := min(min(a, b), c)
end;
```

Uždaviniai

6.4.1. Duotas procedūros aprašas:

```
procedure p (a: integer; var x: integer);
begin x := 2*a end
```

Prieš kreipiantis į procedūrą, buvo atlikti tokie programos sakiniai:

```
a := 10;  b:= 15
```

Nustatykite, kurie iš išvardytų kreipinių netaisyklingi ir kodėl. Jeigu kreipinys taisyklingas, nurodykite rezultatą, gautą atlikus procedūrą.

a) `p(a);`

b) `p(5, a, b);`

c) `p(b, b);`

d) `p(a, b);`

e) `p(7, b);`

f) `p(a+1, b);`

g) `p(a, 12);`

h) `p(a, b+1);`

i) `p(25, a);`

j) `p(true, b).`

6.4.2. Pradiniai duomenys – sveikieji teigiami skaičiai m ir n ($m < n$). Užrašykite sakinius skaičių nuo m iki n kvadratų sumai x ir kubų sumai y rasti. Sumoms skaičiuoti vartokite procedūrą `kvkub`.

6.4.3. Pradiniai duomenys – trys skaičiai (a, b ir c), reiškiantys trikampio kraštinių ilgius, surašyti nemažėjančia eile (t. y. $a \le b \le c$). Sudarykite procedūrą, kurios du rezultatai (juos žymėkite `stat` ir `lyg`) būtų loginė reikšmė: 1) `stat = true`, jei trikampis statusis, ir `stat = false` – priešingu atveju; 2) `lyg = true`, jei trikampis lygiašonis, ir `lyg = false` – priešingu atveju.

6.5. Pradinių duomenų ir rezultatų perdavimas tais pačiais parametrais

Jau sakėme, kad procedūros parametrų-kintamųjų reikšmės saugomos ten pat, kur ir juos atitinkančių procedūros kreipinyje faktinių parametrų-kintamųjų reikšmės. Jeigu, prieš kreipiantis į procedūrą, tokiems faktiniams parametrams jau buvo priskirtos reikšmės, tai jos procedūroje gali būti panaudotos kaip pradiniai duomenys.

1 pavyzdys. Sudarysime labai paprastą procedūrą jos parametro reikšmei padidinti vienetu. Procedūrą pavadinkime vardu `padid` ir įjungsime į programą `pr`.

```
program pr;
  var a: integer;
  procedure padid (var x: integer);
  begin
    x := x + 1
```

```
  end;
begin
  a := 5;
  padid(a);
  write(a);
  padid(a);
  writeln(a: 6)
end.
```

Procedūroje `padid` parametras `x` kartu yra ir pradinis duomuo, ir rezultatas. Nesunku įsitikinti, kad kompiuteris pagal tokią programą išspausdins šiuos rezultatus:

```
6    7
```

Pateikiame dar procedūrų pavyzdžių.

2 pavyzdys. Sudarysime procedūrą dviejų kintamųjų reikšmėms sukeisti vietomis ir ją įtrauksime į programą.

```
program tvarka;
  var x, y, z: integer;
  procedure keisti (var a, b: integer);
    var t: integer;
  begin
    t := a;
    a := b;
    b := t
  end;
begin            { programos pradžia }
  read(x, y, z);
  if x > y then keisti(x, y);
  if y > z then keisti(y, z);
                       { sukeitus y ir z vietomis }
                       { gali vėl tekti keisti vietomis x ir y }
  if x > y then keisti(x, y);
  writeln(x, y:5, z:5)
end.
```

Pagal šią programą gauti rezultatai bus tie patys pradiniai duomenys, parašyti nuo mažiausio iki didžiausio. Pavyzdžiui,

```
22   44   33
```

kompiuteris rašo

```
22    33    44
```

Tokie patys rezultatai būtų rašomi, jei pradiniai duomenys būtų šie:

```
44    33    22
```
arba
```
44    22    33
```
ir t. t.

3 pavyzdys. Sudarysime procedūrą duotajam natūraliajam skaičiui suapvalinti dešimčių tikslumu:

```
procedure apvalus (var x: integer);
begin
  x := (x + 5) div 10 * 10
end;
```

Jeigu procedūros parametras yra byla, tai jis turi būti persiunčiamas kintamuoju. Mat bylos būsena keičiama visada, netgi tuo atveju, kai byla skaitoma: po kiekvieno skaitymo veiksmo pasikeičia žymeklio, rodančio skaitymo vietą, padėtis.

4 pavyzdys. Sudarysime procedūrą didžiausiam skaičiui byloje rasti.

```
procedure maksimumas (var skbyla: text; var maks: integer);
```

```
                          { skbyla – skaičių byla }
                          { maks – didžiausias skaičius byloje }
  var sk: integer;        { perskaitytas skaičius }
begin
  reset(skbyla);
  maks := -maxint-1;          { tikrai nedidesnis už bet kurį skaičių }
                              { dažniausiai mažiausias skaičius kompiuteryje }
                              { yra vienetu mažesnis už -maxint }
  while not eof(skbyla) do
    begin
      read(skbyla, sk);
      if sk > maks            { rastas skaičius, didesnis už maks }
        then maks := sk
    end
end;
```

Procedūroje nerašėme sakinio `assign`. Procedūra turi būti kuo mažiau priklausoma nuo įvairių detalių, tarp jų ir bylų diske.

Procedūra `maksimumas` duoda tik vieną rezultatą. Todėl algoritmą būtų galima išreikšti ir funkcija. Tačiau taip nedarėme dėl to, kad ši procedūra duoda ne tik pagrindinį rezultatą – randa didžiausią skaičių byloje, bet ir keičia bylos būseną: kai skaito skaičius keičia žymeklio, rodančio skaitomą duomenį, vietą. Procedūrai tokie veiksmai priimtini (iš jos galima tikėtis kelių rezultatų). Tuo tarpu iš funkcijos tikimasi tik vieno rezultato. Todėl geras programavimo stilius reikalauja, jog funkcija nekeistų jokių kintamųjų reikšmių, esančių už funkcijos ribų.

Sakoma, kad funkcijos, keičiančios kokių nors kintamųjų reikšmes už jų ribų, duoda šalutinį efektą. Apie šalutinį efektą galima paskaityti straipsnyje [8].

Išnagrinėję pateiktus pavyzdžius, galime aiškiai suvokti, kuo skiriasi parametrai-kintamieji (t. y. tie parametrai, prieš kuriuos procedūros antraštėje yra žodis **`var`**) nuo parametrų-reikšmių. Parametrais-kintamaisiais duomenis galima perduoti iš kreipinio į procedūrą ir atvirkščiai, t. y. perduoti pradinius duomenis procedūrai ir gauti rezultatus iš procedūros, o parametru-reikšme perduodami duomenys tik iš kreipinio į procedūrą. Taigi parametrai-kintamieji yra universalesni už parametrus-reikšmes. Tačiau tai nereiškia, kad visada galima naudoti vien parametrus-kintamuosius. Tais atvejais, kai parametrus perduodami vien pradiniai duomenys, patogiau naudoti parametrą-reikšmę dėl šių priežasčių:

1. Formalųjį parametrą-reikšmę atitinkantis faktinis parametras gali būti ne tik kintamasis, bet ir konstanta arba reiškinys.

2. Parametrų-reikšmių savybės yra tokios pat, kaip ir kitų procedūros programoje aprašytų kintamųjų, tiktai prieš procedūros veiksmus pradžioje jiems priskiriamos pradinės reikšmės – kreipinyje nurodytų faktinių parametrų (kintamųjų, konstantų, reiškinių) reikšmės. Niekas daugiau faktinių ir formaliųjų parametrų nesieja. Vadinasi, procedūra nepakeis kreipinyje parašytų kintamųjų reikšmių. Tai ypač svarbu, kai tą pačią procedūrą vartoja daug programuotojų.

Teoriškai ir funkcija gali turėti parametrų-kintamųjų. Tačiau, vartojant juos funkcijose, suprastėja programavimo stilius, nes funkcijos rezultatas turi būti vienas ir perduodamas funkcijos vardu.

Uždaviniai

6.5.1. Ką išspausdintų kompiuteris, atlikęs programą `apvalus` (žr. 3 pavyzdį), jeigu ciklo viduje esantį sakinį `read(a)` perkeltume į sudėtinio sakinio pradžią, o pradiniai duomenys būtų tie patys?

6.5.2. Duota procedūra:

```
procedure apv (var x: integer; y: integer);
  var p, n: integer;
begin
  n := 1;
  for  p := 1  to  y  do
    n := n * 10;
  x := (x + 5 * (n div 10)) div n * n
end;
```

Paaiškinkite, kokį veiksmą ši procedūra atlieka ir kaip jis pakeistų kintamojo *p* reikšmę, atlikus kreipinį

```
apv(p, 2)
```

jeigu prieš tai kintamojo *p* reikšmė buvo šitokia:

a) 1988;

b) 49;

c) 1500.

6.5.3. Sudarykite procedūrą duotajam skaičiui pakelti nurodytu laipsniu.

6.5.4. Laikrodis rodo laiką, kurį išreiškia trijų kintamųjų reikšmės: valandos (12-os valandų skalėje), minutės ir sekundės. Parašykite ciklą, kuris pakeistų kintamųjų reikšmes taip, kad jos rodytų laiką po sekundės.

6.5.5. Duota programa:

```
program parametrai;
  var a, b, c: integer;
  procedure p (x, y: integer; var z: integer);
  begin
    x := 2; y := 2; z := 2;
    writeln(x, y: 5, z: 5)
  end;
  procedure r (var x, y: integer; z: integer);
  begin
    x := 3; y := 3; z := 3
  end;
begin
  a := 1; b := 1; c := 1;
  p(a, b, c);
  writeln(a, b: 5, c: 5);
  r(a, b, c);
  writeln(a, b: 5, c: 5)
end.
```

Ką išspausdins kompiuteris, atlikęs šią programą?

6.6. Algoritmų užrašymas funkcijomis ir procedūromis

Į funkcijas ir procedūras žiūrėjome kaip į programavimo konstrukcijas, palengvinančias programos skirstymą į dalis. Tačiau tai ne vienintelė funkcijų ir procedūrų paskirtis. Jos dažnai vartojamos algoritmams užrašyti. Iki šiol algoritmą tapatinome su programą ir laikėme, kad programa – tai algoritmo užrašas, artimesnis kompiuteriui. Tačiau algoritmą taip pat sėkmingai galima užrašyti ir funkcija arba procedūra. Kiekviena šių konstrukcijų turi priemones pradiniams duomenims, rezultatams, vidiniams duomenims ir veiksmams su jais aprašyti – viską, ko reikia algoritmui.

Koks skirtumas tarp programos ir funkcijos bei procedūros algoritmų užrašymo požiūriu?

Programa su išoriniu pasauliu bendrauja skaitymo ir rašymo veiksmais. Funkcija ir procedūra su išoriniu pasauliu pasikeičia informacija per parametrus. Kai kalbame apie algoritmą, pirmiausia apibrėžiame pradinius duomenis ir rezultatus. Taigi, funkcijos ir procedūros netgi artimesnės algoritmams, negu programos: čia, kaip ir algoritme, tik išvardijami pradiniai duomenys bei rezultatai ir nesirūpinama jų skaitymu bei rašymu. Dėl to algoritmus įprasta užrašyti funkcijomis ir procedūromis.

Transliatoriui (o tuo pačiu ir kompiuteriui) galima pateikti tik programą. Tad ką daryti su algoritmu, kuris apipavidalintas kaip funkcija arba procedūra?

Tokį algoritmą galima pateikti kompiuteriui, jį įdėjus į gaubiančią programą. Ar tai nepatogumas?

Iš dalies taip. Tačiau šį nepatogumą kompensuoja galimybė funkcijomis ir procedūromis abstrakčiau aprašyti algoritmą, nesigilinant, kaip pateikiami pradiniai duomenys ir kaip apipavidalinami rezultatai.

Algoritmą, užrašytą funkcija arba procedūra, kur kas lengviau įjungti į bet kurią programą, negu algoritmą, užrašytą programa. Šioje knygoje vis dažniau algoritminius uždavinių sprendimus užrašysime funkcijomis ir procedūromis.

Dauguma šioje knygoje pateiktų programų sprendė tokius paprastus uždavinius, kad jų programos neturėjo praktinės vertės. Argi verta rašyti programą dviejų skaičių aritmetiniam vidurkiui rasti arba didesniajam iš dviejų skaičių rasti. Tuo tarpu apipavidalinus kad ir labai mažą uždavinėlį funkcija arba procedūra, ją galima labai paprastai panaudoti daugelyje programos vietų ir daugelyje programų. Akivaizdūs tokių labai paprastų ir dažnai praktiškai vartojamų funkcijų pavyzdžiai gali būti integruotos funkcijos `abs` ir `sqr`. Juk paprasčiau parašyti

a := abs(b),

negu

```
if b < 0 then a := -b
         else a := b.
```

Jeigu algoritmo rezultatas yra viena (paprastoji) reikšmė, tai tokį algoritmą rekomenduojame išreikšti funkcija. Funkciją dažnai patogiau panaudoti programoje, negu procedūrą, kadangi kreipinys į funkciją yra reiškinys ir jį galima rašyti visur ten, kur galima panaudoti reikšmę tiesiogiai, be tarpinio kintamojo rezultatui perduoti.

Uždaviniai

6.6.1. Aritmetinio vidurkio skaičiavimo programą (žr. 1.1 skyr.) apipavidalinkite funkcija.

6.6.2. Palūkanų skaičiavimo programą (žr. 1.4 skyr.) apipavidalinkite funkcija. Dialogo veiksmų funkcijoje nenaudokite.

6.6.3. Palūkanų skaičiavimo programą apipavidalinkite procedūra. Rezultatai – du sveikieji skaičiai – litai ir centai.

6.6.4. Parašykite funkciją, kuri patikrintų, ar iš keturių atkarpų, kurių ilgiai duoti, galima sudaryti kvadratą arba stačiakampį. Aprašykite funkcijos duomenų tipą. Pasinaudokite 4.1.3 uždaviniu ir jo atsakymu.

7. DUOMENŲ STRUKTŪROS

Kai programoje atsiranda daug paprastųjų sakinių arba kelis sakinius reikia laikyti vienu, tuos sakinius jungiame į stambesnius, struktūrinius sakinius, dar vadinamus valdymo struktūromis. Daugelis sakinių pavirsta vienu sakiniu.

Kai programoje atsiranda daug paprastųjų duomenų (reikšmių) arba kelis duomenis reikia laikyti vienu, tuos duomenis jungiame į stambesnius, struktūrinius duomenis, dar vadinamus duomenų struktūromis. Daugelis duomenų (reikšmių) tampa vienu duomeniu (reikšme).

7.1. Įrašas

Dažnai programose pasitaiko tarpusavyje susijusių kintamųjų ir jų reikšmių. Pavyzdžiui, data apibūdinama trimis reikšmėmis – metais, mėnesiu, diena. Visus tris datos komponentus programose pavadindavome skirtingais vardais ir operuodavome kaip su atskirais komponentais. Kai operuojame su viena data, tai nepatogumų nejaučiame. Tačiau, kai toje pačioje programoje prireikia kelių datų, atsiranda kelis kartus daugiau vardų ir sunkiau susigaudyti. Iš dalies tų sunkumų išvengdavome, parinkdami vardus pagal sistemą:

```
mt1, mn1, dn1,
mt2, mn2, dn2  ir t. t.
```

Tačiau Paskalio kalboje yra numatytas kur kas geresnis būdas duomenims grupuoti. Programoje aprašomi nauji duomenų tipai, kurių reikšmės sudaromos iš kitų, jau žinomų duomenų tipų reikšmių. Tokie duomenų tipai vadinami struktūriniais.

Aprašysime duomenų tipą ir pavadinkime jį vardu `data`,

```
type data = record
              metai: 1..maxint;
              mėnuo: 1..12;
              diena: 1..31
            end
```

Tai įrašo tipas. Jo aprašas pradedamas žodžiu **record** ir baigiamas žodžiu **end**. Tarp jų išvardijamos sudedamosios įrašo dalys, vadinamos laukais. Nurodomas kiekvieno lauko vardas ir tipas. Įrašo lauko tipas gali būti įvairus. Čia aprašytą įrašą sudaro trys laukai: `metai`, `mėnuo` ir `diena`. Jų tipai yra įvairios sveikųjų skaičių atkarpos. Konstanta `maxint` žymi didžiausią sveikąjį skaičių kompiuteryje.

Aprašysime daugiau įrašo tipų:

```
type laikas = record
                val: 0..23;
                min: 0..59;
                sek: 0..59
              end;
     trupmena = record            { paprastoji trupmena }
                  s,              { skaitiklis }
                  v: integer      { vardiklis }
                end;
```

Kiekvienas čia aprašytas tipas yra naujas duomenų tipas. Vadinasi, įrašo negalima laikyti vienu duomenų tipu. Tai tipų šeima.

Aprašius duomenų tipą, toliau programoje galima naudoti tipo vardą kintamiesiems aprašyti, pavyzdžiui:

```
var šiandien, gimimodata: data
```

Taip aprašyti kintamieji `šiandien` ir `gimimodata` priklauso tipui `data`. Jų reikšmės yra sudėtinės (įrašai) – jas sudaro trys dalys (laukai).

Ten, kur reikia viso kintamojo reikšmės (įrašo), rašomas kintamojo vardas. Pavyzdžiui, viso įrašo reikšmės priskyrimą galima nurodyti vienu prieskyros sakiniu:

```
gimimodata := šiandien
```

Galima operuoti ir su atskirais įrašo laukais. Tada po kintamojo vardo rašomas taškas, o po jo – reikiamo lauko vardas, atskirtas tašku. Pavyzdžiui,

```
m := šiandien.metai;
d := šiandien.diena;
gimimodata := šiandien.diena
```

Laukus galima rašyti ir kairėje prieskyros sakinio pusėje, pavyzdžiui:

```
gimimodata.metai := 1968;
gimimodata.mėnuo := 3
```

Atlikus šiuos prieskyros sakinius, bus pakeistos dviejų kintamojo `gimimodata` laukų reikšmės, o trečiojo lauko reikšmė išliks ta pati.

Atkreipiame dėmesį į tai, kad vienas prieskyros sakinys

```
gimimodata: = šiandien
```

prilygsta trims paprastųjų reikšmių prieskyros sakiniams:

```
gimimometai.metai := šiandien.metai;
gimimodata.mėnuo := šiandien.mėnuo;
gimimodata.diena := šiandien.diena
```

Taigi, operuodami su sudėtingesnėmis reikšmėmis (įrašais), galime sutrumpinti programos tekstą.

1 pavyzdys.

```
program datos;
  type data = record
                met: 1..maxint;
                mėn: 1..12;
                dien: 1..31
              end;
   var a, b, c, ankstyva: data;
   function ankstesnė(a, b: data): boolean;
        { ankstesnė = true, jeigu a ankstesnė negu b }
   begin
    if a.met < b.met then ankstesnė := true
      else if a.met > b.met then ankstesnė := false
       else if a.mėn < b.mėn then ankstesnė := true
        else if a.mėn > b.mėn then ankstesnė := false
         else if a.dien < b.dien then ankstesnė := true
                                 else ankstesnė := false
    end;
begin
    read(a.met, a.mėn, a.dien);
    read(b.met, b.mėn, b.dien);
    read(c.met, c.mėn, c.dien);
    ankstyva := a;
    if ankstesnė (b, ankstyva)
        then  ankstyva := b;
    if ankstesnė(c, ankstyva)
        then ankstyva := c;
    write(ankstyva.met, ankstyva.mėn: 3, ankstyva.dien: 3)
end.
```

Šios programos pradiniai duomenys yra trys datos, o rezultatas – viena, pati ankstyviausia iš jų data.

Funkcija `ankstesnė` nustato, ar pirmoji data (*a*) yra ankstesnė už antrąją datą (*b*). Jos parametrai – įrašo tipo kintamieji. Jei nebūtų įrašo, reikėtų šešių parametrų.

Pastaba. Kai tipai vadinami vardais, dažnai pasitaikanti klaida yra duomenų tipo vardo vartojimas vietoj kintamojo vardo. Pavyzdžiui, galima rašyti

```
šiandien.met,
```

bet negalima rašyti

```
data.met,
```

nes `šiandien` yra kintamojo vardas, o `data` – duomenų tipo vardas.

2 pavyzdys. Iki 1971 metų D. Britanijos pinigų sistema buvo tokia: svarai sterlingų, šilingai ir pensai; svarą sterlingų sudarė 20 šilingų, šilingą – 12 pensų. Aprašysime įrašą pinigams užrašyti šia sistema ir sudarykime procedūrą dviem duotoms pinigų sumoms sudėti.

```
type pinigai = record
                 svarai: 0..maxint;
                 šilingai: 0..20;
                 pensai: 0..12
               end;
procedure sudėtis (a, b: pinigai; var c: pinigai);
  var p: integer;
begin
  p := a.pensai +
       b.pensai +
       12*(a.šilingai + b.šilingai +
       20*(a.svarai + b.svarai));
  c.pensai := p mod 12;
  p := p div 12;
  c.šilingai := p mod 20;
  c.svarai := p div 20
end;
```

3 pavyzdys. Žmogus turi iš viso 20 pirštų. Jeigu operuotume duomeniu, kurio reikšmės būtų kuris nors (rankos arba kojos) pirštas, tai reikėtų sugalvoti 20 pirštų vardų. Vartojant įrašą, galima apsieiti su mažiau vardų.

