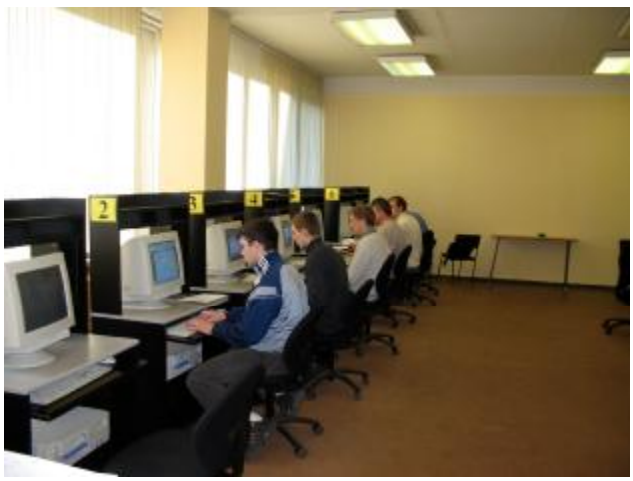
RĪGAS TEHNISKĀ UNIVERSITĀTE

Būvmehānikas katedra

PROGRAMMĒŠANAS PAMATI PASCAL VIDĒ

būvniecības specialitātes studentiem



Rīga, 2005

Publicējamais materiāls domāts Būvniecības fakultātes studentu izmantošanai, apgūstot programmēšanas valodu Pascal. Materiāls sniedz informāciju par Pascal valodas pamatnostādnēm, uzbūves struktūru, programmu sastādīšanas algoritmiem un principiem. Izklāstītais materiāls bagātīgi ilustrēts ar programmu piemēriem un to komentāriem. Veiksmīgas vielas apguves kontrolei studentam tiek piedāvāta virkne paškontroles uzdevumu ar atbildēm.

SATURS

IEVADS 4
ALGORITMIZĀCIJA 6
PASCAL VALODAS PAMATELEMENTI 9
DATU TIPI 10
STANDARTFUNKCIJAS 13
PROGRAMMAS STRUKTŪRA 14
- Apraksta daļa 15
- Operatoru daļa 16

OPERATORI 17
- Procedūru operatori 17
- Paškontroles uzdevumi 23
- Sazarojumu operatori 24
- Izvēles operators CASE ... OF ... END 26
- Paškontroles uzdevumi 28
- Cikla operatori 29
  - Operators FOR ... TO ... DO... 29
  - Operators WHILE ... DO 33
  - Operators REPEAT ... UNTIL... 35
- Paškontroles uzdevumi 36

MASĪVI 38
- Darbības ar masīviem 41
  - Masīvu aizpildīšana. 41
  - Masīvu aizpildīšanas piemēri 42
- Masīvu apstrāde 44
- Masīvu sakārtošana 46
  - Elementu izslēgšana un ievietošana masīvā. 50
- Paškontroles uzdevumi 51

FUNKCIJAS UN PROCEDŪRAS 52
- Paškontroles uzdevumi 58

GRAFISKAIS REŽĪMS GRAPH 62
- Programmas pāriešana grafiskajā režīmā 63

PIELIKUMI 73
- Pielikums I. Paškontroles uzdevumu atbildes. 73

Pielikums 2. Speciālas konstantes draivera tipu norādei. 77
Pielikums 3. Darbs ar programmu 78
Pielikums 4. Paziņojumu kodi par kļūdām Turbo Pascal valodas programmās 81
Literatūra. 81

IEVADS

XX g.s. 40-tajos gados notika principiāls pavērsiens skaitļošanas tehnikas attīstībā. Radās pirmās elektroniskās skaitļojamās mašīnas. Pirmā ESM tika radīta 1945. gada beigās ASV un tātad jau 60 gadus ilgst ESM tehnisko iespēju pilnveidošanas un attīstības periods. Straujā datoru attīstība un to plaša ieviešana dažādās saimniecības nozarēs radīja priekšnosacījumus jaunas zinātnes nozares – informātikas izveidei un attīstībai. Datori faktiski ir informācijas apstrādes darbarīki un to veiksmīga izmantošana lielā mērā saistīta ar datoru matemātiskā aprīkojuma kvalitāti. Tas aktualizē problēmas par informācijas apstrādes algoritmizāciju un skaitļojamo programmu izstrādi un pilnveidošanu.

Risinot konkrētus uzdevumus ar datoru palīdzību vispirms jāsastāda šī uzdevuma atrisināšanas algoritms, t.i. precīzs un nepārprotams priekšraksts jeb norādījums izpildītājam veikt kādu darbību virkni, lai sasniegtu nepieciešamo rezultātu.

Dators nevar atrisināt uzdevumu, kuru tā risinātājs pats neizprot un nevar noformulēt tā atrisināšanas algoritmu. Bet arī tad, ja šāds algoritms ir noformulēts, tas jāievada datorā tam saprotamā formā. Šāda forma ir programma, kas kopā ar izejas datiem tiek ierakstīta datora atmiņā. Diemžēl, datori, pat paši „gudrākie", nesaprot nevienu no cilvēku sazināšanās valodām. Līdz ar to rodas nepieciešamība izveidot „mākslīgu valodu", ko sauc par algoritmisko jeb programmēšanas valodu. Kā viena no pirmajām jāmin Basic valoda, kuras pirmais variants tika radīts 1964. gadā iesācējiem, vienkāršu programmu sastādīšanai. Eksistē simtiem šīs valodas versiju, kuras bieži vien maz savietojamas savā starpā. Basic guvis plašu pielietojumu mikrokalkulatoros. Šī valoda nav domāta lielu, sarežģītu programmu sastādīšanai. Plaši tiek lietotas Quick Basic (firma Microsoft), Turbo Basic (firma Borland) un Visual Basic valodas.