```
type galūnė = (ranka, koja);
     pusė = (kairė, dešinė);
     pirštas = (nykštys, smilius, didysis, bevardis, mažylis);
     p = record
           g: galūnė;
           p: pusė;
           pir: pirštas
         end;
```

Užrašysime funkciją, kuri patikrintų, ar dviejų duotų kintamųjų reikšmės apibūdina tą patį pirštą:

```
function taspats (a, b: p): boolean;
begin
  taspats := (a.g = b.g) and
             (a.p = b.p) and
             (a.pir = b.pir)
end;
```

Šioje programoje kojų pirštus pavadinome tokiais pat vardais, kaip ir rankų pirštus, nors kojų pirštai (išskyrus nykštį) vardų neturi.

Uždaviniai

7.1.1. Aprašykite įrašo tipus, apibūdinančius:

a) tašką koordinačių plokštumoje;

b) tiesinės lygties su dviem kintamaisiais koeficientus.

7.1.2. Laiko tarpą vaizduoja tokio tipo įrašas:

```
type laikas = record
                p: 0..maxint;     { paros }
                h: 0..23,         { valandos }
                min, s: 0..59
              end
```

Sudarykite:

a) funkciją, kuri nustatytų, ar du duoti laiko tarpai yra lygūs;

b) funkciją, kuri nustatytų, ar pirmasis duotas laiko tarpas yra trumpesnis už antrąjį;

c) procedūrą, kuri sudėtų du duotus laiko tarpus;

d) funkciją, kurios reikšmė būtų realiojo tipo skaičius, lygus duotam laiko tarpui, išreikštam paromis.

7.1.3. Aprašykite įrašą kampui, išreikštam laipsniais, minutėmis ir sekundėmis, pavaizduoti. Sudarykite funkciją šitaip pavaizduotam kampui pakeisti radianais ($360^0 = 2\pi$ radianų).

7.1.4. Iki 1920 metų Lietuvoje buvo vartojama colinė matavimo sistema (1 colis = 25,4 mm). 12 colių sudaro pėdą, o 3 pėdos – jardą.

Aprašykite įrašo tipą, tinkantį ilgiui užrašyti coline sistema.

Sudarykite procedūrą dviem ilgiams, užrašytiems coline sistema, sudėti.

Sudarykite funkciją ilgiui, užrašytam coline sistema, pakeisti metrais.

7.2. Įrašas, sudarytas iš įrašų

Įrašo laukas gali būti bet kurio tipo. Pats įrašas taip pat yra duomenų tipas. Vadinasi, įrašo laukas gali būti kitas įrašas. Pateikiame tokių pavyzdžių iš geometrijos.

Pirmiausia aprašysime tašką:

```
type taškas = record x, y: real end
```

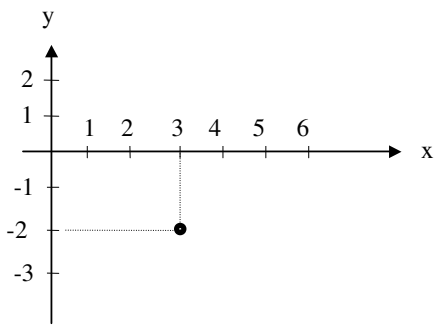
Čia *x* ir *y* yra taško koordinatės plokštumoje. Pavyzdžiui, aprašę kintamąjį

```
var a: taškas
```

ir atlikę prieskyros sakinius

```
a.x := 3.0;
a.y := -2.0
```

gausime 31 paveiksle pavaizduotą tašką.



31 pav. Taškas

Vartodami įrašą `taškas`, aprašykime naujus įrašus:

```
type atkarpa = record a, b: taškas { atkarpos galai }  end;
     apskritimas = record centras: taškas;
                          spindulys: real
                   end;
     trikampis = record a, b, c: taškas end
```

Visuose trijuose tipuose (`atkarpa`, `apskritimas` ir `trikampis`) vartojamas paprastesnis įrašas – `taškas`.

Matome, kad įrašai gali būti vienas kitame. Dėl to įrašai priskiriami prie struktūrinių duomenų tipų, arba duomenų struktūrų.

Kiekvieną iš šių tipų galima buvo aprašyti ir tiesiogiai, nevartojant tarpinio tipo `taškas`, pavyzdžiui:

```
type ap = record
            c: record
                  x, y: real
               end;
            spindulys: real
          end;
type apskr = record x, y,
               spindulys: real
             end;
```

Gali iškilti klausimas, kada naudoti tarpinį tipą ir kada didelį įrašą aprašyti tiesiogiai, iš karto. Patariame pasirinkti tą variantą, kuris yra vaizdesnis, aiškesnis. Pavyzdžiui, jeigu programoje yra aprašoma daug geometrinių figūrų, vaizduojamų taškais, tai, be abejo, tašką geriau aprašyti kaip atskirą tipą ir toliau šį tipą naudoti. Kai aprašoma tik viena figūra, tai ją sudarančias kitas figūras geriau neskirstyti į atskirus tipus. Čia galima įžvelgti analogiją tarp įrašų ir procedūrų bei funkcijų: pasikartojančius veiksmus patogu aprašyti funkcijomis ir procedūromis, o pasikartojančias duomenų grupes – duomenų struktūromis (įrašais).

Grįžkime prie apskritimo aprašo.

Aprašykime tipų `apskritimas` ir `apskr` kintamuosius:

```
var p, s: apskritimas;
      r: apskr;
      t: taškas
```

Atlikę sakinius

```
t.x := 0.0;
t.y := 2.0;
p.centras := t;
p.spindulys := 1.0;
r.x := 4.0;
r.y := 1.0;
r.spindulys := 2.0;
s := p;
s.centras.x := 6.0
```

gausime tris apskritimus.

Kintamojo *t* galima buvo ir nenaudoti. Tada pirmuosius tris prieskyros sakinius reikėtų perrašyti taip:

```
p.centras.x := 0.0;
p.centras.y := 2.0
```

Kintamieji `p` ir `s` yra to paties tipo. Todėl vieno jų reikšmę galima priskirti kitam ir atlikti sakinį `s := p`.

Kintamasis *r* yra jau kito tipo, nors taip pat reiškia apskritimą. Ar galima kintamojo *r* reikšmę priskirti kintamiesiems *p* ir *s*? Paskalio kalboje tai neapibrėžta. Todėl šiuo klausimu programuotojų nuomonės skiriasi. Tačiau logiškiau prieskyros sakinyje skirtingų tipų reikšmių nepainioti – jeigu programuotojas sugalvojo skirtingų tipų vardus, tai matyt, jie kažkuo turi skirtis. Tipus `apskritimas` ir

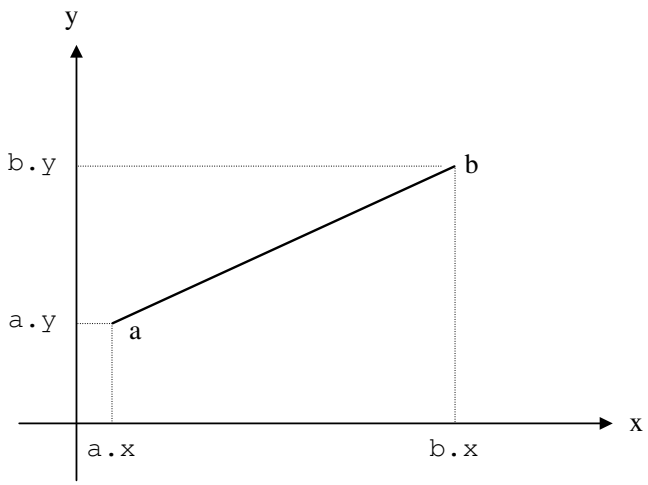
`apskr` geriau laikyti skirtingais ir nepainioti jų kintamųjų reikšmių. Vadinasi, abi prieskyros sakinio pusės turi būti to paties tipo.

Dar pateikiame funkcijų ir procedūrų pavyzdžių, kuriuose yra įrašų. Sakykime programoje aprašyti tokie įrašo tipai:

```
type taškas      = record x, y: real end;
     atkarpa     = record a, b: taškas end;
     apskritimas = record c: taškas;        { centras }
                          r: real           { spindulys }
                   end;
```

1 pavyzdys. Funkcija atstumui tarp dviejų taškų rasti (32 pav.):

```
function atstumas (a, b: taškas): real;
begin
  atstumas := sqrt(sqr(b.x - a.x) + sqr(b.y - a.y))
end;
```



32 pav.

2 pavyzdys. Funkcija atkarpos ilgiui rasti:

```
function ilgis (atk: atkarpa): real;
begin
  ilgis := atstumas(atk.a, atk.b)
end;
```

3 pavyzdys. Funkcija, patikrinanti, ar duotas taškas yra duoto apskritimo viduje:

```
function viduje (t: taškas; ap: apskritimas): boolean;
begin
  viduje := atstumas(t, ap.c) < ap.r
end;
```

4 pavyzdys. Funkcija, patikrinanti, ar kertasi du duoti apskritimai:

```
function kertasi (ap, aps: apskritimas): boolean;
begin
  kertasi := ap.r + aps.r > atstumas(ap.c, aps.c)
end;
```

Uždaviniai

7.2.1. Aprašykite taisyklingąjį daugiakampį apibūdinantį įrašą.

7.2.2. Sudarykite funkciją trikampio plotui rasti. Pradinio duomens tipas `trikampis`. Vartokite šiame skyrelyje aprašytas funkcijas.

7.3. Masyvas

Uždaviniuose dažnai būna didelės vienodo tipo duomenų grupės. Tokius duomenis patogu jungti į masyvus. Masyvas – tai duomenų struktūra, kurią sudaro daugelis vienodo tipo reikšmių.

Išnagrinėkime pavyzdį. Sakykime, reikia turėti duomenų struktūrą, nurodančią pamokų skaičių kiekvieną savaitės dieną. Tam aprašome vardinį tipą *savd*, duomenų tipą pamokų skaičiui kiekvieną savaitės dieną fiksuoti – masyvą, ir kintamąjį *p*:

```
type savd = (pirm, antr, treč, ketv, penk, šešt, sekm);
     pamokos = array [savd] of 0..6;
var p: pamokos;
```

Skirtingai nuo įrašo, masyvo komponentai nurodomi ne laukų vardais, o kito, diskrečiojo duomenų tipo reikšmėmis. Šiuo atveju masyvo komponentai – jie vadinami masyvo elementais – nurodomi tipo `savd` reikšmėmis, kurios vadinamos indeksais.

Kaip matyti iš ankstesnio aprašo, tipas `savd` turi iš viso 7 reikšmes. Vadinasi, masyvas `pamokos` turi 7 elementus. Visas masyvas nurodomas kintamojo vardu, šiuo atveju – vardu `p`. Atskiri masyvo elementai nurodomi masyvo vardu ir po jo – laužtiniuose skliaustuose parašytu indeksu. Kintamasis `p[pirm]` reiškia pamokų skaičių pirmadienį, kintamasis `p[antr]` – pamokų skaičių antradienį ir t. t.

Kokio tipo reikšmes gali įgyti masyvo elementai, rašoma po žodžio **`of`** tipo apraše. Šiuo atveju pamokų skaičius išreiškiamas atkarpos tipu `0..6`. Vadinasi, per dieną gali būti ne daugiau kaip 6 pamokos.

Masyvo elementus galima naudoti, kaip ir paprastus kintamuosius, t. y. kaip reiškinius, ir rašyti kairėje prieskyros sakinio pusėje; jie gali būti faktiniai funkcijų ir procedūrų parametrai.

Štai keli prieskyros sakiniai su kintamojo `p` komponentais:

```
p[pirm] := 5;
p[antr] := p[pirm];
p[treč] := p[pirm]+1;
p[ketv] := 4;
p[penk] := p[ketv];
p[šešt] := 3;
p[sekm] := 0
```

Atlikę šiuos sakinius, gausime kintamojo `p` reikšmę (33 pav.).

pirm	5
antr	5
treč	6
ketv	4
penk	4
šešt	3
sekm	0

33 pav.

Masyvo indeksai turi būti diskrečiojo duomenų tipo reikšmės. Su tokiomis reikšmėmis atliekamos tam tipui leidžiamos operacijos. Ši masyvo indeksų savybė ypač dažnai pritaikoma cikluose: dažniausiai masyvo indeksas įgyja visas leistinas reikšmes. Pavyzdžiui, norint rasti pamokų skaičių `s` per savaitę, reikia parašyti tokį ciklą:

```
s := 0;
```

```
for d := pirm to sekm do
  s := s + p[d]
```

Čia ciklo kintamasis `d` turi būti `savd` tipo. Atlikę šį ciklą su ankstesne kintamojo *p* reikšme, gausme `s = 27`.

Masyvo elementai dar vadinami indeksuotais kintamaisiais.

Aprašome daugiau masyvų tipų:

1. Vienos paros oro temperatūra, išmatuota kas valandą:

```
type temp = array [0..23] of -40..40;
```

2. Šimto skaičių seka:

```
type seka = array [1..100] of integer;
```

3. Kita šimto skaičių seka:

```
type kitaseka = array [2..101] of integer;
```

Masyvas `kitaseka` turi tiek pat elementų, kiek ir masyvas `seka`, tik jo elementai indeksuojami kito intervalo sveikaisiais skaičiais.

Masyvo tipo kintamuosius leidžiama rašyti abiejose prieskyros sakinio pusėse, jie gali būti funkcijų arba procedūrų parametrai.

Masyvo elemento tipas būna įvairus: loginis, vardinis, sveikasis, realusis, įrašas, masyvas ir t. t. Aprašome dar du masyvus.

1. Šešiakampis

```
type šešiakampis = array [1..6] of taškas
```

Čia `taškas` – įrašo tipas, kuris turėjo būti aprašytas prieš tai, pavyzdžiui, taip:

```
type taškas = record x, y: real end
```

Tipą `šešiakampis` galima aprašyti ir iš karto (t. y. be tipo `taškas`):

```
type šešiakampis = array [1..6] of
                     record x, y: real end
```

Aprašome tipo `šešiakampis` kintamąjį

```
var x: šešiakampis
```

Pateikiame tokiais aprašais apibūdintų kintamųjų tipus:

`x` – šešiakampis, t. y. šešių elementų masyvas, kurio kiekvienas elementas yra įrašas, sudarytas iš dviejų realaus tipo elementų;

`x[2]` – taškas, nurodantis antrąjį šešiakampio kampą, t. y. įrašas, sudarytas iš dviejų realaus tipo elementų;

`x[2].x` – taško, nurodančio antrąjį šešiakampio kampą, koordinatė *x* (realaus tipo).

2. Mėnesio kalendorius gali būti pavaizduotas šitaip:

P	2	9	16	23	30
A	3	10	17	24	31
T	4	11	18	25	
K	5	12	19	26	
P	6	13	20	27	
Š	7	14	21	28	
S	1	8	15	22	29

Aprašome masyvo tipą `kalend`, kuriuo būtų galima pavaizduoti minėtą kalendorių:

```
type savd = (pirm, antr, treč, ketv, penk, šešt, sekm);
     savnr = 1..6              { savaitės numeris }
     diena = 0..31;            { nuliu žymėsime tuščius }
                               { kalendoriaus langelius masyve }
     kalend = array [savnr] of
                array [savd] of diena
```

Aprašome tipo `kalend` kintamąjį:

```
var k: kalend;
```

Jeigu kintamojo *k* reikšmė būtų anksčiau pavaizduotas kalendorius, tai masyvas *k*[2] (jis įeina į masyvą *k*) reikštų antrosios mėnesio savaitės dienas – 2, 3, 4, 5, 6, 7, 8, masyvo elementas `k[2][pirm]` būtų lygus 2, o elementas `k[1][pirm]` – nuliui, nes pirmoji pavaizduoto mėnesio savaitė prasideda tik sekmadienį.

Masyvo elementus, nurodomus keliais indeksais, galima rašyti paprasčiau – vidurinių skliaustų] [poras pakeičiant kableliais, pavyzdžiui, vietoj *k*`[2][`*pirm*`]` – rašyti *k*`[2,` *pirm*`]`.

Pateikiame programų su masyvais pavyzdžių.

1 pavyzdys. Procedūra tipo `kalend` masyvui užpildyti:

```
procedure kal (sd: savd;          { kurią savaitės dieną prasideda mėnuo }
               dsk: diensk; { mėnesio dienų skaičius }
               var mėnuo: kalend);
  var mėnd: diena;                { mėnesio diena }
      snr: savnr;
      sdx: savd;
begin
  mėnd := 0;
  for snr := 1 to 6 do
    for sdx := pirm to sekm do
      begin
        if (snr = 1) and (sdx = sd)
          then mėnd := 1          { kalendoriaus pradžia }
          else if mėnd = dsk
            then mėnd := 0        { kalendoriaus pabaiga }
            else if mėnd <> 0 then mėnd := mėnd + 1;
        mėnuo[snr, sdx] := mėnd
      end
end
```

Kad masyvas `k` vaizduotų minėto mėnesio kalendorių, programoje reikėtų parašyti kreipinį

```
kal(sekm, 31, k)
```

2 pavyzdys. Kompiuteris skaito 100 skaičių seką ir rašo tuos pačius skaičius, sutvarkytus didėjančiai (tiksliau, nemažėjančiai):

```
program tvarkymas;
  const n = 100;
  var prad: text;                     { byla, turinti bent 100 skaičių }
      x: integer;                     { skaičius }
      m: array [1..n] of integer;     { sutvarkyti skaičiai }
      i, ind: 1..n;                   { indeksai }
      tinka: boolean;                 { tinka į m[i] vietą }
begin
  assign(prad, 'DUOMENYS.TXT'); reset(prad);
  for ind := 1 to n do
    begin
      tinka := false;
      i := ind;
      read(prad, x);
      while not tinka do
        if i = 1 then tinka := true
          else if m[i-1] <= x then tinka := true
            else begin
                   m[i-1] := m[i];
                   i := i - 1
```

```
                end;
      m[i] := x
    end;
  for i := 1 to n do
    writeln(m[i])
end.
```

Skaičiai masyve rašomi taip, kad visi jo elementai būtų sutvarkyti nuo mažiausio iki didžiausio. Jeigu naujai perskaitytas skaičius x ne mažesnis už paskutinį masyvo elementą `m[i-1]`, tai skaičius įrašomas į paskutinį laisvą masyvo elementą `m[i]`. Priešingu atveju masyvo elementai iš eilės stumiami per vieną vietą į dešinę tol, kol randama tinkama vieta naujam skaičiui `x` įrašyti (tokia, kad kairėje nuo jo neliktų skaičių, didesnių už jį, o dešinėje – visi skaičiai būtų didesni).

3 pavyzdys. Masyvo elementų rikiavimo procedūra.

```
const n = ...;
type masyvas = array [1..n] of integer;
procedure tvarka(var m: masyvas);
  var c: integer;           { masyvo elementas }
      j, k: 1..n;           { masyvo indeksai }
begin
  for k := 1 to n-1 do
    for j := 1 to n-k do
      if m[j] > m[j+1] then
        begin          { du elementai keičiami vietomis }
          c := m[j+1];
          m[j+1] := m[j];
          m[j] := c
        end
end;
```

Šis masyvo elementų rikiavimo metodas vadinamas „burbulo“ metodu. Kiekvienas masyvo elementas pamažu juda link savo vietos panašiai kaip burbulas pamažu kyla į vandens paviršių. Tai pats paprasčiausias ir pats lėčiausias rikiavimo metodas. Efektyvesni rikiavimo metodai puikiai aprašyti knygoje [9].

4 pavyzdys. Pradiniai duomenys – mokinių pažymių seka. Sekos pabaigą pažymėsime nuliu. Sudarykime programą pažymiams sumuoti ir pažymių vidurkiui rasti.