1950.-60 gados bija izstrādāta algoritmiskā valoda ALGOLs. Paskāls kļuva par ALGOLa principu pārmantotāju. **1970.gadā** Cīrihes Federālā Tehnoloģiskā institūta profesors **Niklavs Virts** studentu apmācīšanai programmēšanā **izstrādāja valodu Pascal.** Valodas nosaukums tika izvēlēts par godu franču matemātiķim un filozofam Blēzam Paskālam (1623 – 1662), kurš jau 17. gs. izvirzīja ideju par informācijas datu mehānisku apstrādi un ņēma dalību aritmometra izveidošanā un pilnveidošanā. Salīdzinot ar ALGOLu Pascals ir daudz vienkāršāks un skaidrāks. Tajā labākas iespējas datu apstrādei un ir iebūvētas ievades / izvades procedūras, kādu nebija ALGOLā. Šai valodā programmas viegli lasāmas un tās satur visus stingra programmēšanas stila ievērošanai nepieciešamos elementus. Sākotnēji šī valoda bija ar visai ierobežotām iespējām, bet laika gaitā tā tika papildināta un uzlabota, radās tās versijas. Firma Borland radīja versiju **Turbo Pascal** ar visai plašām iespējām. Šī versija ir augsta līmeņa strukturēta valoda, kas ļauj uzrakstīt jebkura garuma un grūtuma programmas.

Programmēšanas valodu **Pascal** var uzskatīt par universālu. Tai ir vienkārša konstrukcija, ir plašas iespējas izveidoto programmu pareizības kontrolei kā to

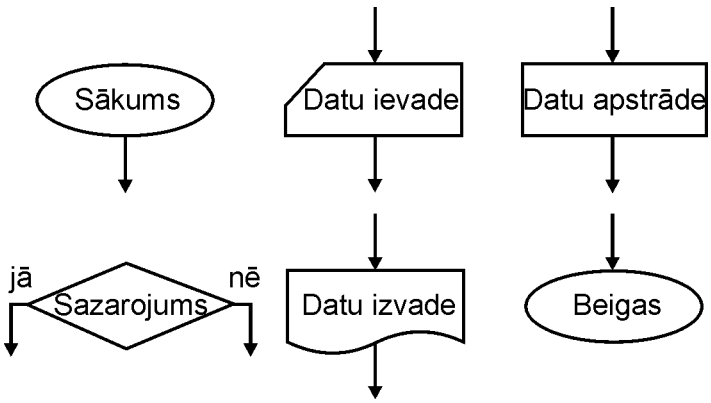
kompilācijas (programmas translācijas mašīnas kodu valodā), tā arī izpildes laikā. Valoda ietver sevī galvenos programmēšanas jēdzienus un konstrukcijas, tajā izmantoti strukturālās programmēšanas principi un tā pielietojama dažāda profila uzdevumu risināšanai, pie kam nezaudējot vienu no savām galvenajām īpašībām - relatīvu vienkāršību.

ALGORITMIZĀCIJA

Cilvēku rīcību ikdienas dzīvē lielā mērā regulē **instrukcijas**, t.i., iepriekš izstrādātas operācijas un to izpildes kārtība, kas ļauj sasniegt vēlamo rezultātu. Kā piemēru var minēt mobilā telefona ekspluatāciju. Tikai precīzi izpildot atbilstošas instrukcijas var sasniegt vēlamo rezultātu. Lai iegūtu matemātiska rakstura uzdevuma atrisinājumu, ir jāzina šī uzdevuma atrisināšanas algoritms.

Par **algoritmu var uzskatīt konkrētas secības darbību kopumu, kura pielietojums dotiem objektiem, nodrošina meklējamā rezultāta iegūšanu.**

Algoritma pierakstam jābūt sadalītam precīzos nošķirtos soļos, kur katrā solī ir paredzēts izpildīt vienu vienkāršu norādījumu. **Katru** šādu **norādījumu sauc par komandu**.



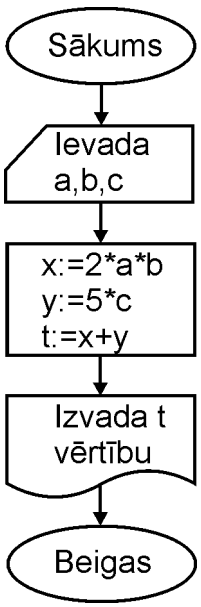
Sastādot, labojot un uzlabojot sarežģītākus algoritmus, to pieraksts teksta formā nav pietiekami uzskatāms un viegli lasāms. Tāpēc nereti programmēšanai paredzētos algoritmus veido ar grafisku elementu palīdzību.

Algoritma pieraksta grafiskā forma ir blokshēma, kura veidota no atsevišķiem tipizētiem elementiem. Attēlā redzami blokshēmās izmantojamie grafiskie elementi.

Vispārīgā gadījumā katra algoritma komanda sastāv no divām daļām:

izpildāmās darbības un *norādes* uz vietu algoritma pierakstā, kur atrodas nākamā izpildāmā komanda.

Strukturētā teksta pierakstā tiek pieņemts, ka pēc katras komandas izpildes algoritma izpildītājam jāpāriet pie komandas, kas atrodas nākamajā rindā, ja vien nav bijis norādījums izpildīt kādu citu komandu. Tāds pats pieņēmums ir spēkā arī **Pascal** programmās.



Blokshēmās uz katru nākamo izpildāmo komandu norāda bultiņa, kura "iziet" no izpildītās komandas. Katram **blokshēmas elementam var pienākt viena bultiņa** (izņēmums ir sākuma elements, kuram nepienāk neviena bultiņa). **No katra blokshēmas elementa iziet tikai viena bultiņa** (izņēmums ir beigu elements, no kura neiziet neviena bultiņa un sazarošanās elements, no kura iziet divas bultiņas). Algoritmus, kuru komandas tiek izpildītas tādā secībā, kādā tās pierakstītas, sauc par **lineāriem algoritmiem**. Blokshēmās lineāro algoritmu pierakstā nelieto sazarošanās elementus.

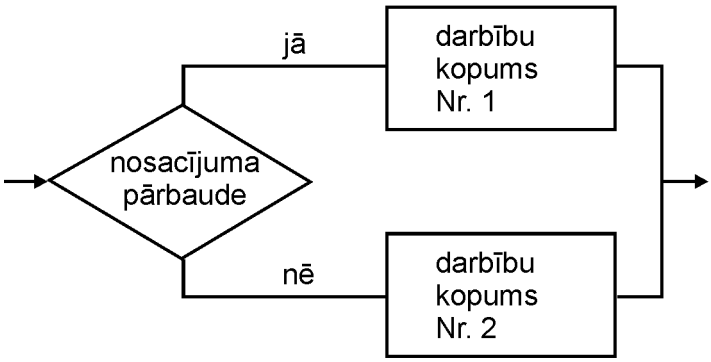
Parasti algoritmi netiek rakstīti vienam konkrētam gadījumam vai uzdevumam, bet gan veselai līdzīgu uzdevumu grupai. Algoritma blokshēmas piemērs, atbilstoši kurai nosaka funkcijas **t=2ab+5c** vērtību, redzams attēlā.