```
program pažymiai;
  var k, p: 0..5;           { pažymiai }
      suma,                 { pažymių suma }
      n: integer;           { pažymių skaičius }
      vidurkis: real;
      paž: array [1..5] of integer;
begin
  for k := 1 to 5 do
    paž[k] := 0;
          { užpildomas pažymių masyvas }
  read(p);
  while p <> 0 do
    begin
      paž[p] := paž[p]+1;
      read(p)
    end;
          { randamas vidurkis }
    n := 0; suma := 0;
    for k := 1 to 5 do
      begin
        n := n+paž[k];
```

```
        suma := suma + paž[k]*k
      end;
    vidurkis := suma/n;
    for k := 1 to 5 do
      writeln(k, ':', paž[k]: 2);
    writeln('Vidurkis ', vidurkis)
end.
```

Uždaviniai

7.3.1. Aprašykite duomenų tipą, kurio reikšmė būtų kiekvieno metų mėnesio vidutinė, žemiausia ir aukščiausia temperatūra.

7.3.2. Sudarykite dvi procedūras vieno mėnesio kalendoriui spausdinti iš tipo `kalend` masyvo šitokiu pavidalu:

a)

```
     2  9 16 23 30
     3 10 17 24 31
     4 11 18 25
     5 12 19 26
     6 13 20 27
     7 14 21 28
  1  8 15 22 29
```

b)

```
                    1
  2  3  4  5  6  7  8
  9 10 11 12 13 14 15
 16 17 18 19 20 21 22
 23 24 25 26 27 28 29
 30 31
```

7.3.3. Ką reikėtų pakeisti 2 pavyzdžio programoje `tvarkymas`, kad pradiniai duomenys būtų rašomi nedidėjimo tvarka?

7.4. Įrašo ir masyvo palyginimas

Įrašas ir masyvas – dvi duomenų struktūros, du duomenų tipai. Išsiaiškinkime, ką jie turi bendra ir kuo jie skiriasi?

Abi struktūras sudaro komponentai – kitos smulkesnės struktūros arba paprastosios (skaliarinės) reikšmės. Jų komponentų tipai gali būti įvairūs: įraše – įrašas arba masyvas, o masyve – masyvas arba įrašas. Vieną struktūrą įterpdami į kitą, galime aprašyti sudėtingiausias duomenų struktūras, panašiai kaip įterpdami vieną į kitą funkcijas arba procedūras, aprašome įvairaus sudėtingumo veiksmus.

Pagrindinis skirtumas tarp įrašo ir masyvo yra tas, kad visi masyvo komponentai turi būti to paties tipo (įrašo komponentai gali būti skirtingų tipų). Dėl to nevienodai nurodomi komponentai. Įrašo komponentai vadinami laukais, kurių vardai nurodomi įrašo apraše. Masyvo komponentus, kurie vadinami elementais, nurodo indeksai. Elementų indeksai yra diskrečiojo duomenų tipo reikšmės, o su jomis galima atlikti operacijas, pavyzdžiui, vartojant funkcijas *succ* ir *pred*, pereiti prie kitos indekso reikšmės. Tokia galimybe paranku pasinaudoti cikluose, o automatiškai perrinkti įrašo laukų negalima. Ši įrašų ir masyvo skirtybė atsiranda ne dėl kokių nors susitarimų arba komponentų žymėjimo, bet dėl to, kad masyvo elementai yra to paties tipo, o įrašo laukai gali būti skirtingų tipų. Jeigu automatiškai perrinktume įrašo laukus, tai programoje nebūtų galima užrašyti operacijų su įrašo laukais, nes nežinotume, kokio tipo gali būti laukas.

Kadangi masyvo elementai indeksuojami diskrečiojo duomenų tipo reikšmėmis, tai jų elementų gali būti labai daug. Tuo tarpu visų įrašo laukų vardus reikia išvardyti įrašo apraše. Dėl to įrašus su daugeliu komponentų nepatogu aprašyti. Kai duomenų struktūroje yra nedaug to paties tipo komponentų, juos galima aprašyti ir įrašu, ir masyvu. Pavyzdžiui, aprašykime įrašu racionaluįį skaičių (arba paprastąją trupmeną):

```
type rac = record
             skaitiklis, vardiklis: integer;
           end
```

arba tokio tipo masyvais

```
type racm1 = array [1..2] of integer
```

arba

```
type racm2 = array [(skaitiklis, vardiklis)] of integer
```

Aprašę kintamuosius

```
var x: rac;
    y: racm1;
    z: racm2
```

trupmenos skaitiklį turėtume nurodyti taip:

```
x.skaitiklis
y[1]
z[skaitiklis]
```

Pirmas ir trečias variantas yra vaizdesni. Tačiau trečiame variante vartojamas papildomas vardinis duomenų tipas, turintis dvi reikšmes.

Kurį variantą pasirinkti? Turbūt lakoniškiausias ir aiškiausias yra pirmasis.

Galima pastebėti analogiją tarp valdymo ir duomenų struktūrų.

Iš paprastųjų sakinių (pavyzdžiui, prieskyros) yra sudaromi sudėtingesni struktūriniai sakiniai – valdymo struktūros (pavyzdžiui, sudėtinis sakinys, ciklo sakinys). Iš paprastųjų duomenų tipų (pavyzdžiui, sveikųjų skaičių, loginių reikšmių) irgi sudaromi sudėtingesni duomenų tipai, kurie vadinami struktūriniais duomenų tipais, arba duomenų struktūromis.

Sudėtingą duomenų struktūrą galima išreikšti vis paprastesnėmis, kol galiausiai prieinama prie paprastųjų duomenų tipų. Pasikartojančius veiksmus galima užrašyti funkcijomis ir procedūromis, o pasikartojančias keliose programos vietose duomenų struktūras patogu aprašyti duomenų tipais.

Išnagrinėję duomenų ir valdymo struktūras, galime sėkmingai programuoti sudėtingus realaus gyvenimo uždavinius, nes jau mokame sudėtingus veiksmus skaidyti į paprastesnius (naudoti funkcijas, procedūras ir valdymo struktūras), o sudėtingas duomenų struktūras skaidyti į paprastesnes duomenų struktūras.

Uždavinys

7.4.1. Kurie iš šių teiginių yra teisingi:

a) masyvo indeksą galima apskaičiuoti;

b) įrašo lauką galima apskaičiuoti;

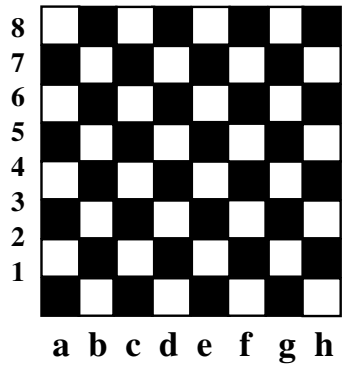
c) vietoj įrašo, kurio visi laukai yra to paties tipo, galima naudoti masyvą;

d) įrašo laukus galima įvardyti vardinio tipo konstantomis.

7.5. Duomenų struktūros sudarymo pavyzdys: šachmatų lenta

Kai gerai parinkta duomenų struktūra, lengviau užrašyti veiksmus su tais duomenimis. Todėl, dar prieš rašant veiksmus, reikia gerai apgalvoti duomenų struktūras ir parinkti pačias tinkamiausias, kuo natūraliau atspindinčias ir uždavinį, ir veiksmus programoje.

Sudarykime duomenų tipą, kurio reikšmė būtų šachmatų lenta (34 pav.). Ją patogu vaizduoti masyvu, sudarytu iš langelių. Lentos stulpelius priimta žymėti raidėmis, o eilutes skaičiais. Todėl pirmiausia aprašysime masyvo indeksų tipus:



34 pav.

```
type stulpelis = (a, b, c, d, e, f, g, h);
     eilutė = 1..8;
```

Dabar galime aprašyti ir masyvą. Galimi du variantai:

1) `type lenta = array [stulpelis] of array [eilutė] of ...`

2) `type lenta = array [eilutė] of array [stulpelis] of ...`

Jeigu priimtume pirmąjį variantą, tai masyvo elementus (langelius) reikėtų indeksuoti, pavyzdžiui, taip: [*a*, 3], jei antrąjį, – tai šitaip: [3, *a*]. Kadangi šachmatų literatūroje priimta pirma rašyti raidę, po to skaičių (pavyzdžiui, *a*3), tai natūralesnis pirmasis variantas.

Dabar reikia aprašyti masyvo elemento – lentos langelio – tipą. Ant lentos statysime šachmatų figūras. Vadinasi, langelio reikšmė bus figūra.

Šachmatų figūros tipą galima aprašyti šitaip:

```
type figūra = (karalius, valdovė, bokštas,
               rikis, žirgas, pėstininkas)
```

Betgi čia dar ne viskas. Šachmatų figūros yra baltos ir juodos, o langeliai gali būti neužimti. Todėl dar reikia aprašyti figūros spalvą:

```
type spalva = (nieko, balta, juoda)
```

Reikšmė `nieko` – tai trečioji „spalva“, reikšianti, kad langelis neužimtas.

Aprašysime masyvo elemento tipą:

```
type langelis = record
                  f: figūra;
                  s: spalva
                end
```

Dabar visus aprašus surašysime pagal Paskalio kalbos taisyklę, kuri teigia, kad naujo duomenų tipo apraše galima naudoti tik jau aprašytus duomenų tipus.

```
type figūra = (karalius, valdovė, bokštas,
               rikis, žirgas, pėstininkas);
     spalva = (nieko, balta, juoda);
     langelis = record
                  f: figūra;
                  s: spalva
                end;
```

```
stulpelis = (a, b, c, d, e, f, g, h);
eilutė = 1..8;
lenta = array [stulpelis] of
          array [eilutė] of langelis;
```

Aprašysime dvi šachmatų lentas:

```
var  x, y: lenta
```

Atskirą lentos langelį galima nurodyti kintamuoju su dviem indeksais, pavyzdžiui:

`x[a][3]` reikš langelį `a3` lentoje `x`,

`y[h][5]` reikš langelį `h5` lentoje *y*.

Baltųjų žirgo ėjimą į langelį `b4` lentoje `x` galima užrašyti taip:

```
x[b][4].f := žirgas;
x[b][4].s := balta
```

Figūros perkėlimą iš langelio `b4` į langelį `c6` (žirgo ėjimu) toje pat lentoje užrašysime šitaip:

```
x[c][6] := x[b][4];
x[b][4].s := nieko
```

Po šių veiksmų langelyje `b4` liks „bespalvė“, nieko nereiškianti figūra. Tai, žinoma, nenatūralu. Norint šito išvengti, galima būtų pritaikyti Paskalio kalbos įrašą su variantine dalimi.

Visų figūrų perkėlimą iš lentos `x` į lentą `y` galima užrašyti vienu prieskyros sakiniu:

```
y := x
```

Uždaviniai

7.5.1. Rubiko kubo sienos nudažytos 6 skirtingomis spalvomis: balta, žalia, raudona, geltona, oranžine, mėlyna. Kiekviena siena padalyta į 9 kvadratėlius.

Aprašykite duomenų tipą bet kuriam kubo spalvų deriniui užrašyti.

7.5.2. Sudarykite funkciją, patikrinančią, ar Rubiko kubas tikrai turi po 9 kiekvienos spalvos kvadratėlius. Panaudokite ankstesnio uždavinio duomenų tipus ir, jei reikia, aprašykite naujus.

7.6. Aibė

Vieno kurio nors diskrečiojo duomenų tipo reikšmių rinkinys vadinamas *aibe*. Pavyzdžiui, turime savaitės dienų tipą

```
type savd = (pirm, antr, treč, ketv, penk, šešt, sekm)
```

Pateiksime keleto šio duomenų tipo aibių.

```
[pirm];
[pirm, penkt];
[šešt, sekm];
[];
[pirm..sekm]
```

Aibės reikšmės suskliaudžiamos į laužtinius skliaustus. Šitaip aibės reikšmė atskiriama nuo paprastojo duomenų tipo reikšmės. Pavyzdžiui, `pirm` yra tipo `savd` reikšmė, o `[pirm]` yra aibės tipo reikšmė.

Aibė `[]` neturi nė vieno komponento. Ji yra tuščia.

Paskutinę pavyzdžių sąraše pateiktą aibę sudaro visos savaitės dienų tipo reikšmės.

Ištisinį į aibę patenkančių reikšmių intervalą galima užrašyti paprasčiau, nurodant tik intervalo rėžius.

Aibėje negali būti pasikartojančių komponentų.

Aibės duomenų tipo aprašas pradedamas baziniais žodžiais **set of**, po kurių nurodomas aibės komponentų tipas. Pateiksime pavyzdžių.

```
type SavDienųAibė = set of savd;
     skaitmenys   = set of 0..9;
     dviženkliai  = set of 10..99;
```

Dabar galima aprašyti šių aibių kintamuosius ir jiems priskirti aibių reikšmes.

```
var sd: savd;
    sk: skaitmenys;
dd: savd;
sk := [1, 3, 5, 7, 9];              { nelyginių skaitmenų aibė }
dd := [pirm..penk];                 { darbo dienų aibė}
```

Aibės komponentų tipas vadinamas aibės baziniu duomenų tipu. Bazinis tipas gali būti bet kuris diskretusis duomenų tipas, turintis nedidelį reikšmių skaičių. Daugumoje kompiliatorių šis skaičius yra 256.

Su aibėmis atliekamos matematikoje žinomos aibių operacijos. Tiktai jos žymimos kitaip.

Operacijos pavadinimas	Žymėjimas matematikoje	Žymėjimas Paskalyje	Pavyzdžiai
Sąjunga	$\cup$	+	`[1..5, 7] + [6, 9]` → `[1..7, 9]`
Sankirta	$\cap$	*	`[1..5, 7] * [3..9]` → `[3..5, 7]`
Skirtumas	\	-	`[1..5, 7] \ [3..9]` → `[1, 2]`
Santykio $\subseteq$	$\subseteq$	<=	`[1..5, 7] <= [3..9]` → `false`
Santykio $\supseteq$	$\supseteq$	>=	`[1..5, 7] >= [3..9]` → `false`
Priklausomybė	$\in$	**in**	`4 in [1..5, 7]` → `true`

Yra matematinių uždavinių, kuriuose operuojama su aibėmis. Tačiau programavime aibės dažniau naudojamos netiesiogiai, kaip patogi išraiškos priemonė sąlygose. Pavyzdžiui, užuot rašę

```
if (a > 10) and (a <= 20)...
```

galime rašyti

```
if a in 11..20...
```

Tokie užrašai žymiai sutrumpėja, kai reikia išrinkti daugelį nesistemingai išsibarsčiusių reikšmių.

Pavyzdys. Funkcija, skaičiuojanti balsių kiekį simbolių masyve.

```
const n = 1000;
type tekstas = array [1..n] of char;
function balsės (t: tekstas): integer;
      { laikoma, kad tekste tik didžiosios raidės
  var balsk,              { balsių skaičius }
      i: integer;
begin
  balsk := 0;
  for i := 1 to n do
    if t[i] in ['A', 'Ą', 'E', 'Ę', 'Ė', 'I',
                'Į', 'Y', 'O', 'U', 'Ų', 'Ū']
     then balsk := balsk + 1;
  balsės := balsk
end;
```

Aibės duomenų tipas dažniausiai laikomas struktūriniu. Tačiau jos komponentų tipas gali būti tik paprastasis. Taigi aibė neturi tokių struktūrizavimo galimybių, kaip masyvas arba įrašas.

Uždaviniai

7.6.1. Kurie iš šių teiginių teisingi:

a) aibės elementai gali būti realieji skaičiai;

b) aibės elementai gali būti masyvai;

c) masyvo elementai gali būti aibės;

d) visi aibės elementai turi būti vienodo tipo.

7.6.2. Kurių iš čia pateiktų aibių reikšmės yra lygios?

```
a := [1, 2, 7] + [5..10];
b := [1..10] - [3..4] + [2];
c := ([1..10] - [3..4] + [2]) * [];
d := [].
```

7.6.3. Kurie aibių reiškiniai užrašyti neteisingai ir kodėl?

a) `[9, 2] + [3];`

b) `[2, 9] + 3;`

c) `[2] in [2, 9];`

d) `2 in [2..9];`

e) `4 in [9..2].`

7.6.4. Tekstinė byla `SK.TEK` yra sudaryta iš sveikųjų skaičių. Teigiama, kad visi joje esantys skaičiai iš intervalo [0; 99] yra skirtingi. Parašykite programą, kuri patikrintų, ar iš tikrųjų taip yra ir, kaip rezultatą, parašytų vieną iš dviejų pranešimų:

```
Byloje SK.TEK visi skaičiai iš intervalo [0; 99] yra skirtingi
Byloje SK.TEK nevisi skaičiai iš intervalo [0; 99] yra skirtingi
```

8. REKURSIJA

Pasaulis nuolat atsinaujina. Vaikai atkartoja tėvus.

Rekursija – tai atkartojimas. Funkcija arba procedūra gali kreiptis į save pačią ir tuo pačiu atkartoti savo pačios veiksmus vėl nuo pradžios, tik su kitais parametrais. Toks atkartojimas vadinamas rekursija.

Rekursija yra ir paprasta, ir sudėtinga. Paprasta dėl to, kad ja dažnai pavyksta labai paprastai ir akivaizdžiai apibrėžti priklausomybes tarp dydžių. Sudėtinga dėl to, kad dažnai sunku suvokti, kaip tuos dydžius rasti, kad jie tenkintų tas priklausomybes.

8.1. Rekursinės funkcijos

Programuojant visada stengiamasi didesnį uždavinį skaidyti į mažesnius arba sudėtingesnį paversti keliais paprastesniais.

Veiksmų sudėtingumas kartais priklauso nuo duomenų. Pavyzdžiui, faktorialą galima apibrėžti šitaip:

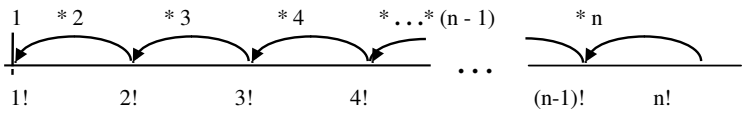
$$n! = \begin{cases} 1, & \text{jei } n = 1, \\ (n-1)! \cdot n, & \text{jei } n > 1. \end{cases}$$

Jei pradinis duomuo lygus vienetui, tai jau pats apibrėžimas pateikia rezultatą. Taigi vienetas yra pats paprasčiausias pradinis duomuo, skaičiuojant faktorialą.

Jeigu pradinis duomuo didesnis už vienetą, tai jo faktorialas išreiškiamas kita, paprastesne operacija – daugyba ir ... faktorialu, bet jau vienetu mažesnio skaičiaus. Taigi iš karto rezultato negauname. Tačiau, toliau taikydami antrąją faktorialo apibrėžimo dalį, faktorialą išreiškiame vis ilgesnės skaičių eilės sandauga:

$$\begin{aligned} n! &= \\ &(n-1)! \cdot n = \\ &\ldots = \\ &3! \cdot \ldots \cdot (n-1) \cdot n = \\ &2! \cdot 3 \cdot \ldots \cdot (n-1) \cdot n = \\ &1! \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n = \\ &1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n. \end{aligned}$$

36 paveiksle grafiškai pavaizduota, kaip einama link vieneto. Kai pasiekiame vienetą, eilėje lieka vien skaičiai ir juos galime sudauginti. Dauginame iš kairės į dešinę. Taigi dabar eile einame priešinga kryptimi – į dešinę nuo vieneto.



36 pav. Skaičiaus *n* faktorialo išreiškimas sandauga *1·2·3·...·(n-1)·n*

Jau esame sudarę faktorialo skaičiavimo algoritmą (žr. 6.2.3 užd.), kuriame skaičiaus *n* faktorialą išreiškėme skaičių nuo 1 iki *n* sandauga. Jau žinojome, kad faktorialas yra natūraliųjų

skaičių sandauga, o kompiuteriui pavedėme tik juos sudauginti, t. y. atlikti antrąją uždavinio dalį – eiti nuo 1 iki *n*. Turint funkcijas, galima kompiuteriui pavesti ir pirmąją uždavinio dalį – „išsiaiškinti“, ar faktorialas yra skaičių sandauga. Tam pakanka faktorialo apibrėžimą užrašyti programavimo kalbos žymenimis:

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

Funkcija ypatinga tuo, kad jos programoje yra kreipinys į ją pačią, kaip ir faktorialo apibrėžime – vėl faktorialas. Funkcijos ir procedūros, kuriose yra kreipinių į jas pačias, vadinamos rekursinėmis.

Pateikiame pavyzdžių.

1 pavyzdys. Kėlimą laipsniu galima apibrėžti šitaip:

$$a^n = \begin{cases} a, & \text{jei } n = 1, \\ a^{n-1} \cdot a, & \text{jei } n > 1. \end{cases}$$

Užrašome rekursinę kėlimo laipsniu funkciją:

```
function laipsnis (a, n: integer);
begin
  if n = 1 then laipsnis := a
           else laipsnis := laispnis(a, n-1)*a
end;
```

2 pavyzdys. Fibonačio skaičių seka yra tokia:

1, 1, 2, 3, 5, 8, 13, 21, ...

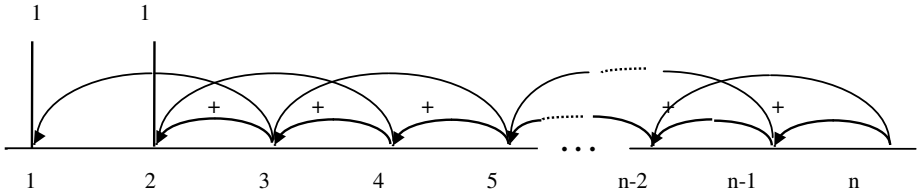
Jos pirmieji du nariai lygūs vienetui, o bet kuris tolesnis narys lygus dviejų prieš jį einančių narių sumai. Taigi *n*-tasis narys $F(n)$ apibrėžiamas šitaip:

$$F(n) = \begin{cases} 1, & \text{jei } n = 1, \\ 1, & \text{jei } n = 2, \\ F(n-1) + F(n-2), & \text{jei } n > 2. \end{cases}$$

Šį apibrėžimą išreiškiame funkcija:

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

Fibonačio sekos *n*-tajam nariui apskaičiuoti reikia dviejų prieš jį einančių narių – dviejų skaičių. Todėl reikalingi du atramos taškai – du skaičiai, kurie nurodyti apibrėžime (1 ir 2) ir į kuriuos atsiremia rodyklės 37 paveiksle.



37 pav. Fibonačio skaičių radimo schema

Kaip matome, rekursinę funkciją sudaryti labai paprasta – reikia tik matematinį jos apibrėžimą užrašyti programavimo kalbos žymenimis.

Kreipinys į funkciją iš jos vidaus išreiškia tos funkcijos veiksmų kartojimą. Tuo rekursinė funkcija panaši į ciklą. Ji, kaip ir ciklas, turi būti baigtinė.

3 pavyzdys. Duota nebaigtinė rekursinė funkcija

```
function ff (x: integer): integer;
begin
  if x = 0 then ff := 1
           else if x < 0 then ff := ff(x-1)
                         else ff := (x+1)
end;
```

Kai $x < 0$, iš funkcijos vidaus kreipiamasi į funkciją *ff* su vis mažesniu parametru; kai $x > 0$, kreipiamasi su vis didesniu parametru. Savaime suprantama, kad x niekada nebus lygus nuliui, todėl visą laiką kreipiniai į funkciją kartosis. Vadinasi, ši funkcija yra baigtinė tik tada, kai $x = 0$.

Parašius rekursinę funkciją, visada reikėtų įsitikinti, ar ji baigtinė.

Uždaviniai

8.1.1. Skaičiaus 2 n-ąjį laipsnį galima apibrėžti šitaip:

$$2^n = \begin{cases} 1, & \text{jei } n = 0, \\ 2^{n-1} \cdot 2, & \text{jei } n > 0. \end{cases}$$

Parašykite rekursinę funkciją pagal šį apibrėžimą.

8.1.2. Duota rekursinė funkcija:

```
function x (a, b: integer): integer;
begin
  if b >= a then x := 0
            else x := x(a-b, b) + 1
end;
```

Kokios šių kreipinių reikšmės:

a) `x(9, 4)`,

b) `x(9, 5)`,

c) `x(29, 2)`?

Kokią aritmetinę natūraliųjų skaičių operaciją apibrėžia ši funkcija?

8.1.3. Šiame skyrelyje pateikta funkcija *f* apibrėžta kaip natūraliųjų skaičių faktorialas. Kas atsitiktų, jeigu kreipinio $f(x)$ parametras $\boldsymbol{x}$ būtų neigiamas skaičius? Ką reikėtų pakeisti funkcijos apraše, kad $f(x)$ būtų lygus 1, kai $x \le 0$?

8.1.4. Natūraliųjų skaičių seka apibrėžiama šitaip:

$a_1 = 1$,

$a_{2n} = a_n$,

$a_{2n+1} = a_n + a_{n+1}$.

Parašykite rekursinę funkciją n-tajam sekos nariui a_n rasti.

8.1.5. Didėjančios aritmetinės progresijos pirmasis narys yra neigiamas, paskutinis narys yra n, o skirtumas d. Reikia rasti šios progresijos teigiamų narių kvadratų sumą. Parašykite dvi funkcijas šiam uždaviniui spręsti: rekursinę ir ne rekursinę.

8.1.6. Vadinamasis dvigubas faktorialas apibrėžiamas šitaip:

f!! = $1 \cdot 3 \cdot 5 \cdot ... \cdot n$, kai *n* nelyginis skaičius
$2 \cdot 4 \cdot 6 \cdot ... \cdot n$, kai *n* lyginis skaičius

Parašykite rekursinę funkciją skaičiaus *n* dvigubam faktorialui rasti.

8.2. Rekursinių funkcijų veiksmai

Rekursines funkcijas atlieka kompiuteris, o programuotojas privalo jas taisyklingai užrašyti. Todėl jis gali ir nežinoti, kokiais paprastesniais veiksmais kompiuteris jas išreiškia. Tačiau, analizuojant painesnes situacijas, ieškant klaidų programose, programuotojui kartais pravartu žinoti, kaip kompiuteris atlieka rekursines funkcijas. Apie tai pakalbėkime plačiau.

8.1. skyrelyje sudarytą faktorialo funkciją įjungsime į programą:

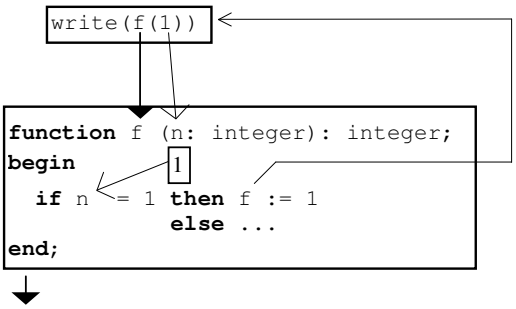
```
program rekursija;
  function f (n: integer): integer;
    begin
      if n = 1 then f := 1
               else f := n*f(n-1)
    end;
begin
  write(f(1));
  writeln;
  write(f(2));
  writeln;
  write(f(3))
end.
```

Išnagrinėsime, kaip atliekami trys programos kreipiniai į faktorialo funkciją.

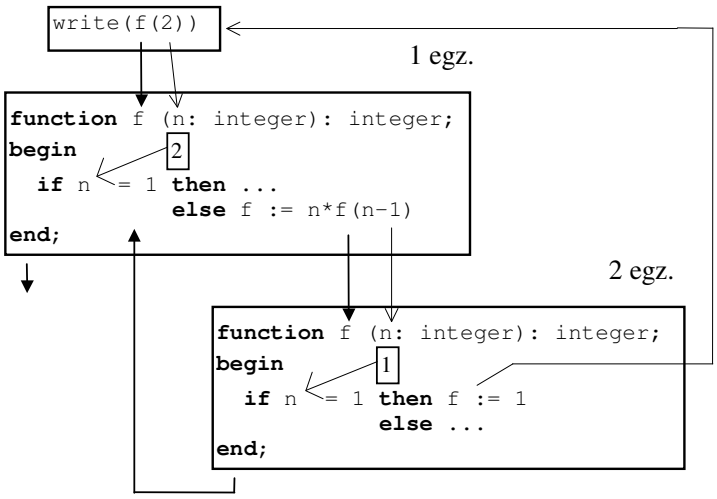
Primename, kad funkcijos kintamiesiems neskiriama vietos atmintyje ir neatliekami funkcijos veiksmai tol, kol į ją nesikreipiama.

Programoje `rekursija` pirmą kartą kreipiamasi į funkciją `f` atliekant sakinį `write(f(1))`. Tai galima įsivaizduoti taip: vietoj kreipinio `f(1)` į sakinį rašoma funkcija `f` (38 pav.), jos kintamiesiems priskiriama vieta atmintyje. Nagrinėjama funkcija `f` turi tik vieną kintamąjį – parametrą `n`. Jam paskirta atmintis 38 paveiksle pavaizduota mažyčiu stačiakampiu. Po kreipimosi į funkciją į tą atminties vietą įrašoma tikroji parametro reikšmė, šiuo atveju – skaičius 1. Toliau atliekami funkcijos veiksmai, šiuo atveju – sąlyginio sakinio šaka po žodžio **then**. Funkcijos vardui `f` priskiriama vieneto reikšmė. Pasiekus funkcijos veiksmų pabaigą rodantį žodį **end**, funkcijos reikšmė perduodama kreipiniui, o pačiai funkcijai ir jos kintamiesiems skirta vieta atmintyje panaikinama. Šiuo atveju funkcija `f` nesikreipia į save, todėl ji atliekama taip, kaip ne rekursinė.

Antrasis kreipinys į funkciją `f` yra sakinyje `write(f(2))`. Dabar $n \neq 1$, ir atliekama sąlyginio sakinio šaka po žodžio **else** (39 pav.). Čia vėl kreipinys į funkciją `f`. O jeigu yra kreipinys, tai į jo vietą įrašomas tos funkcijos aprašas, paskiriama vieta tos funkcijos kintamiesiems. Taip sukuriamas naujas, antrasis, funkcijos `f` egzempliorius, visiškai nepriklausantis nuo pirmojo (39 pav.). Antrojo funkcijos `f` egzemplioriaus faktinis parametras lygus vienetui, todėl atliekama pirmoji sąlyginio sakinio šaka. Taip apskaičiuojama antrojo egzemplioriaus funkcijos reikšmė. Ji perduodama į pirmąjį egzempliorių ir ten įrašoma vietoj kreipinio. Atmintyje panaikinamas antrasis funkcijos egzempliorius. Dabar, kai vietoj kreipinio į funkciją `f` pirmajame egzemplioriuje yra įrašyta reikšmė (vienetas), apskaičiuojama reiškinio reikšmė (2 * 1= 2), ji priskiriama pirmojo egzemplioriaus funkcijos vardui ir perduodama į pagrindinę programos dalį. Kompiuterio atmintyje panaikinamas ir pirmasis funkcijos egzempliorius.



38 pav. Funkcijos `f(1)` reikšmės skaičiavimas. Plonesnėmis linijomis (su rodyklėmis galuose) parodytas duomenų persiuntimas, storesnėmis – veiksmų eiga



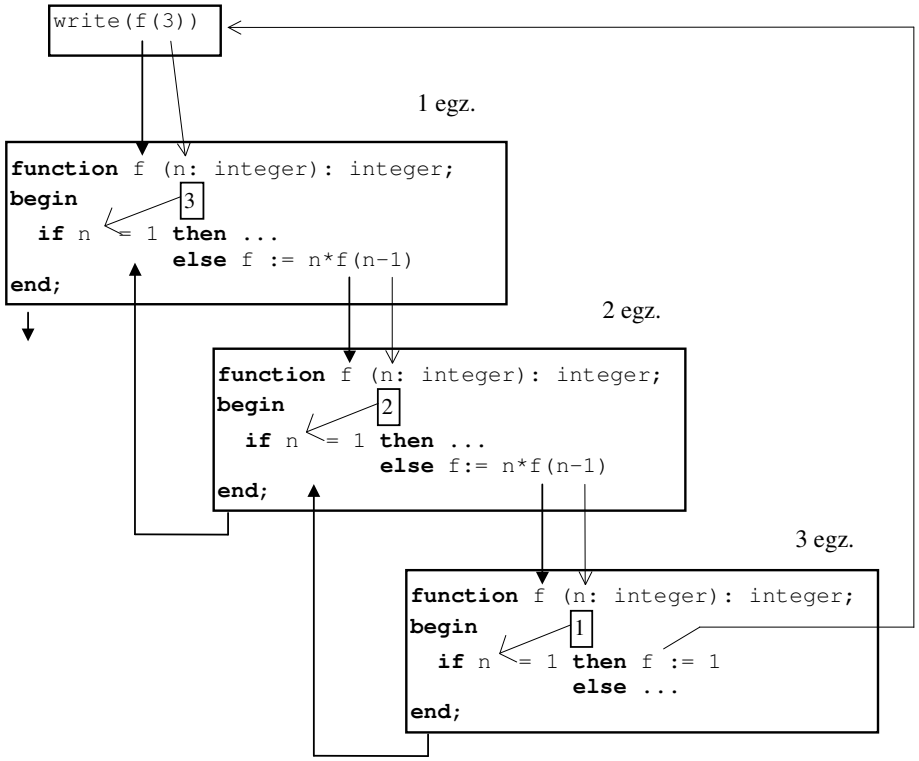
39 pav. Funkcijos `f(2)` reikšmės skaičiavimas. Sukuriami du funkcijos egzemplioriai

Atlikus paskutinį programos sakinį

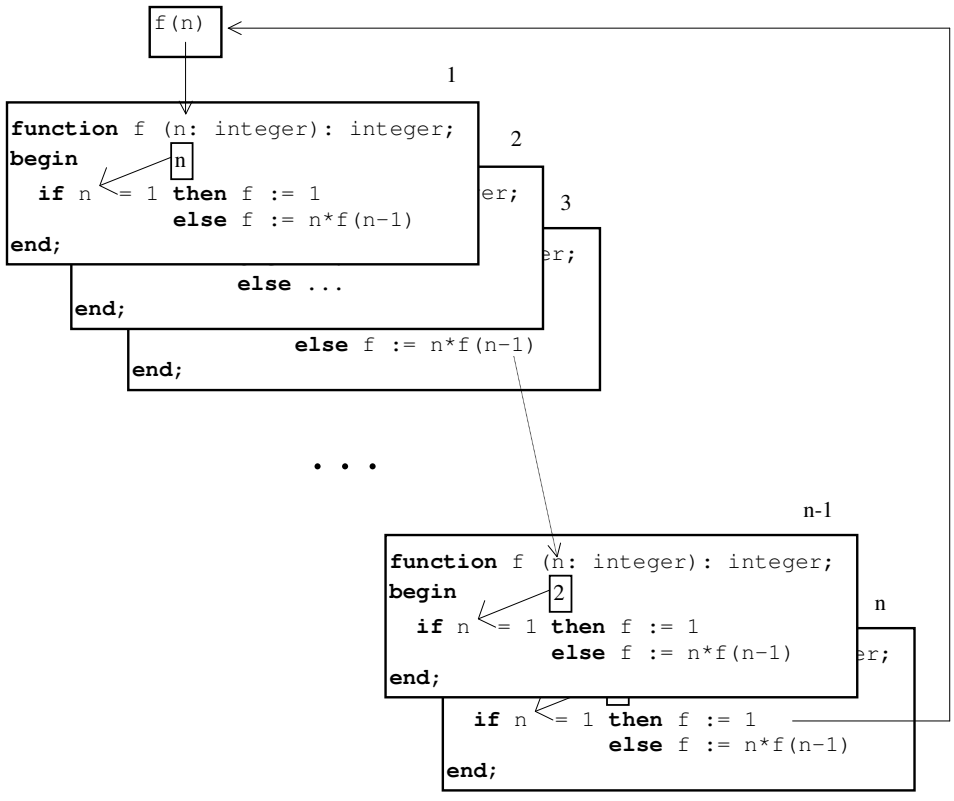
```
write(f(3))
```

iš naujo sukuriami trys funkcijos `f` egzemplioriai (40 pav.), po to funkcijos rastos reikšmės grąžinamos į jų kreipinius ir visi funkcijos egzemplioriai pašalinami iš kompiuterio atmintinės atvirkščia tvarka, negu ten jie buvo įrašyti (pirma – trečias, po to – antras ir paskui – pirmas).

Bendru atveju, kai funkcijos `f` parametras yra `n`, bus sukuriama *n* funkcijos egzempliorių (41 pav.).



40 pav. Funkcijos `f(3)` reikšmės skaičiavimas. Sukuriami trys funkcijos egzemplioriai



41 pav. Kai skaičiuojama `f(n)`, sukuriama *n* funkcijos egzempliorių

Kiekvienas funkcijos egzempliorius turi savus kintamuosius su sava kompiuterio atmintinės sritimi jų reikšmėms saugoti. Dėl to rašant rekursines funkcijas naudinga iš funkcijos iškelti visus nebūtinus kintamuosius.

Pavyzdys. Natūraliųjų skaičių seka apibrėžiama šitaip:

$a_1 = x$,

$a_2 = y$,

$a_n = a_{n-1} + a_{n-2}$.

Parašysime rekursinę funkciją n-tajam sekos nariui a_n rasti. Paprasčiausias sprendimas būtų toks:

```
function narys (n,          { ieškomo nario nr. }
                x,          { pirmasis narys }
                y: integer { antrasis narys } ): integer;
begin
  if n = 1
    then narys := x
    else if n = 2
      then narys := y
      else narys := narys(n-1, x, y) + narys(n-2, x, y)
end;
```

Seka panaši į Fibonačio seką. Skiriasi tik pirmieji du nariai. Jie yra nėra konstantos. Todėl kreipinyje į funkciją juos reikia pateikti kaip funkcijos parametrus. Tačiau toliau, einant gilyn į rekursiją, jų reikšmės nesikeičia. Todėl juos galima iškelti iš funkcijos ir naudoti kaip globaliuosius kintamuosius. Todėl reikia dviejų funkcijų. Į vieną (ne rekursinę) funkciją su visais trimis parametrais kreipiasi programa, o ši funkcija kreipiasi į kitą jos viduje esančią rekursinę funkciją, turinčią tik vieną parametrą.

```
function narys (n,          { ieškomo nario nr. }
                x,          { pirmasis narys }
                y: integer { antrasis narys } ): integer;
  function nar (n: integer): integer;
  begin
    if n = 1
      then nar := x
      else if n = 2
        then nar := y
        else nar := nar(n-1, x, y) + nar(n-2, x, y)
  end;
begin
  narys := nar(n)
end;
```

8.3. Rekursinės funkcijos pritaikymo pavyzdys: didžiausio bendro daliklio uždavinys

Norėdami tiksliai suformuluoti, kokie reikalavimai keliami funkcijai, pirmiausia užrašysime jos antraštę:

```
function dbd (x, y: integer): integer;
```

Laikykime, kad abu parametrai yra neneigiami. Pirmiausia aptarkime ypatingus atvejus: vienas iš parametrų lygus nuliui, abu parametrai lygūs nuliui. Jeigu vienas parametras lygus nuliui,

tai didžiausiu bendru dalikliu laikysime kitą parametrą, nelygų nuliui. Jeigu abu parametrai lygūs nuliui, tai bendras jų daliklis – kiekvienas nelygus nuliui skaičius. Susitarkime, kad tokiu atveju funkcijos reikšmė lygi nuliui.

Dabar jau galime aiškiai suformuluoti uždavinį: reikia sudaryti dviejų neneigiamų skaičių funkciją didžiausiam bendram dalikliui rasti. Jeigu abu skaičiai lygūs nuliui, tai funkcijos reikšmė turi būti lygi nuliui. Jeigu vienas parametrų lygus nuliui, tai funkcijos reikšmė turi būti lygi kitam nelygiam nuliui parametrui.

Taigi nulis yra „atramos“ taškas, nuo kurio galime pradėti rašyti funkcijos veiksmus:

```
if (x = 0) and (y = 0) then bdd := 0
  else if x = 0 then bdd := y
    else if y = 0 then bdd := x
      else ...
```

Funkcijos pradžią jau turime. Tačiau liko pati sunkiausia dalis, kuri turi būti po žodžio **else**.

Pritaikykime didžiausio bendro daliklio savybę: dviejų skaičių didžiausias bendras daliklis lygus mažesniojo skaičiaus ir liekanos, gautos didesnįjį skaičių padalijus iš mažesniojo, didžiausiam bendram dalikliui:

```
dbd(30, 18) = dbd(18, 30 mod 18) = dbd(18, 12).
```

Toliau remdamiesi šia taisykle, gauname:

```
dbd(18, 12) = dbd(12, 6) = dbd(6, 0).
```

Taigi skaičiai, kurių didžiausią bendrą daliklį reikia rasti, nuolat mažėja. Galų gale vienas iš jų tampa lygus nuliui. O tada jau žinome, kaip rasti bendrą daliklį.

Šiuos veiksmus užrašę programavimo kalba ir juos prijungę prie ankstesnių, gauname šitokią rekursinę funkciją:

```
function dbd (x, y: integer): integer;
                { x >= 0, y >= 0 }
begin
  if (x = 0) and (y = 0) then dbd := 0
    else if x = 0 then dbd := y
      else if y = 0 then dbd := x
        else if x >= y then dbd := dbd(x mod y, y)
                       else dbd := dbd(y mod x, x)
end;
```

Dabar pamėginkime patikrinti ir patobulinti funkciją `dbd`.

Pirmos trys sąlyginio sakinio šakos apima visus atvejus, kai bent vienas pradinis duomuo lygus nuliui. Tai faktiškai tos pačios skyrelio pradžioje nagrinėtos sąlygos, tik užrašytos Paskalio kalbos žymenimis, taigi yra teisingos.

Bandykime patobulinti parašytą programos dalį. Atidžiai išnagrinėję tris pirmąsias sąlyginio sakinio eilutes, galime jas pakeisti viena eilute:

```
if (x = 0) or (y = 0) then dbd := x + y
```

Norint įsitikinti, kad šia viena eilute išreikšti tie patys veiksmai, kaip ir trimis ankstesnėmis, reikia įrodyti, kad abiem atvejais gaunama ta pati rezultato `dbd` reikšmė, esant bet kokiems pradiniams duomenims. Įrodoma:

1. Kai `(x = 0) and (y = 0) = true`, tai pirmu atveju atliekamas sakinys `dbd := 0`. Antro atvejo sąlyga `(x = 0) or (y = 0)` tenkinama ir atliekamas sakinys `dbd := 0`. Taigi abiem atvejais gaunamas tas pats rezultatas.

2. Kai $x = 0$ ir $y \neq 0$, pirmu atveju atliekamas sakinys `dbd := y`. Antru atveju sąlyga tenkinama, ir atliekamas sakinys `dbd := x + y`. Kadangi `x = 0`, tai šis sakinys ekvivalentus sakiniui `dbd := y`. Taip matome, abiem atvejais gaunamas tas pats rezultatas.

3. Kai $y = 0$ ir ($x \neq 0$), kintamuosius x ir y sukeitę vietomis, gauname jau įrodytą antrą atvejį.

Taigi funkcijos aprašą galima sutrumpinti:

```
function dbd (x, y: integer): integer;
            { x >= 0,  y >= 0 }
begin
  if (x = 0) or (y = 0) then dbd := x + y
    else if x >= y then dbd := dbd(x mod y, y)
                   else dbd := dbd(y mod x, x)
end
```

Dabar patikrinsime dvi paskutines sąlyginio sakinio šakas. Abiejų šakų didesnysis skaičius (x arba y) dalijamas iš mažesniojo. Į šias šakas patenkama tol, kol nė viena kintamųjų x ir y reikšmė nelygi nuliui. Naujame kreipinyje į funkciją `dbd` didesnioji kintamojo reikšmė pakeičiama tų kintamųjų dalybos liekana. Teigiamųjų skaičių dalybos liekana yra visada mažesnė už dalinį ir daliklį. Taigi kintamųjų x ir y reikšmės nuolat mažės ir pagaliau kuriame nors kreipinyje viena jų pasidarys lygi nuliui. Tada bus patenkama į pirmąją šaką, o ten kreipinių nėra. Vadinasi, ši rekursinė funkcija baigtinė.

Dabar įsitikinkime, kad rezultatas teisingas.

Kai bent vienas funkcijos parametras lygus nuliui, tai atliekama pirmoji sąlyginio sakinio šaka. Rezultatas bus $x + y$, kuris, kaip jau anksčiau įrodėme, yra bendras didžiausias skaičių x ir y daliklis.

Kai nė viena kintamųjų x ir y reikšmė nelygi nuliui, atliekama antra arba trečia sąlyginio sakinio šaka. Kreipiantis į funkciją `dbd`, skaičių x ir y bendras didžiausias daliklis pakeičiamas naujų skaičių bendru didžiausiu dalikliu, kuris abiejose šakose sutampa su ankstesnių skaičių bendru didžiausiu dalikliu. Kai bent vienas skaičių virsta nuliu, patenkama į pirmąją šaką, ir gaunamas teisingas rezultatas.

Kadangi dalybos su liekana rezultatas – liekana – visada mažesnė už daliklį, tai operacijos x **mod** y rezultatas visada mažesnis už y. Todėl galime dar suprastinti funkciją:

```
function dbd (x, y: integer): integer;
begin
  if (x = 0) or (y = 0)
    then dbd := x + y
    else dbd := dbd(y mod x, x)
end
```

Laikykime, kad $x \leq y$. Tada, kreipiantis į funkciją `dbd` iš jos vidaus, pirmuoju parametru reikia rašyti `y mod x`, o antruoju `x`, nes `y mod x < x`. Kai į funkciją `dbd` kreipiamasi pirmą kartą iš programos pagrindinės dalies, gali būti netenkinama sąlyga $x \leq y$. Tada pirmą kartą funkcija atliekama beveik tuščiai – pagal ją tik sukeičiamos parametrų x ir y reikšmės vietomis. Toliau sąlyga $x \leq y$ visada bus tenkinama. Kai $x > 0$, o $y = 0$, pirmiausia taip pat bus sukeičiamos vietomis parametrų reikšmės. Todėl funkciją galima dar suprastinti:

```
function dbd (x, y: integer): integer;
begin
  if x = 0 then dbd := y
           else dbd := dbd(y mod x, x)
end
```

Įrodėme, kad paprasčiausio funkcijos varianto rezultatas teisingas. Kitus funkcijos variantus gavome iš ankstesnių, pakeitę kai kurias programos dalis trumpesnėmis, ekonomiškesnėmis, bet

ekvivalenčiomis programos dalimis. Todėl, jei teisingas pradinis funkcijos variantas, turėtų būti teisingi ir kiti jos variantai.

Kad įsitikintume, jog nepadarėme klaidų, keisdami ir perrašinėdami funkciją, reikėtų ją patikrinti su keliomis būdingiausiomis parametrų reikšmėmis:

```
dbd(0,  25);
dbd(25,  0);
dbd(30, 75);
dbd(13, 13);
dbd(75, 30);
dbd(0,   0);
```

Pamėginkite atlikti funkciją su nurodytais kreipiniais ir įsitikinsite, kad rezultatai šitokie:

```
25    25    15    13    15    0
```

Norint išbandyti funkciją kompiuteriu, reikia sudaryti ją gaubiančią programą. Kad iš karto patikrintume funkciją, esant įvairiems pradiniams duomenims, programoje reikia parašyti daug kreipinių arba vieną kreipinį cikle.

Pateikiame programą, turinčią šešias pradinių duomenų poras:

```
program bandymas;
  var a, b, j: integer;
  function dbd (x, y: integer): integer;
    begin
      if x = 0 then dbd := y
               else dbd := dbd(y mod x, x)
    end;
begin
  for j := 1 to 6 do
    begin
      read(a, b);
      writeln(a: 7, b: 7 , dbd(abs(a), abs(b)): 7)
    end
end.
```

Programos pradiniai duomenys gali būti ir neigiami. Tada randamas jų absoliutinių didumų (modulių) didžiausias bendras daliklis.

Uždaviniai

8.3.1. Parašykite rekursinę funkciją dviejų natūraliųjų skaičių didžiausiam bendram dalikliui (*dbd*) rasti, remdamiesi tokiomis *dbd* savybėmis:

$dbd(a, b) = a$, jei $a = b$;

$dbd(a, b) = dbd(a, b - a)$, jei $a < b$;

$dbd(a, b) = dbd(a - b, b)$, jei $a > b$.

8.3.2. Pradiniai duomenys – skaičių, nelygių nuliui, seka. Sekos pabaigoje – nulis. Vartodami funkciją `dbd`, sudarykite programą visos sekos skaičių didžiausiam bendram dalikliui rasti. Jeigu seką sudaro tik vienas nulis, tai jo didžiausias bendras daliklis lygus nuliui, jeigu nelygus nuliui skaičius ir nulis, – tai didžiausias bendras daliklis lygus pirmajam skaičiui.

8.3.3. Sudarykite funkciją dviejų skaičių mažiausiam bendram kartotiniam rasti. Pavartokite ją visų skaičių nuo 1 iki 10 mažiausiam bendram kartotiniam rasti.

8.3.4. Sudarykite loginę funkciją, patikrinančią, ar du duoti skaičiai yra tarpusavyje pirminiai.

8.4. Rekursinės procedūros

Rekursinės gali būti ne tik funkcijos, bet ir procedūros.

1 pavyzdys. Pradiniai duomenys – skaičių seka. Jos ilgis nežinomas. Paskutinis sekos skaičius – nulis, o visi kiti skaičiai nelygūs nuliui. Reikia sudaryti programą pradiniams duomenims rašyti atvirkščia tvarka. Pavyzdžiui, jeigu pradiniai duomenys yra skaičiai

```
3    25   115   400   13    0
```

tai kompiuteris turi rašyti

```
0    13   400   115   25    3
```

Sudarome programą su rekursine procedūra `atb`:

```
program atbulai;
  procedure atb;
    var x: integer;
  begin
    read(x);
    if x <> 0 then atb;
    writeln(x)
  end;
begin
  atb
end.
```

Procedūra `atb` yra be parametrų. Pagrindinę programos dalį sudaro vienintelis sakinys – kreipinys į procedūrą `atb`. Procedūra `atb` turi tik vieną kintamąjį `x`. Kiek kartų bus kreipiamasi iš procedūros `atb` į ją pačią, priklausys nuo pradinių duomenų sekos ilgio. Išnagrinėkime atskirus atvejus.

Kai seką sudaro tik vienas nulis (nepamirškime ir tokio atvejo!), pirmuoju sakiniu perskaitoma x reikšmė, antrasis sakinys – kreipinys į procedūrą `atb` neatliekamas (nes x = 0), trečiuoju sakiniu spausdinamas nulis. Štai ir atlikta procedūra, kartu ir visa programa.

Kai seką sudaro du skaičiai ($x_1 \neq 0$ ir $x_2 = 0$), pirmasis jų priskiriamas kintamajam x. Sąlyga $x \neq 0$ tenkinama, todėl pakartotinai kreipiamasi į procedūrą `atb`. Antrajame procedūros egzemplioriuje bus perskaitytas antrasis skaičius – nulis. Sąlyga $x \neq 0$ netenkinama, ir iš antrojo procedūros egzemplioriaus nebebus kreipiamasi į procedūrą `atb`. Trečiuoju sakiniu bus išspausdinta kintamojo x reikšmė (nulis), ir skaičiavimas grįš į pirmąjį procedūros egzempliorių. Čia bus atliekamas trečiasis sakinys `write(x)` ir rašoma pirmojo pradinio duomens reikšmė (prisiminkime, kad ji buvo priskirta pirmojo procedūros egzemplioriaus kintamajam x dar prieš kreipiantis į antrąjį egzempliorių).

Nesunku įsitikinti, kad n skaičių sekai sukuriama procedūros `atb` n egzempliorių. Spausdinimo sakinys yra po kreipinio į procedūrą, todėl skaičiai spausdinami atvirkščia tvarka negu buvo kreiptasi į procedūra (atvirkščia tvarka negu skaičius perskaitė kompiuteris).

Uždaviniai

8.4.1. Duota programa:

```
program ra;
  procedure s (n: integer);
  begin
    write(n);
    if n > 10 then s(n-1)
              else writeln;
```

```
    write(n: 4)
  end;
begin s(15) end.
```

Ką parašys kompiuteris, atlikęs programą?

8.4.2. Paaiškinkite, kaip atliekamos šitokios procedūros:

a)

```
procedure xxx (n: integer);
begin
  if n = 5 then write('*')
           else xxx(n-1)
end;
```

b)

```
procedure yyy (b: boolean);
begin
  if b then yyy(true)
end
```

8.5. Rekursija ir ciklas

Rekursinių funkcijų ir procedūrų veiksmai gali būti kartojami be ciklo. Jeigu kreipinys į funkciją arba procedūrą yra jos pačios viduje, tai reiškia, kad ten bus pakartotinai atliekami jos pačios veiksmai, tik galbūt jau su kitais duomenimis. Taigi rekursija išreiškia veiksmų kartojimą.

3.7 skyrelyje sakėme, kad bet kokiai veiksmų atlikimo tvarkai aprašyti pakanka trijų valdymo struktūrų: sakinių sekos, sąlyginio sakinio ir ciklo. Ciklą pakeitę rekursija, gauname kitą trejetą – sakinių seką, sąlyginį sakinį ir rekursiją. Šiomis konstrukcijomis (valdymo struktūromis) taip pat galima išreikšti kiekvieną veiksmų atlikimo tvarką.

Kadangi rekursija yra abstraktesnė už ciklą konstrukcija, ją labiau mėgsta teoretikai. Ciklus vertina praktikai, nes juose konkrečiau nurodoma veiksmų atlikimo tvarka. Programos su ciklais būna ekonomiškesnės negu tuos pačius veiksmus aprašančios rekursinės programos.

Teoriškai kiekvieną rekursiją galima pakeisti ciklu, ir atvirkščiai.

1 pavyzdys. Bendro didžiausio daliklio funkcijoje (žr. 8.3 skyr.) rekursiją pakeiskime ciklu.

Nagrinėjame antrąjį, šiek tiek patobulintą, bet dar negalutinį rekursinės funkcijos variantą Kadangi iš anksto neaiškus veiksmų kartojimo skaičius, tai vartosime **while** ciklą. Rekursinės funkcijos pradžioje užrašytą sąlygą

```
(x = 0) or (y = 0)
```

galima laikyti ciklo pabaigos sąlyga. Vadinasi, ciklas turi vykti tol, kol ši sąlyga netenkinama. Todėl ciklo antraštė turėtų būti šitokia:

```
while not (x = 0) or (y = 0) do
```

Pritaikę loginių reiškinių pertvarkymo taisykles (žr. 3.2 skyr.), sąlygą (tuo pačiu ciklo antraštę) suprastiname:

```
while (x <> 0) and (y <> 0) do
```

Dabar galime užrašyti ir visą funkciją:

```
function dbd (x, y: integer): integer;
begin
  while (x <> 0) and (y <> 0) do
    if x >= y then x := x mod y
              else y := y mod x;
```

```
    dbd := x + y
end;
```

Rekursinę funkciją pavyko suprastinti išbraukus ir sąlyginį sakinį, nes jame galima buvo lengvai sukeisti vietomis parametrus. Šią funkciją taip pat prastinti neverta, nes, norint kintamųjų reikšmes sukeisti vietomis, reikėtų naujo kintamojo ir daugiau prieskyros sakinių.

Rekursiją pakeisti ciklu nesunku ir kituose šiame skyriuje pateiktuose pavyzdžiuose, išskyrus paskutinį (procedūra `atb`, žr. 8.4 skyr.). Mat jame rekursija vartojama ne tik veiksmams kartoti, bet ir pradinių duomenų sekai įsiminti ir išsaugoti – kiekviename procedūros `atb` egzemplioriuje įsimenama vis naujo pradinio duomens reikšmė. O kol kas nenagrinėjome kitokių būdų didesniam duomenų kiekiui įsiminti.

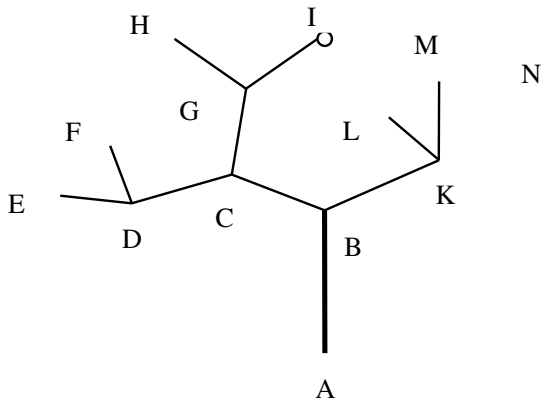
Ką tikslingiau naudoti – rekursiją ar ciklą, – priklauso nuo konkretaus uždavinio ir jo algoritmo. Universalių pasirinkimo receptų nėra. Kiekvienu konkrečiu atveju reikia naudoti vaizdžiausias ir ekonomiškiausias išraiškos priemones.

8.6. Sprendimų paieška grįžties metodu

Tik nedaugelio uždavinių sprendimus galima išreikšti formulėmis arba lygtimis. Dažnai sprendinio tenka ieškoti bandymų keliu.

Pateiksime gana universalų uždavinių sprendimo metodą, kuris dažnai vadinamas grįžimo arba bandymų ir klaidų metodu. Sistemingai bandomi visi galimi sprendimų keliai ir visada atrandamas sprendinys, jeigu tik jis egzistuoja arba, peržiūrėjus visus galimus kelius, nustatoma, kad sprendinio nėra (gali būti ir tokių atvejų).

Metodo esmę pailiustruosime „laipiojimu“ po medį (42 pav.).



42 pav. Medis, po kurį laipioja kirminas, norėdamas surasti obuolį

Tarkime, kad po medį laipioja kirminas. Nuo žemės (taško A) jis nori pasiekti obuolį, kabantį ant šakos galo I. Kol kirminas lipa kamienu iki taško B, jam viskas aišku, nes kito kelio nėra. Taške B medis šakojasi. Kurią šaką pasirinkti? Mat kirminas, būdamas viename šakos gale, dar nemato, kas yra kitame jos gale (obuolys, tuščias šakos galas ar naujas išsišakojimas). Todėl jis nežino, kurį kelią (kurią šaką) pasirinkti. Belieka išbandyti visus kelius (ištyrinėti visas šakas). Tam, kad būtų sistemingai išbandyti visi keliai, sutarkime, kad kirminas visada pasirenka kairiausią dar neišbandytą šaką. Taigi jis eina į tašką C, po to į tašką D, po to – į tašką E. Toliau nebėra kur eiti (ir obuolio nėra). Tenka grįžti atgal į tašką D. Dabar iš taško C kirminas eina jau į tašką G (kelias CD

jau išbandytas). Iš G į H – ir vėl tuščias šakos galas. Tenka grįžti į tašką H. Toliau kirminas eina į tašką I ir sėkmingai pasiekia obuolį. Uždavinys išspręstas.

Jo sprendinys – kelias ABCGI. Šis kelias buvo nutiestas palaipsniui, išbandant šitokias kelio atkarpas:

A
AB
ABC
ABCD
ABCDE
ABCD
ABCDF
ABCD
ABC
ABCG
ABCGH
ABCG
ABCGI

Atkreipiame dėmesį, kad tada, kai sužinoma, kad nueita klaidingu keliu (pasiekiamas tuščias šakos galas), uždavinį spręsti pradedama ne iš naujo, o daromas tiktai vienas žingsnis atgal ir iš čia vėl bandome eiti. Jeigu žengus žingsnį atgal pasirodo, kad ir čia nebėra naujų nenagrinėtų kelių, tai žengiamas dar vienas žingsnis atgal ir t.t., kol pasiekiamas taškas, iš kurio eina bent vienas dar neišbandytas kelias. Jeigu sugrįžtama į pradinį taškas ir iš jo nebėra naujų kelių, tai reiškia, kad išbandyti visi galimi keliai ir nė vienas nepasiekė tikslo. Tokiu atveju uždavinys sprendinio neturi.

Laipiojimas po medį yra akivaizdi sprendimo paieškos grįžties metodu iliustracija. Metodą galima taikyti daugeliui uždavinių spręsti. Tai kelio paieška labirinte, teksto perskaitymas šachmatų žirgo ėjimu, šachmatų figūrų išdėstymas lentoje taip, kad būtų tenkinamos tam tikros sąlygos ir daugelis kitų uždavinių, kurie dėl sprendimo paieškos sudėtingumo dažnai pateikiami kaip galvosūkiai.

Pavyzdys. Sudarysime šachmatų lentos apėjimo žirgu algoritmą. Šachmatų žirgo ėjimu turi būti pereinami visi lentos langeliai taip, kad kiekviename langelyje žirgas pabuvotų tik vieną kartą. Pradinė žirgo padėtis – apatinis kairysis langelis. Pirmiausia pateiksime algoritmo eskizą. Procedūros eiti veiksmus detalizuosime vėliau.

```
const n = 8;              { lentos dydis }
type koord = 1..n;        { koordinatės }
     lenta = array [koord, koord] of integer;
procedure žirgas (var len: lenta);
  var i, j: koord;
      viskas: boolean;
  procedure eiti (i: integer;     { ėjimo numeris }
                  x, y: koord;    { pradinė žirgo padėtis}
                  var len: lenta;
                      viskas: boolean); { užpildyta visa lenta }
```

Ėjimai surašomi į lentą (t.y. į masyvą `len`). Jie pradedami nuo langelio (x, y) – laikoma, kad ten jau yra žirgas. Ėjimai numeruojami pradedant skaičiumi i. Kintamojo `viskas` reikšmė `true`, jeigu sprendinys egzistuoja (t.y. po vieną kartą žirgas apsilankė visuose langeliuose) ir `false` priešingu atveju.

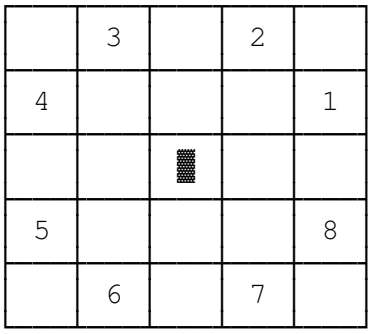
```
begin       { žirgas }
  viskas := false;
  for i := 1 to n do
```

```
    for j := 1 to n do
      len[i, j] := 0;
  len[1, 1] := 1;              { pradinė žirgo padėtis }
  eiti(2, 1, 1, len, viskas)
end;
```

Algoritme užrašyti veiksmai pakankamai aiškūs. Visa uždavinio sprendimo esmė slypi procedūroje eiti, kuri turi užpildyti ėjimų eilės numeriais visus likusius lentos langelius. Detalizuokime šią procedūrą.

Sudaryti procedūrą, kuri iš karto išspręstų visą uždavinį (t.y. užpildytų visus lentos langelius žingsnių numeriais), sunku. Todėl uždavinį suprastinsime panaudodami rekursiją. Sudarysime procedūrą, kuri darytų tik vieną žirgo ėjimą ir užpildytų tik vieną lentos langelį. Tam, kad būtų daromas naujas ėjimas, procedūra turi vėl kreiptis pati į save. Kreipiniai tęsiasi tol, kol užpildoma visa lenta arba prieinama aklavietė, t.y. lenta dar neužpildyta, o eiti nėra kur. Tada grįžtama atgal ir bandoma eiti nauju keliu.

Kai žirgas nėra lentos krašte, galimi ėjimo variantai, parodyti paveiksle. Taigi, jeigu pradinė žirgo padėtis – masyvo `len` langelis `len[x, y]`, tai bet kuri (viena iš aštuonių) nauja padėtis parodyta 43 paveiksle.



43 pav. Šachmatų žirgo ėjimai

```
len[x + cx[k], y + cy[k]]
```

Čia k - varianto eilės numeris, o masyvų `cx` ir `cy` reikšmės nustatomos iš pateikto brėžinio:

```
cx[1] :=  2;  cy[1] :=  1;
cx[2] :=  1;  cy[2] :=  2;
cx[3] := -1;  cy[3] :=  2;
cx[4] := -2;  cy[4] :=  1;
cx[5] := -2;  cy[5] := -1;
cx[6] := -1;  cy[6] := -2;
cx[7] :=  1;  cy[7] := -2;
cx[8] :=  2;  cy[8] := -1;
```

Kai žirgas yra arčiau lentos krašto, tai variantų skaičius mažesnis – reikia atmesti tuos variantus, kai bent vienas (arba abu) masyvo `len` indeksai nepatenka į atkarpą 1..n (t.y. nurodo langelį už lentos ribų).

Pateikiame procedūrą `eiti`.

```
procedure eiti (i: integer; { ėjimo numeris }
                x, y: koord;      { pradinė žirgo padėtis }
                var len: lenta;
                    viskas: boolean);   { užpildyta visa lenta }
```