Bieži vien vienu un to pašu rezultātu var sasniegt pēc formas dažādos veidos. Formas izmaiņa dod iespēju meklēt racionālāko risinājuma variantu konkrētā uzdevuma atrisināšanai ar mērķi taupīt datora resursus un radīt ērtas programmas. Tā, piemēram, vidējo vērtību var noteikt kā visu lielumu algebriskās summas dalījumu ar summējamo lielumu skaitu, $a_v = (\Sigma a_i)/n$, vai arī summējot katra lieluma n-tās daļas $a_v = \Sigma a_i/n$. Otrajā gadījumā tiek veikta (n-1) papildus aritmētiska darbība.

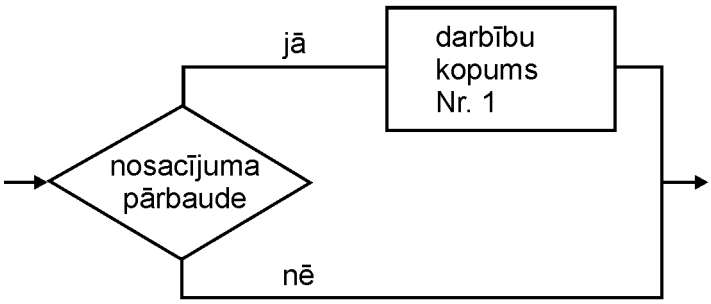
No algebras zināms, ka lielums $c = a^2 - b^2$ iegūstams gan kā $c = a \cdot a - b \cdot b$, gan kā $c = (a-b)(a+b)$. Šai piemērā darbību skaits abos gadījumos ir vienāds, atšķirīga ir izpildes secība.

Gadījumos, kad aprēķinu secība un saturs mainās atkarībā no iegūtajiem starprezultātiem, mēs nonākam pie sazarotas aprēķinu shēmas. Aprēķinu gaitā, nonākot pie kāda loģiska nosacījuma, tiek pārbaudīts, vai šis nosacījums izpildās, vai nē. Pozitīva rezultāta gadījumā uzdevuma risinājums tiek veikts izpildot vienas izskaitļošanas operācijas, bet pretējā gadījumā citas operācijas.

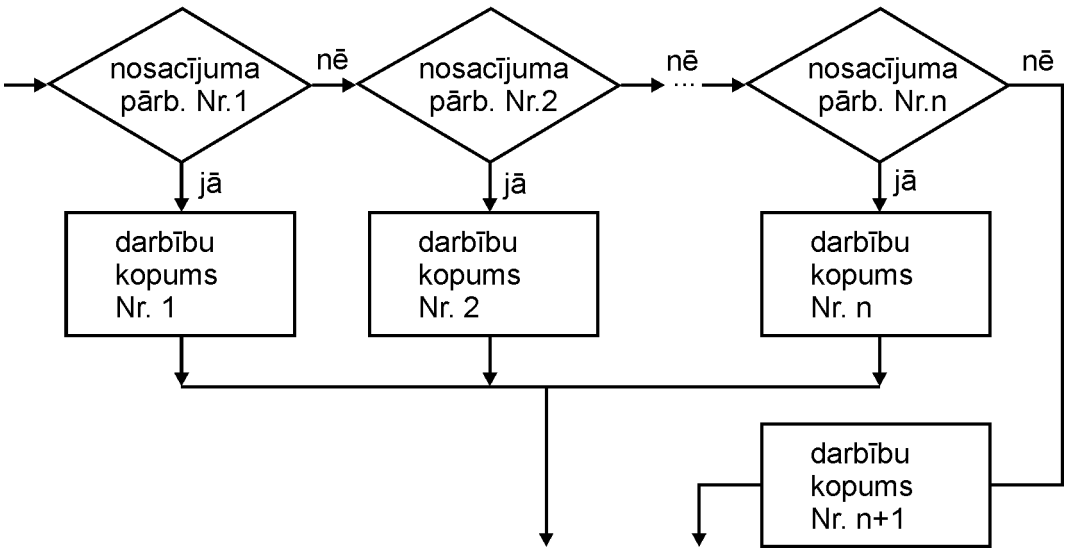
a)


jā
darbību kopums Nr. 1
nosacījuma pārbaude
nē
darbību kopums Nr. 2

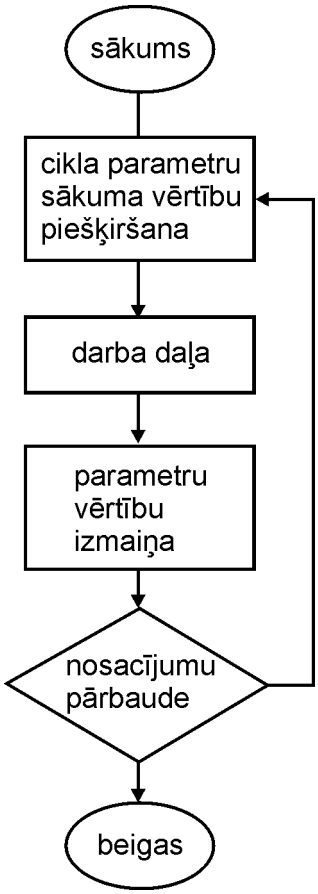
b)


jā
darbību kopums Nr. 1
nosacījuma pārbaude
nē

c)


nosacījuma pārb. Nr.1
nē
nosacījuma pārb. Nr.2
nē
...
nosacījuma pārb. Nr.n
nē
jā
jā
jā
darbību kopums Nr. 1
darbību kopums Nr. 2
darbību kopums Nr. n
darbību kopums Nr. n+1

Tāds gadījums parādīts blokshēmā **a).** Iespējams arī nepilnā sazarojuma gadījums, kad nosacījuma pārbaudes rezultātā tiek izpildīta tikai viena operātoru grupa. Šis gadījums attēlots blokshēmā **b).** Nav izslēgts gadījums, ka izpildot pirmā vai otrā veida operācijas atkal nonākam pie kāda cita loģiskā nosacījuma, kurš atkal regulē turpmāko aprēķinu gaitu. Attēlā **c)** parādīts viens no šāda veida iespējamiem gadījumiem.