```
const nn = 64;   { lentos dydis n×n }
          cx: array [1..8] of integer =
                        (2, 1. -1, -2, -2, -1, 1, 2);
          cy: array [1..8] of integer =
                        (1, 2, 2, 1, -1, -2, -2, -1);
var k: 0..8;           { varianto numeris }
    u, v: integer;     { nauja žirgo padėtis }
begin       { eiti }
  k := 0;
  repeat
    k := k + 1;
    u := x + cx[k];
    v := y + cy[k];
    if (u >= 1) and (u <= n) and (v >= 1) and  (v <= n)
       then       { naujos koordinatės dar patenka į lentą }
         if len[u, v] = 0
           then  { langelis dar tuščias }
             begin
               len[u, v] := i;
               if i < nn then
                 begin
                   eiti (i+1, u, v, len, viskas);
                   if not viskas then len[u, v] := 0
                 end
                 else viskas := true
              end
    until viskas       { užpildyta visa lenta }
       or (k = n)      { peržiūrėti visi variantai }
end;   { eiti }
```

Belieka procedūrą įdėti į programą.

Pastaba. Masyvo ir įrašo tipų konstantas turi tik Turbo Paskalis. Jeigu jų nenaudotume, vietoj konstantų `cx` ir `cy` reikėtų aprašyti tokio pat tipo kintamuosius ir programos (procedūros) pradžioje jiems priskirti pradines reikšmes. Šitokios konstantos Turbo Paskalyje vadinamos tipizuotomis konstantomis. Iš tikrųjų jos nėra konstantos, nes jų reikšmes galima keisti. Tačiau jeigu tipizuota konstanta aprašyta procedūroje (funkcijoje), tai jos rekšmė išsaugoma ir procedūrai (funkcijai) baigus darbą. Tokios rūšies duomenis turi ir kitos programavimo kalbos. Tik jie vadinami statiniais kintamaisiais.

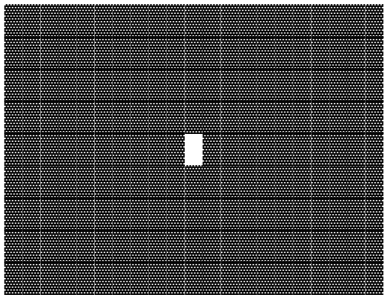
Panašiai rekursinės programos sudaromos ir kitiems uždaviniams, kuriuos sprendžiant variantai peržiūrimi tol, kol gaunamas bent vienas sprendinys, arba nustatoma, jog sprendinys neegzistuoja. Jeigu norima rasti visus galimus sprendinius, tai po pirmojo rasto sprendinio skaičiavimai nebaigiami – ieškoma kitų sprendinių tol, kol randami visi.

Abiejuose pavyzdžiuose pateikėme rekursinius algoritmus. Šios rūšies uždaviniams rekursija gerai tinka, kadangi paieškos požiūriu visi kelio taškai yra lygiaverčiai. Todėl nuėjus į naują tašką tolesnį kelią galima ieškoti vėl pagal tą patį algoritmą. Tačiau galima parašyti ir nerekursinius algoritmus – juk kiekvieną rekursinį algoritmą galima išreikšti ne rekursiniu (iteraciniu) algoritmu. Tiktai nenaudojant rekursijos tektų parašyti šiek tiek daugiau veiksmų – ir tuos, kuriuos neišreikštiniu pavidalu atlieka rekursija per procedūrų (arba funkcijų) parametrų perdavimo mechanizmą.

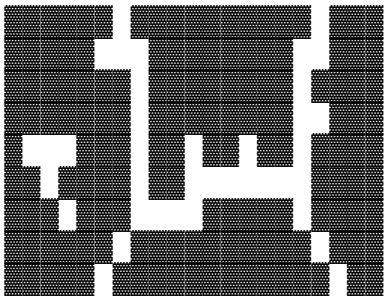
Praktikos darbas

8.6.1. Kelio paieška labirinte. Labirintas vaizduojamas n×m (n, m – nelyginiai skaičiai) langelių dydžio lenta (44 pav.) Vieni langeliai juodi, kiti – balti. Vaikščioti galima tik baltais langeliais horizontaliai arba vertikaliai. Eiti įstrižai negalima. Kelias prasideda nuo centrinio langelio. Todėl jis turi būti baltas. Reikia rasti bent vieną kelią iš centrinio langelio į lentos kraštą.

Parašykite algoritmą keliui rasti ir apipavidalinkite jį rekursine procedūra. Po to procedūrą įjunkite į programą ir išbandykite kompiuteriu.



a



b

44 pav. 9×21 langelių dydžio labirintas. Pavaizduoti du variantai: a) tuščias tik centrinis langelis (iš jo išėjimo nėra) ir b) yra daugiau tuščių langelių (iš centrinio langelio yra du išėjimai – abu į viršų)

9. TEKSTAI

Su tekstais dirbame daugiau, negu su skaičiais. Bet veiksmai su tekstais paprastesni. Todėl ir tekstinių uždavinių programavimui skiriame mažiau dėmesio – tik vieną šios knygos skyrių.

Su tekstų dorojimu susiduriame, kai reikia tvarkyti tekstus. Tai vertimo iš vienos kalbos į kitą, redagavimo, kalbos analizės uždaviniai. Programa taip pat tekstas. Todėl programos analizės bei transliavimo uždaviniuose reikia atlikti daug veiksmų su tekstais. Tai gana sudėtingos programos ir mes jų nenagrinėsime. Keletą paprastesnių programų transliavimo programų galima rasti knygose [2, 3]. Čia pateiksime tik elementarius algoritmus su tekstais.

9.1. Simboliai

Pats mažiausias ir nebedalomas teksto elementas yra simbolis. Jis priklauso simbolių tipui `char`. Simbolių tipą galima laikyti specialia vardinio duomenų tipo rūšimi ir taikyti visas vardinio duomenų tipo operacijas. Iš tikrųjų taip ir yra.

Su simboliais atliekamos tos pačios operacijos ir funkcijos, kaip ir su vardinio duomenų tipo reikšmėmis: <, <=, =, <>, >, >=, `succ, pred`. Be to, Paskalis turi integruotas funkcijos, būdingos simbolių tipui.

Funkcijos `ord(s)` argumentas `s` yra simbolis, bendru atveju – simbolių tipo reiškinys, o rezultatas yra sveikųjų skaičių atkarpos 0..255 tipo. Tai simbolio `s` kodas kompiuteryje.

Funkcija `chr(i)` atlieka atvirkščią veiksmą, negu `ord` – skaičių iš minėto intervalo paverčia simboliu.

Kadangi funkcijos `ord` ir `chr` yra atvirkštinės, tai jų parametrai ir reikšmės susijusios taip:

```
chr(ord(s)) = s,
ord(chr(i)) = i.
```

Daugelio operacijų su simboliais rezultatas priklauso nuo simbolių surikiavimo į eilę. O jo neapibrėžia nei Paskalio kalba, nei jo dialektai. Tiesiog pasakoma, kad simboliai surikiuoti pagal jų kodus kompiuteryje. Reikalaujama tik, kad skaitmenų simboliai būtų surikiuoti pagal eilę '0', '1', ... , '9' ir jų kodai eitų ištisai, t.y, tarp skaitmenų negali būti jokių kitų simbolių, išskyrus skaitmenis.

Simbolių kodai kompiuteryje priklauso nuo to, kokia (kokios valstybės) kodavimo lentelė tuo metu veikia kompiuteryje. Dėl to iš operacijų su simboliais mažai naudos.

Funkciją `ord` patogu vartoti, kai reikia skaitmens simbolį paversti vienaženkliu sveikuoju skaičiumi. Jeigu funkcijos argumento `s` reikšmė yra skaitmens simbolis, tai norint šį simbolį pakeisti skaičiumi `i`, rašome:

```
i := ord(s) - ord('0')
```

Pagal šį sakinį visada simbolis bus pakeistas jį atitinkančiu vienženkliu skaičiumi, kad ir kaip kompiuteryje būtų koduojami simboliai, nes skaitmenų simboliai eina iš eilės be tarpų visuose kompiuteriuose. Pavyzdžiui, personaliniuose kompiuteriuose naudojamoje ASCII kodų lentelėje nulio kodas yra 48, o trejeto kodas yra 51. Todėl gausime:

```
ord('3') - ord('0') ⇒ 3.
```

Dideliuose kompiuteriuose naudojamame EBCDIC kodų lentelėje nulio kodas yra 240, o trejeto kodas yra 243. Todėl ir čia minėto reiškinio reikšmė bus ta pati – skaičius 3.

Jeigu tiesiogiai į reiškinį rašytume kodą, pavyzdžiui simbolio pakeitimą vienaženkliu skaičiumi užrašytume taip:

```
i := ord(s) - 48
```

tai toks veiksmas nebūtų universalus, nes jis tiktų tik vienai (personalinių) kompiuterių klasei. Be to ir skaityti tokį užrašą būtų sunkiau, nes norint jį suvokti reikia žinoti, kad skaičius 48 reiškia nulio kodą. Dėl to simbolių kodai vartotini tik tada, kai atliekami veiksmai su kodais, pavyzdžiui perkodavimo algoritmuose.

Kitose programose, kai dirbama su tikrais (literatūriniais) tekstais, tiesioginio simbolių kodų panaudojimo reikia vengti.

1 pavyzdys. Skaitymo procedūra `read` gali skaityti realiuosius skaičius, kuriuose trupmeninė dalis nuo sveikosios skiriama tašku. Lietuvoje, o taip pat daugelyje Europos valstybių priimta trupmeninę skaičiaus dalį nuo sveikosios skirti kableliu. Parašysime procedūrą šitaip klaviatūra surinktiems skaičiams skaityti. Laikysime, kad skaičiai užrašyti vien dešimtainėmis trupmenomis, t.y. rodyklinis jų užrašas nevartojamas.

```
procedure readreal (var byla: text; var r: real);
  var s: char;              { skaitomas simbolis }
      d: real;              { trupmenos daugiklis }
      neigiamas: boolean;
begin
  r := 0; d := 1; neigiamas := false;
  read(byla, s);
  while s = ' ' do          { praleidžiami tarpai }
    read(byla, s);
  if s = '-' then begin
                    neigiamas := true;
                    read(byla, s)
                  end
             else if s = '+'
                    then read(byla, s);
  while (s in ['0'..'9']) and not eof(byla) do
    begin
      r := r*10 + (ord(s) - ord('0'));
      read(byla, s)
    end;
  if s = ',' then           { skaičius turi trupmeninę dalį }
    begin
      read(byla, s);        { praleidžiamas kablelis }
      while s in ['0'..'9'] do
        begin
          d := d*0.1;
          r := r + (ord(s)-ord('0'))*d;
          read(byla, s)
        end
    end;
  if neigiamas then r := -r
end;
```

Panašiai skaičius iš simbolių formuoja ir integruota skaitymo procedūra `read`. Apskritai, visos tekstinės bylos yra sudarytos vien iš simbolių. Tiktai skaitymo procedūros tuo atveju kai jų parametras yra skaičius iš bylos skaito visus skaičių sudarančius simbolius (iki aptinka simbolį, kuris nebepriklauso skaičiui) ir iš jį suformuoja sveikąjį arba realųjį skaičių, pagal kintamojo tipą. Deja, pastarosios galimybės neturi programuotojo sudaromos procedūros – jose reikia nurodyti parametrų skaičių ir kiekvieno jų tipą.

Simbolio skaitymas procedūra `read` yra pats paprasčiausias: perskaitomas vienas simbolis, tas, kurį rodo bylos žymeklis, žymeklis paslenka per vieną poziciją į dešinę ir ima rodyti kitą simbolį byloje. Jokių išimčių nėra, nes visi simboliai, ar tai būtų tarpas, ar apostrofas, ar bet kuris kitas simbolis. Tarpas yra lygiavertis simbolis. Apostrofas tekstinėje byloje taip pat yra apostrofas, nes bylą sudaro vien simboliai ir nėra nuo ko jų atskirti. Tuo tarpu programos tekste tik dalis simbolių naudojami kaip simbolinio tipo reikšmės. Todėl ten reikalingi specialūs simboliai, kurie juos atskirtų nuo kito programos teksto. Šiam tikslui ir yra vartojami apostrofai.

Taigi, simbolinis duomenų tipas yra pats svarbiausias duomenų saugojimo diske tipas.

Teoriškai pakaktų tokių procedūrų, kurios gali skaityti ir rašyti tik simbolio tipo duomenis, nes visų kitų tipų duomenys kompiuterio išorėje vaizduojami simboliais. Tačiau praktiškai tai būtų nepatogu, nes reikėtų detaliai programuoti duomenų skaitymo ir rašymo veiksmus. Dėl to šie veiksmai įjungti į procedūras

read ir write. Kai šių procedūrų parametrai yra kitokio tipo (negu tipo char), tai skaitomi simboliai automatiškai keičiami to tipo reikšmėmis arba to tipo reikšmės keičiamos rašomais (spausdinamais) simboliais.

Grįžkime prie simbolių kodavimo. Aiškinome, kokie pavojai slypi, kai programoje tiesiogiai naudojamos simbolių kodų reikšmės. Betgi kompiuterio atmintinėje saugomi simbolių kodai, o ne patys simboliai. Todėl gali kilti klausimas, ar tos simbolių eilutės, kurias įterpiame į rašymo sakinius, bus matomos vaizduoklio ekrane arba išspausdintos teisingai?

Kompiuteris rodys ekrane arba išspausdins lygiai tokias simbolių eilutes, kokias matome parašytas programoje. Jeigu programos tekstą norėsime perkelti į kompiuterį, kuriame naudojama kita kodavimo lentelė, tai tada programą reikės perkoduoti lygiai taip, kaip perkoduojami visi kiti perkeliami tekstai. Kai programos tekstas bus perkoduotas, naujas kompiuteris rodys teisingus simbolius programos tekste, o taip pat tos programos išspausdintuose arba ekrane rodomuose rezultatuose.

Nuo kodavimo nepriklauso lyginimo = ir <> , o taip pat priklausomybės aibei **in** operacijų rezultatai.

Uždaviniai

9.1.1. Kodėl procedūroje readreal nėra kreipinių į procedūras assign ir reset?

9.1.2. Parašykite dvi funkcijas: vieną, pakeičiančią simbolį vienaženkliu skaičiumi, kitą – vienaženklį neneigiamą skaičių simboliu. Prieš tai aprašykite dvi atkarpas: sveikųjų skaičių ir simbolių.

9.1.3. Įsivaizduokime, kad skaitymo procedūra *read* gali skaityti tik char tipo duomenis. Sudarykite procedūrą vienam sveikajam skaičiui skaityti.

Praktikos darbai

9.1.1. Realiojo skaičiaus formavimas iš simbolių. Papildykite 1 pavyzdžio procedūrą taip, kad ji galėtų skaityti ir skaičius, užrašytus rodikliniu pavidalu.

9.1.2. Realiojo skaičiaus pavertimas simbolių seka. Sudarykite realiųjų skaičių rašymo procedūrą tokią, kad trupmeninė skaičiaus dalis būtų skiriama kableliu. Procedūra turi turėti du variantus: vieną skaičiui rašyti dešimtainės trupmenos pavidalu (ji turi tris parametrus: rašomą skaičių ir du skaičius, kuriais nurodomas formatas) ir rodikliniu pavidalu (turi du parametrus – skaičių ir formatą).

9.2. Simbolių eilutės

Simbolis yra per mažas teksto elementas. Todėl vartojami didesni. Paskalio standarte stambesni teksto elementai gaunami panaudojant pakuotus simbolių masyvus (**packed array**). Turbo Paskalyje ir kituose Paskalio dialektuose vartojamos simbolių eilutės. Su simbolių eilučių kintamaisiais ir konstantomis jau esame susipažinę. Dabar panagrinėsime, eilučių tipo reiškinius. Su eilutėmis yra apibrėžta viena operacija ir keletas funkcijų. Panagrinėsime jas.

e+ee Operacijos rezultatas – eilutė, gauta sujungus jos operandus – eilutes e ir ee į vieną eilutę, pavyzdžiui,

'PROGRAMA' + 'AVIMAS' ⇒ 'PROGRAMAVIMAS'

copy(e, i, n) Funkcijos rezultatas – *n* simbolių ilgio eilutė – eilutės e dalis, kurios pradžia yra *i*-asis jos simbolis, pavyzdžiui,

copy('informacija', 3, 5) ⇒ 'forma'

length(e) Funkcijos rezultatas – eilutės e ilgis, pavyzdžiui,

length('Jonas ir Ona') ⇒ 12

pos(e, ee) Funkcijos rezultatas – simbolio eilės numeris (pozicija) eilutėje ee, nuo kurio prasideda eilutė e, pavyzdžiui,

pos('ir', 'Jonas ir Ona') ⇒ 7

Jei eilutės e nėra eilutėje ee, tai pos(e, ee) = 0.

Darbą su eilutėmis pailiustruosime pavyzdžiais.

1 pavyzdys. Mažosios raidės keitimas didžiąja.

```
function didr (s: char): char;
  const dr = 'AĄBCČDEĘĖFGHIYĮJKLMNOPQRSŠTUŲŪWVXZŽ';
        mr = 'aąbcčdeęėfghiyįjklmnopqrsštuųūwvxxž';
  var nr: integer;          { raidės numeris }
      e: string;
begin
  nr := pos(s, mr);
  if nr = 0 then e := s
            else e := copy(dr, nr, 1);
  didr := e[1]
end;
```

2 pavyzdys. Pranešimo šifravimas Cezario šifru. Šifruojama šitaip: *n*-toji abėcėlės raidė keičiama *n+2*-aja, priešpaskutinė – pirmąja, o paskutinė – antrąja. Kiti simboliai nešifruojami.

Parašysime vienos raidės šifravimo funkciją.

```
function Cezaris (s: char): char;
  const dr = 'AĄBCČDEĘĖFGHIYĮJKLMNOPQRSŠTUŲŪWVXZŽ';
        mr = 'aąbcčdeęėfghiyįjklmnopqrsštuųūwvxzž';
  var   nr: integer;        { raidės numeris }
        ss: string;         { užšifruota raidė }
begin
  ss := s;
  nr := pos(s, dr);
  if nr <> 0                              { šifruojama didžioji raidė }
    then
      begin
        if nr = length(dr)
          then ss := copy(dr, 2, 1)          { paskutinė }
          else if nr = length(dr)-1
                 then r := copy(dr, 1, 1)    { priešpaskutinė }
                 else r := copy(dr, nr+2, 1)  { n+2 }
      end;
  nr := pos(s, mr);
  if nr <> 0                              { šifruojama mažoji raidė }
    then
      begin
        if nr = length(mr)
          then ss := copy(mr, 2, 1)          { paskutinė }
          else if nr = length(mr)-1
                 then ss := copy(mr, 1, 1){ priešpaskutinė }
                 else ss := copy(mr, nr+2, 1)  { n+2 }
      end;
  Cezaris := ss[1]
end;
```

Šioje funkcijoje naudojome atskirus didžiųjų ir mažųjų raidžių sąrašus (konstantas `dr` ir `mr`). Dėl to atskirai teko šifruoti didžiąsias ir mažąsias raides, ir funkcijos algoritme gavome dvi labai panašias dalis. Jas galima būtų aprašyti funkcija. Kitas programos patobulinimo kelias – vartoti bendrą didžiųjų ir mažųjų raidžių sąrašą

```
'AaĄąBb...ZzŽž'
```

Pastaruoju atveju raidę *n* reikėtų keisti ne *n+2*-aja, o *n+4*-aja raide ir atskirai rašyti paskutinių keturių raidžių keitimą pirmosiomis keturiomis.

Uždaviniai

9.2.1. Parašykite funkciją `mažr`, kuri didžiąją raidę pakeistų mažąja, t.y. atvirkščią 1 pavyzdžio funkcijai `didr`.

9.2.2. Ar galima tvirtinti, kad

a) `didr(mažr(c)) = c;`

b) `mažr(didr(c)) = c`

čia `didr` – 1 pavyzdžio funkcija,

`mažr` – 1 uždavinio funkcija.

Praktikos darbas

9.2.1. Pranešimų šifravimas. Parašykite dvi programas teksto byloms šifruoti: vieną užšifruojančią atvirą tekstą Cezario kodu, kitą – iššifruojančią. Vienos raidės šifravimui panaudokite 2 pavyzdžio funkciją.

Programas išbandykite su bet kokiomis jūsų kompiuteryje esančiomis tekstų bylomis. Byla, gauta ją užšifravus ir vėl iššifravus, turi sutapti su pradine.

9.3. Žodžių rikiavimas

Žodžiai rikiuojami pagal abėcėlę. Skirtingų kalbų abėcėlės turi skirtingus raidžių rinkinius. Skiriasi ir aidžių išdėstymas. Pavyzdžiui, lietuviškoje abėcėlėje raidė *y* yra greta raidės *i*, o angliškoje abėcėlėje raidė *y* yra priešpaskutinė, prieš *z*. Paskalio kalboje pagrindinis dėmesys skiriamas skaičiavimams, o ne tekstų tvarkymui. Jos standarte nėra apibrėžtas simbolių išdėstymas. Tuo pačiu neapibrėžti ir simbolių lyginimo operacijų rezultatai. Tas pats pasakytina ir apie daugumą Paskalio dialektų. Todėl parodysime, kaip galima pačiam programuotojui apibrėžti simbolių išdėstymo eilę ir pagal ją rikiuoti simbolius bei žodžius.

Simbolių išdėstymą galima apibrėžti juos surašius abėcėlės tvarka į simbolių eilutę:

```
abc = 'aąbcčdeėęfghiįyjklmnoprsštuųūvzž'
```

Raidės eilės numerį abėcėlėje galima gauti taikant integruotą funkciją `pos`, o pagal gautą funkcijos rezultatą jau galima rikiuoti simbolius arba žodžius.

Pavyzdys. Procedūra žodžiams rikiuoti pagal abėcėlę.

```
const žsk = ...;                { žodžių skaičius }
type žodis = string;
     žodžiai = array [1..žsk] of žodis;
procedure rikiavimas (var žž: žodžiai);
var ž: žodis;
    žž: žodžiai;
    j, k: 1..žsk;               { žodžių masyvo indeksai }
  function daugiau (a, b: žodis): boolean;
    const abc = 'aąbcčdeęėfghiįyjklmnoprsštuųūvzž';
    var i: integer;
        lygu: boolean;
  begin
    i := 1;
    lygu := true;
    while lygu and (i <= length(a)) and (i <= length(b)) do
      begin
        lygu := copy(a, i, 1) = copy(b, i, 1);
        i := i+1
      end;
    if lygu
```

```
        then daugiau := length(a) > length(b)
        else daugiau :=
              pos(copy(a, i, 1), abc) > pos(copy(a, i, 1), abc)
    end;
  begin
    for k := 1 to žsk-1 do
      for j := 1 to žsk-k do
        if daugiau(m[j], m[j+1]) then
          begin          { du gretimi žodžiai keičiami vietomis }
            ž := žž[j+1];
            žž[j+1] := žž[j];
            žž[j] := ž
          end
  end;
```

Šioje procedūroje vartojamos tik mažosios raidės.

Praktikos darbas

9.3.1. Žodynas. Žodynuose įprasta žodžius rikiuoti taip, kad šios raidžių grupės būtų laikomos lygiavertėmis:

a, ą;

e, ę, ė

i, į, y;

u, ų, ū

Parašykite žodžių su lygiavertėmis raidėmis rikiavimo procedūrą.

9.4. Teksto redagavimas

Knygų, straipsnių ir kitokių tekstų rašymas yra kūrybinis darbas ir jį žmogus atlieka geriau už kompiuterį. Tačiau yra daugybė tekstų analizės ir tvarkymo darbų, kuriuos galima pavesti kompiuteriui.

Tekstai redaguojami literatūriškai ir techniškai. Literatūrinis redagavimas – tai geresnių išraiškos priemonių, geresnių terminų parinkimas, įvairūs stilistiniai taisymai, kurių tikslas – padaryti tekstą taisyklingesnį, logiškesnį ir aiškesnį nekeičiant jo esmės bei juo perteikiamos informacijos. Techninis redagavimas – tai teksto suskirstymas į puslapius ir jo išdėstymas puslapiuose, tarpų tarp žodžių sureguliavimas.

Redagavimui, ypač techniniam naudojami kompiuteriai. Tam yra parašytos specialios programos, vadinamos tekstų dorokliais arba redaktoriais, kurios padeda rinkti ir taisyti tekstus. Čia pateiksime porą nedidelių programų, kurios padės susidaryti vaizdą apie tekstų doroklių atliekamus veiksmus.

1 pavyzdys. Pradiniai duomenys – tekstas, sudarytas iš žodžių. Žodžiai skiriami tarpais. Laikoma, kad žodžiui priklauso ir greta jo parašyti, bet neatskirti tarpais skyrybos ženklai. Tarp gretimų žodžių gali būti vienas ir daugiau tarpų. Pateikiame programą, kuri tekstą pertvarko taip, kad tarp gretimų žodžių būtų tik po vieną tarpą, t.y. panaikina nebūtinus tarpus.