Ciklisku aprēķinu gadījumā katra cikla izpildes gaitā izmainās cikla parametri un atkarībā no to vērtībām ciks tiek atkārtots vai arī pārtraukts. Šāds algoritms organizē neierobežotu ciklu skaitu. Tā blokshēma parādīta attēlā.

PASCAL VALODAS PAMATELEMENTI

Programmēšanas valodas pieraksta, līdzīgi kā sarunu valodas pieraksta, pamatelementi ir burti, no kuriem tiek veidoti vārdi. Vārdi veido teikumus, bet no teikumiem sastāv jebkurš teksts. Pascal valodas ‘’alfabētu’’ veido sekojoši simboli:

26 latīņu alfabēta burti (a, ...z);

10 arābu cipari (0,1,2,3,4,5,6,7,8,9);

pasvītrojuma zīme (_);

speciālie simboli.

Speciālie simboli

Attēls	Nosaukums	Attēls	Nosaukums	Attēls	Nosaukums
+	pluss	**#**	numura zīme	**[ ]**	kvadrātiekavas
-	mīnuss	**$**	dolāra zīme	**{ }**	figūriekavas
*	zvaigznīte	**=**	vienāds	**:=**	piešķiršana
/	slīpa svītra	**>**	lielāks	**. .**	divi punkti
.	punkts	**<**	mazāks	**(**)**	Iekava - zvaigznīte
,	komats	**>=**	lielāks vai vienāds	**(..)**	Iekava - punkts
:	kols	**<=**	mazāks vai vienāds	**@**	adreses zīme
;	semikols	**<>**	nevienāds		
‘	apostrofs	**( )**	apaļās iekavas		

Tikai no šiem simboliem tiek veidoti programmās izmantojamo objektu vārdi (identifikatori). **Identifikatori** tiek veidoti no burtiem, cipariem un pasvītrojuma zīmes **(bet ne speciāliem simboliem)**. Identifikatora garums nav ierobežots, bet to salīdzināšanai tiek ņemti vērā tikai pirmie 63 simboli. Identifikatora pirmajam simbolam jābūt burtam vai pasvītrojuma zīmei **(bet ne ciparam)**. Mazie un lielie burti **ir līdzvērtīgi**.

Turbo Pascal valodā ir vairāk kā **60 rezervētu vārdu**, kuri lietojami tikai tiem paredzētā nolūkā (sk. tab.). Turpmākajā izklāstā tiks atšifrēta konkrēto vārdu pielietojamība. Pagaidām aprobežosimies ar Pascal programmēšanas valodā visbiežāk izmantojamo rezervēto vārdu nozīmes skaidrojumu. Programmas nosaukums (virsraksts) seko vārdam ***program***, bet bibliotēkas moduļu aprakstu ievada vārds ***unit***. Mainīgo, konstanšu un programmas sastāvdaļu – apakšprogrammu procedūru un funkciju apzīmēšanai tiek izmantoti vārdi ***var, const, procedure, function***. Lietotāja uzdoto mainīgo tipu aprakstam izmanto ***type, array, string, record...end, file of...*** Salikto operātoru, kā arī programmas operātoru daļas sākumu un beigas iezīmē vārdi ***begin, end***. Programmas izpildes gaitu regulējošie operatori ir ***if...then...else, for .. .to .. .do, repeat ... until, case ... of ... end, for ... downto ... do, while ... do***. Bibliotēkas moduļos tiek izmantoti vārdi ***implementation, interfac.*** Aritmētisko un loģisko operāciju apzīmējumos tiek izmantoti vārdi ***div , mod , shl, shr, and, or, not*** un daži citi.

Pascal valodā rezervētie vārdi

absolute	**and**	**array**	**assembler**
begin	**break**	**case**	**const**
constructor	**continue**	**destructor**	**div**
do	**downto**	**else**	**end**
external	**far**	**file**	**for**
function	**goto**	**if**	**implementation**
in	**inline**	**interface**	**interrupt**
label	**mod**	**near**	**nil**
not	**object**	**of**	**or**
packed	**private**	**procedure**	**program**
public	**record**	**repeat**	**set**
shl	**shr**	**string**	**then**
to	**type**	**unit**	**until**,
uses	**var**	**virtual**	**while**
with	**xor**		

DATU TIPI

Pascal valodā izmantojamie dati iedalāmi trīs grupās: **skalārie**, **strukturētie** un **pārsūtāmie**.

Skalārie dati ir:

veselo skaitļu tips – tā atslēgas vārds ir **integer**;

reālo skaitļu tips - tā atslēgas vārds ir **real**;

loģiskais tips - tā atslēgas vārds ir **boolean**;

simbolu tips - tā atslēgas vārds ir **char**.

Veselo skaitļu tips. Turbo Pascal versijā ir **pieci** veselo skaitļu tipi. Šī grupa apvieno veselo skaitļu kopumu dažādos diapazonos:

Tips	Diapazons	Aizņemtās atmiņas apjoms
shortint	-128...127	1 baits
integer	-32768...32767	2 baiti
longint	-2147483648...2147483647	4 baiti
byte	0...255	1 baits
word	0...65535	2 baiti

Ar šiem veselo skaitļu tipa mainīgajiem var veikt aritmētiskas operācijas **+, -, *, / , div**, **mod**. Šo operāciju rezultātā iegūstam arī veselus skaitļus, izņemot dalīšanu (**/**), kuras rezultātā iegūstam racionālu skaitli.

Operācija **div** (no vārda *division* – dalīšana) ir dalīšana bez atlikuma (atlikums tiek atmests), bet operācija **mod** (*modulus* **–** mērs) saglabā tikai divu veselus skaitļu dalījuma atlikumu. Operācija a **mod** b izpildīsies tikai tai gadījumā, ja b>0.

Piemēri: ja 7/3=2,3(3), tad 7 **div** 3=2, bet 7 **mod** 3=1;

2 **div** 7=0; (-7) **div** 2=-3; 7 **div** (-2) = -3;

(-7) **div** (-2)=3; 23 **mod** 12=11.

Veselā skaitļa tipam izmantojamas divas iebūvētās procedūras:

dec(x,n) - samazina x vērtību par n, ja n nav noteikts, tad par 1.

inc(x,n) - palielina x vērtību par n, ja n nav noteikts, tad par 1.

Izmantojot **mod** dalīšanu, varam spriest par dota skaitļa dalāmību. Tā kā 9 mod 3 rezultāts ir nulle, tad dotais skaitlis dalās ar skaitli 3 bez atlikuma. Dalot (**mod**) jebkuru skaitli ar 2, nosakām, vai šis skaitlis ir pāra, vai nepāra skaitlis. Šāds paņēmiens plaši tiek lietots programmu izveidē.

Jāņem vērā, ka gadījumos kad veicamo darbību rezultātā iegūstam skaitli, kurš neatbilst norādītajam diapazonam, kļūda netiek uzrādīta un šī skaitļa vietā tiek izvadīts kāds cits (kļūdains) skaitlis.

Reālo skaitļu tipi. Turbo Pascal-ā ir pieci reālo skaitļu tipi. Šo tipu skaitļiem atbilst reālā skaitļa jēdziens matemātikā:

Tips	Diapazons	Ciparu skaits mantisā	Aizņemtās atmiņas apjoms
real	± (2.9 E-39 ... 1.7 E38) (jeb $2{,}9\cdot10^{-39}...1{,}7\cdot10^{38}$)	11-12	6 baiti

single	± (1.5 E-45...1.7 E38) (jeb $1,5 \cdot 10^{-45}$...$1,7 \cdot 10^{38}$)	7-8	4 baiti
double	± (5.0 E-324...1.7 E308) (jeb $5,0 \cdot 10^{-324}$...$1,7 \cdot 10^{308}$)	15-16	8 baiti
extended	± (3.4 E-4932...1.1 E4932) (jeb $3,4 \cdot 10^{-4932}$...$1,1 \cdot 10^{4932}$)	19-20	10 baiti
comp	± 9.2 E+18 (jeb $9.2 \cdot 10^{18}$)	19-20	8 baiti

Comp faktiski ir veselo skaitļu tips ar palielinātu diapazonu.

Reālā tipa skaitļu pierakstam iespējamas divas pieraksta formas:

- **ar fiksētu komatu** (punktu), piem., 16.1346495 , 0.4395847, 2.5000000;
- **ar peldošu komatu** (eksponentforma), piemēram, 3.5E-11, -1E+3 (šo skaitļu vērtības attiecīgi ir $3.5 \cdot 10^{-12}$ un -1000).

Ar reāliem skaitļiem var izpildīt visas tās pašas aritmētiskās darbības, kā ar veseliem skaitļiem (izņemot darbības **mod** un **div**), tikai rezultāts būs reāls skaitlis.

Gadījumos, ja izpildot darbības rezultāts pārsniedz pieļaujamo augšējo robežu, tad tiek uzrādīta kļūda, bet, ja rezultāts ir mazāks par diapazona apakšējo robežu, attiecīgajam identifikatoram tiek piešķirta nulles vērtība.

Simbolu virkne. To pārstāv datu tips **string**, kas paredzēts simbolu tabulas elementu virknes uzglabāšanai un apstrādei. Maksimālais simbolu virknes elementu skaits ir 255. Simbolu virkne jāiekļauj apostrofos: 'Anna', 'X1', 'Y=7'.

Simbolu tips. Šī tipa vērtības ir galīga skaita sakārtotas simbolu tabulas elementi. **Simbolu tipa** (**char**) lielumi var būt jebkurš simbols, kurš atrodas datora simbolu kodu tabulā. Visi simboli sakārtoti augošā secībā pēc to kodiem, kas mainās no 0 līdz 255. Piem., **chr**(65) ir burts **A**, bet **chr**(66) - burts **B**.

Simboli, kas ieslēgti apostrofos ir simboliskā tipa konstantes, piem., ‘x’, ’6’, ’darbs’.

Simbolu tipam eksistē divas savstarpēji apgrieztas standartfunkcijas:

ord(c) - piešķir simbola **c** kārtas numuru simbolu tabulā;

chr(i) - piešķir simbolu, kuram atbilst kārtas numurs **i**.

Tā, piem., **ord**(A) = 65, bet **chr**(65) =’A’.

Loģiskā tipa (**boolean**) lielumiem ir iespējamas tikai divas vērtības: **true** (patiess) un **false** (aplams).

Ar tām var izpildīt loģiskās operācijas. Biežāk lietojamās ir: **and**, **or**, **not**. Loģiskā tipa funkcija ir **odd**(x). Šīs funkcijas vērtība ir **true**, ja x ir nepāru skaitlis un **false** - pretējā gadījumā.

Atvasinātie un uzskaitāmie tipi. Aprakstot uzskaitāmo tipu, programmētājs uzdod (uzskaita) visas tās vērtības, kuras mainīgajam varēs piešķirt, ja tas būs aprakstīts kā šī tipa mainīgais. Uzskaitāmā tipa mainīgie programmā var pieņemt tikai tādas vērtības, kuras uzskaitītas vērtību sarakstā.

Piemēram: **type**

Vasara= ('Junijs', 'Julijs', 'Augusts');

Uzskaitāmajam tipam pieļaujamas tikai salīdzināšanas operācijas un tam pielietojamas standartfunkcijas:

succ(x) -> uzdod aiz simbola x sekojošu simbolu no vērtību saraksta;

pred(x) -> uzdod pirms simbola x iepriekšējo simbolu no vērtību saraksta;

ord(x) -> uzdod simbola x kārtas numuru konkrētā simbolu kopā.

Intervāla tips. Jebkuram skalāram lielumam, izņemot daļskaitļus, var izveidot jaunu skalāru lielumu ar vērtību ierobežojumu kādā intervālā. Intervāla noteikšana paaugstina programmas drošību, ekonomē atmiņu, kā arī programmas izpildes laikā kontrolē piešķiršanas operācijas. Intervālu var noteikt:

1) tipu apraksta daļā:

type <tipa nosaukums>=<konstante1> .. <konstante2>;

2) mainīgo apraksta daļā:

var <mainīgais>:<konstante1>..<konstante2>;

kur <konstante1> - apakšējā robeža;

<konstante2> - augšējā robeža.