```
program red;
  var prad,              { padinių duomenų byla }
      rez: text;         { rezultatų byla }
      s,                 { tik ką perskaitytas simbolis }
      sa: char;          { anksčiau perskaitytas simbolis }
begin
  assign(prad, 'PRADDUOM.TEK'); reset(prad);
  assign(rez, 'REZULT.TEK'); rewrite(rez);
  sa := ' ';
  while not eof(prad) do
```

```
    begin
      read(prad, s);
      if (s = ' ') and (sa = ' ')
        then writeln(rez, ' ')    { rašomas tik vienas (pirmas) tarpas }
        else if s <> ' '
                then write(rez, s)
      sa := s;                    { dabar perskaitytas simbolis tampa ankstesniu }
      if eoln(prad)
        then begin
               writeln(rez);
               read(prad);
               sa := s
             end
    end;
  close(rez)
end;
```

Pradinių duomenų pavyzdys:

```
Kalnai  kelmuoti,   pakalnės nuplikę!
  Kas jūsų grožei    senobinei    tiki?
 Kur toj puikybė jūsų    pasidėjo?
Kur  ramus  jūsų  ūžimas  nuo  vėjo,
Kai     balto miško lapeliai šlamėjo
   Ir senos   pušys  siūravo, braškėjo?
```

Rezultatų pavyzdys:

```
Kalnai kelmuoti, pakalnės nuplikę!
Kas jūsų grožei senobinei tiki?
Kur toj puikybė jūsų pasidėjo?
Kur ramus jūsų ūžimas nuo vėjo,
Kai balto miško lapeliai šlamėjo
Ir senos pušys siūravo, braškėjo?
```

2 pavyzdys. Parašysime programą, kuri ne tik pašalina nereikalingus tarpus tarp žodžių, bet ir suskirsto tekstą į vienodo ilgio eilutes. Paskutinis žodis eilutėje gali netilpti. Tada jis (visas) keliamas į naują eilutę.

Programa turėtų būti panaši į 1 pavyzdžio programą. Tačiau joje negalima iš karto rašyti tik ką surinktą žodžio simbolį į rezultatų bylą, o pirmiau reikia įsiminti visą žodį, nes kol žodis neperskaitytas iki galo, tol neaišku, ar jis tilps toje eilutėje, ar reikės jį kelti į naują.

```
program taisymai;
  const eil = 60;       { teksto eilutės (puslapyje) ilgis }
  var prad,             { padinių duomenų byla }
      rez: text;        { rezultatų byla }
      s: char;          { tik ką perskaitytas simbolis }
      žodis: string;    { perskaitytų simbolių seka be tarpų }
      poz: integer;     { pirmoji laisva pozicija teksto (puslapio) eilutėje }
      k: 1..eil;
begin
  assign(prad, 'PRADDUOM.TEK'); reset(prad);
  assign(rez, 'REZULT.TEK'); rewrite(rez);
  žodis := '';
  poz := 1;
```

```
  while not eof(prad) do
    begin
      if eoln(prad)
        then begin
               readln(prad);
               s := ' '
             end
        else read(prad, s);
      if s <> ' ' then žodis := žodis+s;
      if (length(žodis) = eil) or
          eof(prad) or
         (length(žodis) > 0) and (s = ' ')
         then begin     { write }
                if (poz+length(žodis)-1 > eil) and (poz > 1)
                  then begin          { pereinama į naują eilutę }
                         writeln(rez);
                         poz := 1
                       end;
                if poz > 1        { eilutė jau pradėta }
                  then begin          { rašomas tarpas tarp gretimų žodžių }
                         write(rez, ' ');
                         poz := poz + 1
                       end;
               write(rez, žodis);     { rašomas žodis }
               poz := poz + length(žodis);
               žodis := ''
             end
    end;
    writeln(rez); close(rez)
end.
```

Čia pateikėme teksto tvarkymo programas naudodami simbolių eilutes. Paskalio standartas ir kai kurie jo dialektai neturi simbolių eilučių tipo ir kintamųjų. Be jų taip pat galima programuoti tokius uždavinius, tik vietoj simbolių eilučių reikia naudoti simbolių masyvus. Masyvai neturi eilutėms būdingų operacijų (operacijos +, funkcijos lenght), todėl tokiais atvejais jų veiksmus reikėtų programuoti – išreikšti esamomis operacijomis ir funkcijomis.

Praktikos darbas

9.4.1. Skyrybos ženklų tvarkymo programa. Rašybos taisyklės nustato, tarp kurių skyrybos ženklų ir žodžių turi būti paliktas tarpas (arba daugiau tarpų) arba tarpo neturi būti. Taisyklės yra tokios:

1. Po taško (sakinio pabaigoje), dvitaškio, kabliataškio ir kablelio reikia palikti tarpą, o prieš šiuos ženklus tarpo nereikia.

2. Skliaustų arba kabučių išorėje reikia palikti tarpą, o jų viduje tarpo nereikia.

3. Abipus brūkšnio paliekami tarpai.

4. Abipus brūkšnelio tarpų nereikia.

Parašykite programą šiam uždaviniui spręsti.

UŽDAVINIŲ SPRENDIMAI (ATSAKYMAI)

1.1.1.

```
program vidurkis3;
  var a, b, c, vid: real;
begin
  read(a);
  read(b);
  read(c);
  vid := (a+b+c)/3;
  writeln(vid: 8: 2)
end.
```

1.1.2. Paskalio kalbos sistemos terpėje esančiu *dorokliu* renkami ir taisomi programų tekstai. Paskalio kalbos *transliatorius* arba *kompiliatorius* išverčia programos tekstą iš *Paskalio* kalbos į *kompiuterio* kalbą.

1.3.1. Taip.

1.3.2.

```
255  255
255   11
```

1.3.3. `256`

1.5.1.

`a(5)`	varduose skliaustai nevartojami;
`Lt.`	varduose taškai nevartojami;
`a'`	varduose apostrofai nevartojami;
`begin`	vardas negali sutapti su baziniu žodžiu;
`prekės kaina`	varduose negali būti tarpų.

1.5.2. Yra 3 skirtingi vardai. Sutampa:

```
vid         VID            Vid
TrikPlotas  Trikplotas
```

1.5.3.

Sveikieji skaičiai	Realieji skaičiai	Simboliai	Eilutės
`1998`	`1.25E4`	`'+'`	`'1.25'`
`0`	`125E2`	`''''`	`'STALAS'`
`-56`	`1.25`	`' '`	`'a+b'`
	`0.0`	`'5'`	`'nulis'`
			`''`

Simboliai gali būti traktuojami kaip vieno simbolio eilutės.

1.5.4.

```
12E-13
3E6
-1E-9
```

1.5.5. Visos eilutės skirtingos. Taigi, jų yra 9.

1.5.6.

Baziniai žodžiai: **`program`**, **`var`**, **`begin`**, **`end`**.
Integruoti vardai: `read`, `writeln`.
Programuotojo sudaryti vardai: `vidurkis`, `a`, `b`, `vid`.

1.6.1.

```
program pasveikinimas;
begin
  writeln('Sveiki, visi')
end.
```

1.6.2. Komentarai gali būti įvairūs. Pateikiame vieną galimų variantų.

```
var prad: real;        { pradinė pinigų suma }
    proc: real;        { metinių palūkanų procentas }
    mn: integer;       { indėlio laikymo banke trukmė (mėn.) }
    galut: real;       { galutinė suma su palūkanomis }
```

2.1.1. Jeigu prieskyros sakinyje būtų užrašyta konstanta *n*, tai pataisius programą (kai reikia rašyti tikslesnius rezultatus), „susigadintų“ skaičiavimai.

2.2.1.

sveikojo: c, d, e.

realiojo: a, b, f.

2.2.2. a) 2; b) 1.

2.2.3. b, c, f, i.

2.2.4.

```
program GeomVidurkis;
  var a, b, geom: real;
begin
  read(a);
  read(b);
  geom := sqrt(a*b);
  writeln(geom: 8: 2)
end.
```

2.2.5.

```
Lt := trunc(suma);
ct := round((suma-Lt)*100)
```

2.3.1.

```
22 33 22
```

2.3.2.

```
444 333 444
```

2.3.3. Š programa neturi prasmės, nes kompiuteriui liepiama spausdinti kintamojo `a` reikšmę prieš tai negu ji buvo perskaityta – atliekant pirmąjį sakinį kintamojo reikšmė dar neapibrėžta.

2.4.1.

```
program centai;
  var suma,                                   { pinigų suma }
      mon: integer;                           { monetų skaičius }
begin
  read(suma);
  mon :=       suma div 50;  suma := suma mod 50;
  mon := mon + suma div 20;  suma := suma mod 20;
  mon := mon + suma div 10;  suma := suma mod 10;
  mon := mon + suma div  5;  suma := suma mod  5;
  mon := mon + suma div  2;  suma := suma mod  2;
  mon := mon + suma { mod 1 };
  writeln(mon)
end.
```

2.4.2.

```
c := a;
a:= b;
b:= c;
```

2.4.3. a)

```
program SkSuma3;
  var a,                      { triženklis skaičius }
      sum: integer;           { ir jo skaitmenų suma }
begin
  read(a);
  sum := a div 100 + a mod 100 div 10 + a mod 10;
  writeln(sum)
end.
```

b)

```
program Keitimas3;
  var a,                { triženklis skaičius }
      atv: integer;     { perrašytas atbulai }
begin
  read(a);
  atv := a mod 10 * 100 +
         a mod 100 div 10 * 10 +
         a div 100;
  writeln(atv)
end.
```

2.5.1.

```
program TrysBylos;
  var prad,                       { rezultatų byla }
      pirma, antra: text;         { pradinių duomenų bylos }
      x, y: integer;              { perskaityti skaičiai }
begin
  assign(prad, 'A:\SKAIČIAI.TXT'); rewrite(prad);
  assign(pirma, 'C:\DUOMENYS\PIRMA.TXT'); reset(pirma);
  assign(antra, 'C:\DUOMENYS\ANTRA.TXT'); reset(antra);
  read(pirma, x); writeln(prad, x);
  read(antra, x, y); writeln(prad, x, y: 10);
end.
```

2.5.2.

```
program TiesLygtis;
var a, b,          { lygties koeficientai }
    x: real;       { lygties šaknis }
    byla: text;    { pradinių duomenų byla }
begin
  assign(byla, 'C:\PRAD.XXX'); reset(byla);
  read(byla, a, b);
  x := b/a;
  writeln(x)
end.
```

2.6.1. Ten pat, kur ir Paskalio programa.

2.6.2.

```
program laikas;
  var p,                    { parų skaičius }
      h,                    { valandų skaičius }
      min: integer;         { minučių skaičius }
begin
  assign(input, 'DUOM.TXT'); reset(input);
  assign(output, 'REZ.TXT'); rewrite(output);
  read(p);
  h := p*24;      write(h: 4);
  min := h*60;    writeln(min: 6)
end.
```

3.1.1.

a ir b: sveikųjų skaičių tipo kintamajam negalima priskirti loginės reikšmės;

c ir d: loginio tipo kintamajam neleidžiama priskirti skaičių reikšmės;

e: negalima lyginti sveikųjų skaičių su loginėmis reikšmėmis;

f ir g: aritmetinės operacijos su loginėmis reikšmėmis neatliekamos;

h: sveikųjų skaičių tipo kintamajam neleistina priskirti loginės reikšmės.

3.1.2.

```
TRUE
FALSE
TRUE
```

3.1.3. Jeigu kintamieji a ir x yra tokių duomenų tipų, kurių reikšmes galima lyginti (pvz., abi reikšmės skaičiai, abi – simboliai ir pan.), o kintamasis b yra loginio tipo, tai dviguba nelygybė virsta taisyklingu Paskalio reiškiniu, tik ji nebeturi įprastos matematinės skaičių intervalo prasmės.

3.1.4.

a	b	a<b	a<=b	a=b	a<>b	a>b	a >=b
false	false	false	true	true	false	false	true
false	true	true	true	false	true	false	false
true	false	false	false	false	true	true	true
true	true	false	true	true	false	false	true

3.1.5. a) `true`; b) `false`; c) `true`; d) `true`; e) `true`; f) `false`; g) `true`; h) `false`; i) `true`; k) `false`; l) `true`.

3.1.6.

```
a) rez := (a = b) and (b = c);
b) rez := (a <> b) and (b <> c) and (a <> c);
c) rez := (a = b) or (b = c) or (a = c);
d) rez :=  (a mod 2 = 0) and
           (b mod 2 = 0) and
           (c mod 2 = 0);
e) rez := (a > 0) and (b > 0) and (c > 0) and
          (a <= 100) and (b <= 100) and (c <= 100).
```

3.1.7.

a) `(a < b) and (b < c) and (c < d)`

b) `(a <= b) and (b <= c) and (c <= d)`

3.1.8.

a) `(a < b+c) and (b < a+c) and (c < a+b);`

b)
```
(a < b+c) and (b < a+c) and (c < a+b) and
  ((a = b) or (b = c) or (a = c));
```

c) `(a = b) and (b = c).`

3.1.9.

```
sqrt(sqr(cx-tx) + sqr(cy-ty)) < r
```

3.1.10.

a)
```
a and not b and not c and d  or
  not a and b and c and not d
```

b)
```
a and b or c and d
```

c)
```
not a and not c  or
  not b and not d
```

3.1.11. `(a or c) and (b or d)`

3.1.12.

```
(m mod 400 = 0) or (m mod 100 <> 0) and (m mod 4 = 0)
```

3.1.13.

```
a+1 > a  true
a+1 > 1  false true
a*5 > 0  false
a*a > 0  false true
a*a >= 0 true.
```

3.2.1. a) `not (a or b)` arba `not a and not b`; b) `true`

3.3.1. a =4 , b = 8

3.3.2. a = 10, b = 5 arba a = 5, b = 10.

3.3.3. Programa teisinga ir ji duos tą patį rezultatą, kaip ir programa `minimumas`.

3.3.4.

```
if a mod 2 = 0 then f := a+b
               else f := a*b
```

3.3.5. 4.

3.3.6.

```
program olimpiadanra;
  var metai,
      nr: integer; { olimpiados eilės numeris }
begin
  read(metai);
  write(metai: 4, ' metai yra ');
  if (metai >= 1896) and (metai mod 4 = 0)
    then writeln('olimpiniai')
    else writeln('neolimpiniai')
end.
```

3.3.7. Jeigu rašymo sakiniuose nebūtų nurodyta tarpų, tai tarp rašomų žodžių bei skaičių taip pat nebūtų tarpų ir jie susilietų. Būtų rašoma, pavyzdžiui, taip:

```
Skaičius12345nedalusiš7
```

3.3.8. Perpratus apgaulingą uždavinio formuluotę matyti, kad tereikia pakeisti skaičiaus ženklą priešingu.

```
program LabaiPaprasta;
  var x: integer;
begin
  x := -x;
  writeln(x)
end.
```

3.4.1. Taip, programos `minimumas` ir `mini` ekvivalenčios, t.y. visada duoda tą patį rezultatą.

3.4.2.

```
a) if a <> b then if a < b then b := b - a
                           else a := a - b;
   if a < b then b := b - a
            else if a > b then a := a - b
b) if a < b then a := a + 1
            else if a > b then b := b + 1
c) if a < b then b := b - 1
            else if a > b then a := a - 1
```

3.4.3. a) `1 2 3 4`

b) `4 3 1 2`

Kai $a = b$, tai neatliekama nė viena sąlyginio sakinio šaka, neįvedami kiti du pradiniai duomenys, kintamųjų c ir d reikšmės lieka neapibrėžtos.

3.4.4.

```
if a > b then begin c := 1; d := 2 end
         else begin c := 3; d := 4 end
```

3.4.5. Taip

3.4.6. Ne visada. Kai $a = 5$, rezultatai skiriasi: a) $c = 0$, b) $c = 2$.

3.4.7.

```
program perdaug;
  var a, b: integer;
begin
  read(a, b);
```

```
  if a > 10 then b := b - 5;
  if a > 5  then begin a := 5;
                       b := b - 5
                 end;
  writeln(a, b: 6)
end.
```

3.4.8. $a \geq 0,\ b \geq 0,\ c \leq 10.$

3.4.9.

a) `if (a > b) and (b < c) then a := a + 1;`

b) `if not log then a := a + 1.`

3.4.10.

```
program dalus7;
  var x: integer;
begin
  read(x);
  write('Skaičius ', x, ' ');
  if x mod 7 <> 0
    then write('ne');
  writeln('dalus iš 7')
end.
```

3.6.1. a) 5; b) 6.

3.6.2. 2 kartus, $a = 3$, $b = 5$, $s = 8$.

3.6.3. $a = 4$, $b = 2$. Ciklui priklauso tik vienas sakinys, esantis po jo antraštės. Sakinys $b := b + 1$ ciklui nepriklauso ir atliekamas tik vieną kartą. Kad programa būtų aiškesnė ir neklaidintų skaitytojo, šį sakinį reikėtų rašyti naujoje eilutėje ir lygiuoti su ciklo antraštės pradžia:

```
while a <= 3 do
  a := a+1;
b := b+1
```

3.6.4

```
program minimumas;
  var prad: text;
      sk,              { perskaitytas skaičius }
      min: integer;    { mažiausias skaičius tarp jau perskaitytų }
      skbyla: text;    { skaičių byla }
begin
  assign(skbyla, 'SKAIČIAI.TEK'); reset(skbyla);
  min := maxint;             { tikrai nemažesnis už bet kurį skaičių }
  while not eof(skbyla) do
    begin
      read(skbyla, sk);
      if sk < min            { rastas skaičius, mažesnis už min }
        then min := sk
    end;
  writeln(min)
end.
```

3.7.1. a ir c.

3.7.2. Ciklo sakinys, prasidedantis žodžiu **`while`** valdo *vieno* sakinio kartojimą, o ciklo sakinys, prasidedantis žodžiu **`repeat`** valdo *sakinių sekos* kartojimą.

3.8.1.

```
program log1;
  const epsilon = 1E-6;      { liekamasis narys }
  var k: integer;
      ex, narys: real;
begin
  ex := 0.0;
  narys := 1.0;
  k := 1;
  while narys >= epsilon do
    begin
      ex := ex + narys;
      narys := narys*ex/k;
      k := k + 1
    end;
  writeln(ex)
end.
```

3.8.2.

```
program pipi;
  const epsilon = 1E-10;     { liekamasis narys }
  var k : integer;
      pi, narys: real;
begin
  pi := 0.0;
  narys := 1.0;
  k := 1;
  while abs(narys) >= epsilon do
    begin
      pi := pi + narys;
      if narys < 0 then narys := 1/k
                   else narys := -1/k;
      k := k + 2
    end;
  writeln(pi)
end.
```

4.1.1. a) `bevardis`; d) `false`; f) `Liūtas`; g) `false`, kiti – netaisyklingi.

4.1.2.

{ 3, 4, 5 } tipų vardų (`asmuo`, `giminė`) negalima vartoti prieskyros sakiniuose;

{ 7 } abu lyginimo operacijos operandai turi būti to paties tipo;

{ 10 } funkcijos `pred` parametras negali būti pirmoji tipo reikšmė.

4.1.3.

```
if (a = b) and (b = c) and(c = d) then forma := kv
    else if (a = b) and (c = d) or
            (a = c) and (b = d) or
            (a = d) and (c = b) then forma := st
                                else forma := ne
```

4.1.4.

```
if a = sekm                          { sekmadienis – paskutinė diena }
  then gretimos := b = šešt
  else if a = pirm                   { pirmadienis – pirmoji diena }
         then gretimos := b = antr
         else gretimos := (b = pred(a) or b = succ(a);
```

4.2.1. Visada teisingi: 3, 4;
visada neteisingi 5,6;
ne visada teisingi: 1, 2.

4.2.2.

```
var x1:  -9..15;
    x2: -15..9;
    x3: -10..10;
    x4:  -5..-1;
    x5: -50..50;
    x6: -58..66;
    x7: -28..40.
```

4.3.1.

```
program mėnuo;
  var mėnnr: integer;
begin
  read(mėnnr);
  case mėnnr of
    1: write('sausis');     2: write('vasaris');
    3: write('kovas');      4: write('balandis');
    5: write('gegužė');     6: write('birželis');
    7: write('liepa');      8: write('rugpjūtis');
    9: write('rugsėjis');  10: write('spalis');
   11: write('lapkritis'); 12: write('gruodis')
  end;
  writeln
end.
```

4.3.2.

```
if log then b := b+1
       else a := a+1
```

4.3.3.

```
program dienos;
  var metai, mėnuo, diena: integer;
begin
  read(metai, mėnuo);
  case mėnuo of
    1, 3, 5, 7, 8, 10, 12: diena:= 31;
    4, 6, 9, 11:           diena:= 30;
    2: if (metai mod 400 = 0)
            or ((metai mod 100 <> 0) and (metai mod 4 = 0))
          then diena := 29
          else diena := 28
  end;
  writeln(diena)
end.
```

4.4.1. a) 11; b) 0; c) 3; d) 1; e) 11; f) 2a+1; g) 11; h) 0.

4.4.2. `0 1 2 3 4 5`

4.4.3.

```
program kvadratnelyg;
  var j, suma: integer;
begin
  suma := 0;
  for j := 10 to 99 do
    if j mod 2 <> 0 then suma := suma + j*j;
  writeln(suma)
end.
```

4.4.4.

```
program GeomProgresija;
  var a,          { pirmasis narys }
      q,          { vardiklis }
      k: integer;
begin
  read(a, q);
  write(a);
  for k := 2 to 10 do
    begin
      a := a * q;
      write(a)
    end
end.
```

4.4.5.