Piemēram:
```
type  Burts='A'..'Z';
      Int=1..100;
var   i,j,k: Int;
      s,m: Burts;
```

Lietojot programmā standarta datu tipus, tie nav speciāli jādefinē, bet programista izveidotie tipi jāuzrāda pēc vārda **type**.

Piemēram:
```
type  Diena =('Sestdiena', 'Svetdiena');
      Menesis=('Julijs', 'Augusts');
```

Tipi Diena un Menesis ir jauni tipi, bet vārdi ('Sestdiena', 'Svetdiena', 'Julijs', 'Augusts') pieder pie standarta tipa **string**.

STANDARTFUNKCIJAS

Pascal valodas kompilatorā iebūvēto funkciju skaits nav liels. Biežāk lietojamās sakopotas tabulā.

Standartfunkcijas

Funkcijas nosaukums	**Funkcijas nozīme**	**Tips**	
		argumentam	funkcijai
abs(x)	$\|x\|$	**integer**	**integer**
sqr(x)	x^2	**real**	**real**
sin(x)	sinx	**integer**	**real**

cos(x) **arctan**(x) **exp**(x) **ln**(x) **sqrt**(x)	cosx arctgx e^x lnx $\sqrt{x}$	vai **real**	
trunc(x) **round**(x)	[x] –skaitļa veselā daļa noapaļošana līdz veselam	**real**	**integer**
odd(x)	nepārības pārbaude	**integer**	**boolean**

Trigonometrisko funkciju arguments **jāuzdod radiānos**. Gadījumos, kad arguments dots loka grādos, jāizmanto pārejas sakarība **x=(x^o)*pi/180**.

Logaritmu ar patvaļīgu bāzi **a** (nevis **e**) noteikšanai var izmantot sakarību **$\log_a(x)$ = ln(x)/ln(a).**

Jāņem vērā, ka Turbo Pascal-ā kāpināšana tiek realizēta izmantojot logaritmu īpašību: ja c = a^b, tad **ln**(c) = b**ln**(a) un c = **exp**(b**ln**(a)). Līdz ar to nav iespējams kāpināt negatīvu skaitli. Šim nolūkam lietderīgi izmantot cikla operatorus.

Gadījuma skaitļu ģenerēšana ar funkciju RANDOM. Daudzām parādībām dabā, tehnikā, ekonomikā un citās jomās ir gadījuma raksturs. Piemēram, metot metamo kauliņu, iepriekš nav iespējams paredzēt kāds skaitlis uzkritīs. Šādu parādību realizēšanai datorā var izmantot gadījuma skaitļu funkciju **random**, t.i., funkciju, kura izvada programmas "iedomātu" skaitli. Šo funkciju var izmantot arī gadījumos, kad programmas darbības pārbaudei nepieciešamas daudzas gadījuma rakstura skaitliskas vērtības.

Pieraksts **random**(n) ģenerē gadījuma skaitli no intervāla

0 <= random(n) < (n -1).

Gadījumos, kad **n** nav norādīts (izmantojam pierakstu **random** bez argumenta), tiek ģenerēts skaitlis no intervāla **(0,1),** tātad decimāldaļskaitlis.

Izpildot gadījuma skaitļu ģenerēšanu **vairākkārt**, lai nodrošinātu citu gadījuma skaitļu izvēli, tiek lietots operators **randomize.** Šis operators programmā izpildāms pirms funkcijas **random.**

PROGRAMMAS STRUKTŪRA

Risinot uzdevumus ar datoru palīdzību ir nepieciešams sastādīt programmas, kuras dotu iespēju datoram saprotamā veidā realizēt atbilstošo aprēķinu algoritmu. Jebkura programma satur mērķtiecīgā secībā izvietotu operatoru kopumu. Katrai programmai jāsatur vismaz viens izpildāmais operators, pretējā gadījumā tai nav jēgas.

Pascal valodas programmas struktūra parasti veidota trīs daļās:

program <nosaukums>**;**	{Turbo Pascal-ā nosaukums **var arī nebūt}**
<apraksta daļa>**;** vai datu apraksts	{tiek uzskaitītas visas turpmāk izmantojamās konstantes, iezīmes, mainīgie, apakšprogrammas un citi objekti }

begin <operatoru daļa> vai darbību apraksts **end.**	{darbību operatori tiek izvietoti to izpildīšanas secībā}

Programmas nosaukums nav obligāts. Vienkāršāku programmu gadījumos var nebūt arī <apraksta daļa> un tad programma sastāv tikai no <operatoru daļas>. Svarīga programmu noformēšanas sastāvdaļa ir komentārs teksta veidā, kurš ievietojams figūriekavās { } vai iekavās ar zvaigznītēm (* *). Komentārs palīdz izprast programmas tekstu un atvieglo tās lasīšanu citam speciālistam. **Komentārs nepiedalās programmas izpildē.**

Apraksta daļa

Šai daļā jābūt uzskaitītiem visiem <operatoru daļā> izmantojamajiem identifikatoriem, iezīmēm, funkcijām un procedūrām. Identifikatoriem tiek norādīts to tips. Apraksta daļa sastāv no sešām iespējamām pozīcijām. Tiek uzrādītas tikai tās pozīcijas, kuras ir nepieciešamas konkrētajā programmā. Ne vienmēr visas no sešām iespējamajām pozīcijām tiek izmantotas. Var būt arī gadījumi, kad apraksta daļa ir „tukša", t.i. tā nesatur nevienu no pozīcijām.

Šīs pozīcijas ir:

programmā pielietojamo bibliotēkas **moduļu** saraksts (programmā apzīmē ar atslēgvārdu **uses**);

iezīmju saraksts (**label**);

konstanšu apraksts (**const**);

no standartmoduļiem atšķirīgu **papildtipu** definēšana (**type**);

mainīgo apraksts (**var**);

procedūru un **funkciju** apraksts (**procedure**, **function**).

Apraksta pozīciju secība ir viennozīmīga (tā jāievēro):

1. Gadījumos, kad sastādāmajā programmā tiks izmantots kāds no **Pascal** valodas bibliotēkas standartmoduļiem, tas apraksta daļā jāuzrāda.

Piem.: **uses crt**, **graph**, u.c.

Var tikt veidoti arī nestandarta moduļi un turpmākajā programmēšanas daļā lietoti kā bibliotēkas sastāvdaļas.

2. Gadījumos, kad programmas operatoru daļā tiks veikta operatoru izpildes secības maiņa ar operatora **goto** palīdzību izmantojot iezīmes, tās jāuzrāda iezīmju daļā pēc atslēgvārda **label**. Iezīmes parasti ir naturālo skaitļu veidā, bet var būt arī teksta veidā.

Piem.: **label** 1, 5, 172, m1, stop;

Operatoru izpildes secības maiņas vietā pēc operatora **goto** tiek norādīta attiecīgā iezīme. Šo pašu iezīmi norāda pirms nākamā izpildāmā operatora to atdalot ar kolu.

Piem.: **goto** 5;

.....

5 : **write**(‘iezime ir 5’);

3. Gadījumos, kad programmā ir izmantojami konstanti lielumi, tie tiek uzrādīti pēc atslēgvārda **const**. Skaitliskās konstantes var būt veseli vai racionāli skaitļi. Racionālus skaitļus var pierakstīt divos veidos – ar **fiksētu vai peldošu punktu** (komatu). Simbolveida konstante tiek ieslēgta apostrofos. Loģiskām konstantēm iespējamas divas vērtības - **true** un **false**.

Piem.: **const** a =12; b=-20; c=5.34; d=**false**; e=**true**;

f=’Latvija’; vidvecums=35; fonds=1.5E+04; max=60;

4. Gadījumos, kad programmā nepieciešams izmantot tipus, kuri nav Pascal valodas standarttipi, tos definē pozīcijā aiz atslēgvārda **type**.

Piem.: **type** diena=(pirmd, otrd, ... svetd);

Vards=(aija, maija, gatis, sandris);

5. Aiz atslēgvārda **var** jāuzskaita **visi** mainīgie, kuri tiks izmantoti konkrētajā programmā un jāuzrāda to tips.

Piem.: **var** x, y, suma: **real;**

i, n: **integer**;

atbilde: **boolean**;

otrd: **diena**;

gatis, maija: **vards;**

6. Pēc atslēgvārda **procedure**, **function** tiek nosauktas un aprakstītas visas procedūras un definētas funkcionālās sakarības, norādot to skaitlisko vērtību noteikšanas algoritmus. Procedūras ir neatkarīgas programmas daļas, kuras veic konkrētas darbības. To struktūra ir analoga programmas struktūrai un tās var uzskatīt par mini programmām, kuras nepieciešamajā vietā tiek izsauktas ar to nosaukumiem (vārdiem). Piemērus skaties turpmāk.

Operatoru daļa

Tā ir programmas pamatdaļa un tajā tiek norādītas izpildāmās darbības to izpildīšanas secībā. **Operators ir sintaktiska konstrukcija** - pabeigta programmēšanas valodas frāze, **kas raksturo konkrētu datu apstrādes etapu.** Katrai komandai algoritma pierakstā atbilst konkrēts operators programmā. Operatoru skaits programmā nav ierobežots. Operatorus vienu no otra atdala ar semikolu, tātad **katra operatora beigās jāliek semikols (;)**. Izņēmums ir programmas beigas. Programmai vienmēr jābeidzas ar punktu.

Operatoru izpildes secības organizēšanai izmanto operatoru iekavas, t.i., vārdus **begin** un **end**. Tie lietojami tikai pārī. Tas nozīmē, ka pārbaudot programmu, jāseko tam, lai līdzīgi kā algebrisko iekavu gadījumā, atverošo iekavu skaits būtu vienāds ar aizverošo iekavu skaitu. Kā jau tika minēts, programmas lasīšanas atvieglojumam tiek izmantoti komentāri brīva teksta veidā to ieslēdzot figūriekavās { } vai (* *). Iekavās var tikt iekļauti arī veseli programmu fragmenti, tā izveidojot pamatprogrammas vienkāršotu variantu. Šādi ieslēgumi neietekmē programmas izpildes gaitu.

OPERATORI

Programmas operatoru daļa sastāv no operatoriem, kuri var būt tukši, vienkārši un salikti. Operatorus vienu no otra atdala ar semikolu. Gadījumos ja operators atrodas pirms vārda **else**, tad aiz tā semikolu neliek. **Tukšajā operatorā** nav neviena simbola un tas nenosaka nevienu darbību. **Vienkārši** ir **operatori**, kuru sastāvā nav iekļauti citi operatori. Tādi operatori ir piešķires operators, datu ievades un izvades procedūru operatori un pārejas operatori. **Saliktie operatori** ir konstrukcijas, kuras sastāv no citiem (vienkāršiem) operatoriem. Šādas konstrukcijas tiek iekļautas operatoru iekavās **begin ..end;**.

Piešķires operators ir fundamentāls un uzskatāms par pamatoperatoru. Šī operatora simbolisks apzīmējums ir specifisks simbolu sakopojums **:=** un tiek lietots starp mainīgā identifikatoru un šim mainīgajam piešķiramo vērtību, kura var tikt uzdota skaitliskā vai izteiksmju veidā.

Piem.: **a1:= 2.5; y:= x/(1-x); f:= 3*c + 2*sin(x+pi/2); k:=n>m;**

Svarīgi izsekot tam, lai izteiksmes vērtība labajā pusē atbilstu kreisā pusē norādītā mainīgā tipam. Tomēr ir pieļaujams **real** tipa mainīgajam piešķirt **integer** tipa izteiksmi.

Izteiksmju veids viennozīmīgi nosaka darbību izpildes secību: tās izpildās no kreisās uz labo pusi ņemot vērā to prioritāšu secību (vispirms izpildās **<, >, =, >=, =**, tad *** , /, div, mod** un kā pēdējās **+, -**)

Procedūru operatori

Ievades un izvades operatori. Izejas datu ievadei izmanto ievades operatoru **read**, aiz kura iekavās tiek uzrādīti ievadāmo lielumu identifikatori. Rezultātu izvadei izmanto izvades operatoru **write**, aiz kura iekavās tiek uzrādīti izvadāmo lielumu identifikatori vai izteiksmes, kuru vērtības jāizvada.