```
program AritmProgresija;
  var a,          { pirmasis progresijos narys }
      d,          { skirtumas }
      n,          { pirmasis sumuojamas narys }
      k: integer;
begin
  read(a, d, n);
  a := a + d*(n - 1);
  write(a);
  for k := 2 to 10 do
    begin
      a := a + d;
      write(a: 5)
    end;
  writeln
end.
```

4.4.6. a) n=5, m=2;
b) n=10; m=1;
c) n=10; m=0;
d) n=1; m=12;
n=2; m=6;
n=3; m=4;
n=4; m=3;
n=6; m=2;
n=12; m=1;
e) n=1; m=13;

f) tokių reikšmių nėra.

4.4.7. 10 kartų. Reikšmės reiškinių, esančių ciklo **for** antraštėje, apskaičiuojamos vieną kartą, prieš atliekant ciklą. Todėl tolesni jų keitimai nebeturi įtakos ciklo kartojimų skaičiui.

4.4.8. Ne.

4.4.9. 1) Ciklo kintamojo reikšmės negalima keisti ciklo viduje (sakinys `k := k+1`); 2) Ciklo kintamojo reikšmė neapibrėžta baigus ciklą (sakinys `writeln(k)`); 3) nereikalingas kabliataškis po sakinio `writeln(k);` 4) Sakinyje `writeln(k)` reikėtų nurodyti rašymo formatą, kad nesusilietų gretimi skaičiai.

4.4.10. Programoje `pirma` yra dvi klaidos: 1) panaudota neapibrėžta kintamojo `k` reikšmė ir 2) kintamojo `k` reikšmė atliekant ciklą paskutinį kartą viršija jo apraše nurodytą atkarpos tipo rėžį.

4.4.11. 1) Ne visi transliatoriai (tarp jų ir Turbo Paskalis) tikrina, ar keičiama ciklo kintamojo reikšmė cikle. 2) Tai, kad ciklo kintamojo reikšmė neapibrėžta, dar nereiškia, kad jis neturi jokios reikšmės. Jis gali turėti reikšmę, tiktai Paskalio kalba nenustato kokia ta reikšmė turi būti. Ta pati programa, sutransliuota kitu transliatoriumi, gali priskirti kitą reikšmę.

4.5.1.

```
program dalikliai3;
  var n,                { duotas skaičius }
      dal: integer;     { kandidatas į duoto skaičiaus daliklius }
begin
  read(n);
  dal := 1;
  while dal*dal <= n do
    begin
      if n mod dal = 0
        then begin
               write(dal, ' ');
               if n div dal <> dal
                 then writeln(' ', n div dal)
                 else writeln
             end
    end;
    dal := dal + 1
end.
```

Į vieną eilutę rašomi du skaičiai: daugiklis ir už jį didesnis jo „porininkas“. Kadangi vieneto „porininkas“ yra *n*, tai nebereikia programos pabaigoje spausdinti *n*. Dar vienas sąlyginis sakinys reikalingas tam, kad būtų išvengta dviejų vienodų daliklių rašymo, kai daliklis lygus savo „porininkui“.

4.5.2.

```
program daugikliai;     { pirminiai }
  var n,                { duotas skaičius }
      d: integer;       { kandidatas į duoto skaičiaus daugiklius }
begin
  read(n);
  d := 2;
  while d >= n do
    begin
      while n mod d = 0 do
        begin
          writeln(d);
          n := n div d
```

```
        end;
      d := d + 1
  end
end.
```

4.5.3.

```
program dalus235;
  const žingsnis = 2*3*5;
  var k: 1..10;
      sk: integer;
begin
  sk := 0;
  for k := 1 to 10 do
    begin
      sk := sk + žingsnis;
      writeln(sk)
    end
end.
```

4.6.1. Kokie bus aritmetinių operacijų su realiaisiais skaičiais rezultatai, tiksliai pasakyti negalima. Kompiuteris, kuriame realieji skaičiai vaizduojami taip, kaip aprašyta šiame skyrelyje, turėtų pateikti tokius (apytikslius) rezultatus:

```
7.00000E-05
7.54321E-05
```

4.6.2. d ir g.

5.1.1. Ne. Pavyzdžiui, kai n = -5, tai pagal pirmąją formuluotę turi būti sumuojami skaičiai -4, -2, o pagal antrąją – -6, -4, -2.

5.2.2. Nepasakyta, kaip turi būti pateikiamas rezultatas. Formuluotę būtų galima papildyti šitaip: *jeigu yra bent vienas nulis, spausdinti žodį TAIP, priešingu atveju – NE.*

5.4.1. Kai *a = 0*, pagal pirmąjį sakinį bus atliekamas veiksmas `b := 10`, pagal antrąjį – `b := 5`

5.4.2.

a) $a \le 15$;

b) ($a = 0$) **or** (a = 1).

5.4.3.

a) ir b) $a \le 10$.

5.4.4.

a) $k > 0$;

b) $k \le 0$;

c) ($k \ge 0$) **and** (k **mod** 2 = 0).

5.8.1.

```
program simboliai;
  var a, i, s: integer;
begin
  read(a);
  s := 0;
```

```
  for i := 1 to a do
    s := s+i*i*(i+1)
  writeln(s)
end.
```

5.8.2.

a) 5;

b)

```
program tvarka;
  var i, s: integer;
begin
  s := 1;
  i := 10;
  while i mod 7 = 0 do
    i := i - 1; s := s + i*i;
  if s > 25
    then if s = 99
           then s := s + 1
           else s := s div i - 2
    else s := s - 1;
  writeln(s)
end.
```

c) 5.

5.9.1.

```
program NatSuma;
  var n, suma: integer;
begin
  read(n);
  suma := (n*n + n) div 2;
  writeln(suma)
end.
```

6.2.1. d) ir f) faktinių parametrų skaičius nelygus formaliųjų parametrų skaičiui;
e) faktinių parametrų tipas nesutampa su formaliųjų parametrų tipu.

6.2.2.

```
function min (a, b: integer): integer;
begin
  if a <= b then min := a
            else min := b
end;
```

6.2.3.

```
function fakt (n: integer): integer;
  var f, k: integer;
begin
  f := 1;
  for k := 1 to n do
    f := f * k;
  fakt := f
end;
```

6.2.4. `write(max4(a mod 10, b mod 10, c mod 10, d mod 10))`

6.2.5.

```
function sksuma (n: integer): integer;
  var suma: integer;
begin
  suma := 0;
  while n > 0 do
    begin
      suma := suma + n mod 10;
      n := n div 10
    end;
  sksuma := suma
end;
```

6.2.6.

```
function progresija (a, b, c: integer): boolean;
begin.
  progresija := b-a = c-b
end;
```

6.2.7.

```
function tūkst (n: integer): integer;
  const priedas = 500;      { apvalinimo konstanta }
begin
  if n > 0 then n := n + priedas
           else n := n - priedas;
  tūkst := n div 1000
end;
```

6.3.1. Taip.

6.3.2. a) jeigu viena data yra prieš kalendoriaus reformą, o kita – po jos, tai iš gauto dienų skaičiaus atimti 10 (tiek dienų „dingo“ keičiant kalendorių).

b) jeigu būtų į programą įjungta funkcija, tikrinanti, ar teisingi pradiniai duomenys, tai reikėtų prie neegzistuojančių datų priskirti datas nuo 1582 m. spalio 5 d. iki tų pat metų spalio 14 d.

6.3.3.

```
function ArDataTeisinga (m, mėn, d: integer): boolean;
  var mp: 28..31; { paskutinė mėnesio diena }
  function kel (m: integer): boolean;
  begin
   kel := (m mod 400 = 0) or
          (m mod 100 <> 0) and
          (m mod 4 = 0)
  end;
begin
  case mėn of
    1, 3, 5, 7, 8, 10, 12: mp := 31;
    2: if kel(m) then mp := 29
                 else mp := 28;
    4, 6, 9, 11: mp := 30
  end;
  ArDataTeisinga := (m >= 1563) and (m <= 3000) and
                    mėn in [1..12] and
                    d in [1..mp]
end;
```

6.4.1. a) ir b) – faktinių parametrų skaičius turi būti lygus formalių jų parametrų skaičiui; c) b = 30; d) b = 20; e) b = 14; f) b = 22; g) ir h) formalųjį parametrą – kintamąjį galima keisti tik kintamuoju (ne konstanta ir ne reiškiniu); i) a = 50; j) formaliųjų ir juos atitinkančių faktinių parametrų tipai turi sutapti.

6.4.2.

```
kvkub(m-1, a, b);
kvkub(n, c, d);
x := c-a;
y := d-b
```

6.4.3.

```
procedure trikampis (a, b, c: integer);
  var stat, lyg: boolean);
begin
  stat := a*a + b*b = c*c;
  lyg := (a = b) or (b = c)
end;
```

6.5.1.

```
25     30
 4      0
```

6.5.2. Pirmasis parametras x apvalinamas 10^y tikslumu.

a) 2000; b) 0; c) 1500.

6.5.3.

```
procedure laipsnis (var p: integer; n: integer);
            { p keliamas laipsniu n }
  var pn, i: integer;
begin
  pn := 1;
  for i := 1 to n do
    pn := pn * p;
  p := pn
end;
```

6.5.4.

```
procedure sek (var h, min, s: integer);
begin
  if s < 59 then s := s+1
    else begin
      s := 0;
      if min < 59 then min := min+1
        else begin
          min := 0;
          if h < 12 then h := h+1
                    else h := 1
        end
    end
end;
```

Kitas variantas

```
procedure sekundė (var h, min, s: integer);
begin
  s := s + 1;
  min := min + s div 60;
```

```
  s := s mod 60;
  h := h + min div 60;
  min := min mod 60;
  if h = 13 then h := 1
end;
```

6.5.5.

```
2    2   2
1    1   2
3    3   2
```

6.6.1.

```
function vid (a, b: real): real;
begin
  vid := (a+b)/2
end;
```

6.6.2.

```
function pinigai (prad, proc: real; mn: integer): real;
begin
  pinigai := prad + prad*(proc/100*mn/12)
end;
```

6.6.3.

```
procedure LtCt (prad, proc: real; mn: integer;
                var Lt, ct: integer);
  var suma: real; { galutinė suma (su palūkanomis) }
begin
  suma := prad + prad*(proc/100*mn/12);
  Lt := trunc(suma);
  ct := round((suma-Lt)*100)
end;
```

6.6.4.

```
type forma: (kv, st, ne);
      { kv – kvadratas (tuo pačiu ir stačiakampis) }
      { st – stačiakampis bet ne kvadratas }
      { ne – ne stačiakampis }
function kvstat (a, b, c, d: real): forma;
begin
  if (a = b) and (b = c) and(c = d) then forma := kv
    else if (a = b) and (c = d) or
            (a = c) and (b = d) or
            (a = d) and (c = b) then forma := st
                                else forma := ne
end;
```

7.1.1

a) `record x, y: real end;`

b) `record a, b, c: real end;`

7.1.2 a)

```
function lygu (a, b: laikas): boolean;
```

```
begin
  lygu := (a.p = b.p) and
          (a.h = b.h) and
          (a.min = b.min) and
          (a.s = b.s) and
end;
```

b)

```
function trumpesnis (a, b: laikas): boolean;
begin
  trumpesnis := a.s + 60*(a.min + 60*(a.h + 24*a.p))
              < b.s + 60*(b.min + 60*(b.h + 24*b.p))
end;
```

c)

```
procedure suma (a, b: laikas; var c: laikas);
  var s: integer;
begin
  s := a.s + b.s +
       60*(a.min + b.min + 60*(a.h + b.h + 24*(a.p + b.p));
  c.s := s mod 60;
  s := s div 60;
  c. min := s mod 60;
  s := s div 60;
  c.h := s mod 24;
  c.p := s div 24
end;
```

Kitas variantas

```
procedure suma (a, b :laikas; var c: laikas);
  var s: integer;
      p: 0..1;
begin
  s := a.s + b.s;
  c.s := s mod 60; p := s div 60;
  s := a.min + b.min + p;
  c. min := s mod 60; p := s div 60;
  s := a.h + b.h +p;
  c.h := s mod 24; p := s div 24;
  c.p := a.p + b.p + p
end;
```

d)

```
function paros (a: laikas): real;
begin
  paros := ((a.s/60 + a.min)/60 + a.h)/24 + p
end;
```

7.1.3.

```
type kampas = record
                laipsn: 0..maxint;
                min, s: 0..59
              end;
function radianai (k: kampas): real;
  const pi = 3.1415926536;
begin
  radianai := ((k.s/60 + k.min)/60 + k.laipsn)/360*2*pi
end;
```

7.1.4.

```
type matas = record
               jardas: 0..maxint;
               pėda: 0..2;
               colis: 0..11
             end;
procedure sudėti (a, b: matas; var c: matas);
  var x: integer;
begin
  x := a.colis + b.colis +
       12*(a.pėda + b.pėda + 3*(a.jardas + b.jardas));
  c.colis := x mod 12;
  x := x div 12;
  c.pėda := x mod 3;
  c.jardas := x div 3
end;
function metrai (a: matas): real;
begin
  metrai := (((a.jardas * 3) +
              a.pėda) * 12 + a.colis)*25.4/1000
end;
```

7.2.1.

```
record
  n: 3..maxint;  { kampų skaičius }
  a: real        { kraštinės ilgis }
end;
```

7.2.2.

```
function plotas (t: trikampis): real;
  var atk: atkarpa;
      a, b, c,         { kraštinių ilgiai }
      p: real;         { pusė perimetro }
begin
  atk.a := t.a; atk.b := t.b;
  a := ilgis(atk);
  atk.a := t.b; atk.b := t.c;
  b := ilgis(atk);
  atk.a := t.c; atk.b := t.a;
  c := ilgis(atk);
  p := (a + b + c)/2;
  plotas := sqrt(p*(p-a)*(p-b)*(p-c))
end;
```

7.3.1.

```
type temp = array[1..12] of
  record
    vid, min, max: real
  end
```

7.3.2. a)

```
  procedure kalendorius(mėn: kalend);
    var sav: savnr;
        sd: savd;
```

```
  begin
    for sd := pirm to sekm do
      begin
        for sav := 1 to 6 do
          if mėn[sav, sd] <> 0
            then write(mėn[sav, sd]: 3)
            else write('   ');
        writeln
      end
  end;
```

b)

```
procedure kalendorius (mėn: kalend);
  var sav: savnr;
      sd: savd;
begin
  for sav := 1 to 6 do
    begin
      for sd := pirm to sekm do
        if mėn[sav, sd] <> 0
          then write(mėn[sav, sd]: 3)
          else write('   ');
      writeln
    end
end
```

7.3.3. Vidiniame cikle operaciją < pakeisti operacija >.

7.4.1. a ir c

7.5.1.

```
type spalva = (balta, žalia, raudona, geltona, oranžinė, mėlyna);
     siena = array [1..3] of
               array [1..3] of spalva;
     šonas = (kairė, dešinė, priekis, užpakalis, viršus, apačia);
     kubas = array [šonas] of siena;
```

7.5.2.

```
function Rubikas (x: kubas): boolean;
  var spal: array [spalva] of 0..54;
      sp: spalva;
      š: šonas;
      i, j: 1..3;
      b: boolean;
begin
  for sp := balta to mėlyna do
    spa[sp] := 0;
  for š := kairė to apačia do
    for i := 1 to 3 do
      for j := 1 to 3 do
        begin
          sp := kubas[š][i][j];
          spal[sp] := spal[sp] + 1
        end;
  b := true;
  for sp := balta to mėlyna do
    b := b and (spal[sp] = 9);
```

```
  Rubikas := b
end;
```

7.6.1. d.

7.6.2. `a=b, c=d.`

7.6.3.

b) abu operandai (ir antrasis!) turi būti aibės;

c) operacijos **in** pirmasis operandas turi būti aibės elemento tipo;

e) neteisinga antrojo operando aibė: antrasis atkarpos rėžis turi būti nedidesnis už pirmąjį.

7.6.4.

```
program skirtingi;
  var byla: text;                 { skaičių byla }
      skirtingi: boolean;         { ar visi perskaityti skaičiai skirtingi }
      sk: integer;                { perskaitytas skaičius }
      esami: set of 0..99;        { perskaitytų skaičių aibė }
begin
  assign(byla, 'SK.TEK'); reset(byla);
  skirtingi := true;
  while not eof(byla) do
    begin
      read(byla, sk);
      if sk in [0..99]
        then
          begin
            if sk in esami then skirtingi := false;
            esami := esami + [sk]
          end
    end;
  write('Byloje SK.TEK ');
  if not skirtingi then write('ne');
  writeln('visi skaičiai iš [0; 99] yra skirtingi')
end.
```

8.1.1.

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

8.1.2. a) 2; b) 1; c) 14; **div**.

8.1.3. Rekursija būtų nebaigtinė.

Sąlygą `n = 1` reikėtų pakeisti sąlyga `n < 1`.

8.1.4.

```
function narys (n: integer): integer;
begin
  if n = 1
    then narys := 1
    else if n mod 2 = 0
      then narys := narys(n div 2)
```

```
      else narys := narys(n div 2) + narys(n div 2 + 1)
end;
```

8.1.5.

```
function suma (n, d: integer): integer; { rekursinė }
begin
  if n < 0
    then suma := 0
    else suma := n*n + suma(n-d)
end;

function suma (n, d: integer): integer; { ne rekursinė }
  var s: integer;
begin
  s := 0;
  while n > 0 do
    begin
      s := s + n*n;
      n := n-d
    end;
  suma := s
end;
```

8.1.6.

```
function ff (n: integer): integer;
begin
  if (n = 0) or (n = 1)
    then ff := 1
    else ff := n*ff(n-2)
end;
```

8.3.1.

```
function bdd (a, b: integer): integer;
begin
  if a = b then bdd := a
    else if a < b then bdd := bdd(a, b-a)
      else bdd := bdd(a-b, b)
end
```

8.3.2.

```
program dalikliai;
  var d, daliklis: integer;
  function dbd(x, y: integer): integer;
  begin
    if x = 0 then dbd := y
      else dbd := dbd(y mod x, x)
  end;
begin
  daliklis := 0;
  read(d);
  while d <> 0 do
    begin
      daliklis := dbd(daliklis, abs(d));
      read(d)
    end;
  write(daliklis)
end.
```

8.3.3.

```
program kartotiniai;
  var k, kartotinis: integer;
  function mbk (a, b: integer): integer;
    function dbd (x, y: integer): integer;
    begin
      if x = 0 then dbd := y
        else dbd := dbd(y mod x, x)
    end;
  begin
       mbk := a div dbd(a, b) * b
  end;
begin
  kartotinis := 1;
  for k := 2 to 10 do
    kartotinis := mbk(kartotinis, k);
  writeln(kartotinis)
end.
```

8.3.4.

```
function tarpusavyp (a, b: integer): boolean;
  function dbd (x, y: integer): integer;
  begin
    if x = 0 then dbd := y
      else dbd := dbd(y mod x, x)
  end;
begin
  tarpusavyp := dbd(a, b) <= 1
end;
```

8.4.1.

```
15  14  13  12  11  10
10  11  12  13  14  15
```

8.4.2. a) kai n ≥ 5, rašoma `*`; kai n < 5, skaičiavimas nesibaigs;

b) kai *b* = *false*, nebus atliekami jokie veiksmai; kai *b* = *true*, skaičiavimas nesibaigs.

9.1.1. Prieš skaitymą procedūra `readreal` byla turi būti paruošta analogiškai kaip ir prieš skaitymą integruotą procedūra `read`.

9.1.2.

```
type skai = 0..9;
     simb = '0'..'9';
function sk (si: simb): skai;
begin
  sk := ord(si) - ord('0')
end;
function si (sk: skai): simb;
begin
  si := chr(ord('0') + sk + 1)
end;
```

9.1.3.

```
procedure readsveikas (var byla: text; var p: integer);
```

```
  var s: char;
      neigiamas: boolean;
begin
  p:= 0; neigiamas := false;
  read(byla, s);
  while s = ' ' do     { praleidžiami tarpai }
    read(byla, s);
  if s = '-'
    then begin
           neigiamas := true;
           read(byla, s)
         end
    else if s = '+' then read(byla, s);
  while s in ['0'..'9'] do
    begin
      p := p*10 + (ord(s) - ord('0'));
      read(byla, s)
    end;
  if neigiamas then p := -p
end;
```

9.2.1.

```
function mažr (s : char) : char;
  const dr = 'AĄBCČDEĘĖFGHIYĮJKLMNOPQRSŠTUŲŪWVXZŽ';
        mr = 'aąbcčdeęėfghiyįjklmnopqrsštuųūwvxxž';
  var   nr: integer;          { raidės numeris }
        ss: string;
begin
  nr := pos(s, dr);
  if nr = 0 then ss := s
            else ss := copy(mr, nr, 1);
  mažr := ss[1]
end;
```

9.2.2.

Abiem atvejais – ne.

Literatūra

1. V. Tumasonis. Paskalis ir Turbo Paskalis 7.0. Vilnius: Ūkas, 1993.
2. V. Grabauskienė. Susipažinkime – transliatorius. Kaunas: Šviesa, 1994.
3. V. Dagienė, G. Grigas, K. Augutis. Šimtas programavimo uždavinių. Kaunas: Šviesa, 1986.
4. V. Dagienė, G. Grigas. Lietuvos jaunųjų programuotojų konkursai. Kaunas: Šviesa, 1994.
5. G. Grigas. Sveikieji ir realieji skaičiai. – Informatika, 1989, Nr. 13, p. 49–68.
6. G. Grigas. Kontrolinių pradinių duomenų parinkimas atsižvelgiant į funkcijos lūžio taškus. – Informatika, 1992, Nr. 22, p. 7–15.
7. V. Dagienė. Algoritmo teksto išdėstymas. – Informatika, 1995, Nr. 26, p. 29–47.
8. L. Paliulionienė. Šalutinis funkcijų efektas. – Informatika, 1994, Nr. 24, p. 22–28.
9. B. Burgis ir kt. Kompiuterika moksleiviams ir studentams. Kaunas: Technologija, 1993.