Piemēram,

operators **read**(x) pieprasa ievadīt parametra x vērtību;

operators **read**(x,y,z) pieprasa ievadīt trīs parametru x,y un z vērtības.

Ievadi veic ar klaviatūras taustiņu palīdzību, pie kam programmu nevarēs palaist, ja nebūs ievadītas **visu uzrādīto identifikatoru** vērtības.

Izvades operators **write**(s) izvada (uz ekrāna) lieluma s vērtību. Lai izvadītu arī tekstu, jāizmanto pieraksts **write**('s=',s). Lietojot pierakstu **write**(s,p,(s+p)/2) tiks izvadīti trīs vērtības, no kurām pirmā būs lieluma s vērtība, otrā būs lieluma p vērtība, bet trešā būs šo lielumu vidējā vērtība.

Operatoru **read** un **write** vietā var lietot operatorus **readln** un **writeln**, bet tādā gadījumā pēc katra operatora izpildes kursors pārvietojas uz jaunu rindu – rezultāti tiek uzrādīti kolonas veidā. Operators **readln;** bez operanda, t.i. ievadāmā parametra, **veido pauzi,** kura turpinās tik ilgi, kamēr tiek nospiests taustiņš **Enter**. Šādu paņēmienu izmanto programmu beigās, lai saglabātu redzamus iegūtos rezultātus (pretējā gadījumā tūlīt pēc programmas beigām ekrānu aizsedz logs ar programmas tekstu). Operators w**riteln;** bez izvadāmo parametru saraksta pārceļ kursoru uz nākamās rindas sākumu. Tādā veidā iespējams, piemēram, atdalīt vienu no otra programmas rezultātus ar vienu vai vairākām tukšām rindām. Izvades

operatora **Piemēri**:

Gadījumā, ja s=2, bet p=5, operators

write(s,p)	izvada 25
write(‘s= ‘,s,’p=’,p)	izvada s=2p=5
write(s,’ ‘,p)	izvada 2 5
write(‘s=‘,s,’ ‘,’p=’,p)	izvada s=2 p=5

Gadījumā, ja izvadāmais rezultāts ir reāls skaitlis, lai ierobežotu ciparu skaitu aiz komata, tiek uzdots **skaitļa formāts**. To veic sekojošā veidā **write**(s:m:n), kur m ir izvadāmā skaitļa kopējais ciparu skaits, bet n – ciparu skaits aiz komata. Veselu skaitļu gadījumā norādāms tikai ciparu skaits (m). Jāņem vērā, ka tais gadījumos, kad programmas lietotājs ir izvēlējies pārāk mazu **m** vērtību, izvadāmā parametra formāts automātiski tiek palielināts. Gadījumos, kad n ir par mazu, notiek noapaļošana. Izvadāmais teksts tiek piespiests labajai izvades lauka malai. Tā, piemēram, a:=4.321; write(‘a=’:10, a:7:3); izvadīs uz ekrāna:

xxxxxxxxa=xx4.321 (x – tukša pozīcija).

Beznosacījuma pārejas operators GOTO. Operatoru izpildes secību programmā var mainīt, lietojot beznosacījuma pārejas operatoru **goto**, aiz kura jāraksta iezīme. Jāatceras, ka iezīme jādefinē programmas apraksta daļā **label**.

Gadījumā, ja programmā, kura aprēķina dota skaitļa **x** kvadrātu, nepieciešams lietotājam dot iespēju izvēlēties, darbu beigt vai turpināt, lietderīgi izmantot operatoru **goto**. Šādu iespēju realizē programma:

Program kvadrats;	
label 1;	{tiek definēta iezīme ar nosaukumu **1**}
var x, y: **real**;	{tiek definēti mainīgie}
a: **char**;	
begin	
1: **write**(‘Ievadi x vertibu: ’);	{izvada paziņojumu monitorā}
readln(x);	{tiek ievadīts mainīgais **x**}
y:=**sqr**(x);	{mainīgais **y** iegūst vērtību $\mathbf{x^2}$}
writeln(‘x kvadrātā ir’, y);	{izvada paziņojumu un **y** vērtību }
writeln(‘Darbu turpināsim? (J/N)’);	{izvada paziņojumu monitorā}
readln(a);	
if a=’j’ **then goto** 1;	{mainīgā **a** ievadīšana no klaviatūras}
end.	{pāreja uz iezīmi **1**, ja lietotājs ir piespiedis **'j'**}

Piezīme: Nav ieteicams lietot daudzus **goto** operatorus vienā programmā. Tas apgrūtina programmas lasīšanu. Operatora **goto** pielietojums neatbilst labam programmēšanas stilam.

Procedūra gotoXY. Operatori **read** un **write** izvada informāciju displeja ekrāna vietā, uz kuru norāda kursors. Programmas izpildei sākoties, kursors vienmēr atrodas displeja ekrāna kreisajā augšējā stūrī. Pēc operatora **write** izpildes kursors paliek nākamajā pozīcijā aiz tikko izvadītā simbola, bet pēc operatora **writeln** izpildes kursors tiek pārnests uz nākamās rindas sākumu. Tātad, izvadot informāciju displeja ekrānā, kursors pakāpeniski pārvietojas pa kreisi un uz leju.

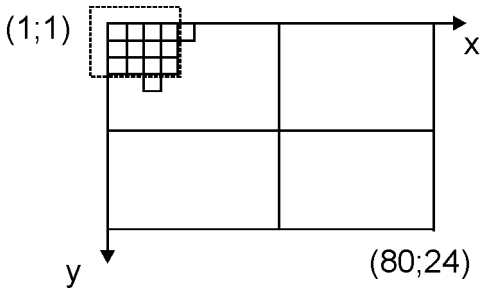
Gadījumos, kad nepieciešams novietot kursoru brīvi izvēlētā ekrāna punktā, jālieto procedūra **gotoXY**. Procedūra **gotoXY** novieto kursoru punktā (x,y).

Vispārējā pieraksta forma ir:

gotoXY (<mainīgais_x>,<mainīgais_y>),

kur: <mainīgais_x> - kursora koordināte pa horizontāli (kolonnas numurs); <mainīgais_y> - kursora koordināte pa vertikāli (rindas numurs).

Var uzskatīt, ka ekrāns sadalīts neredzamās rūtiņās, kur katrā rūtiņā vienlaikus var ierakstīt vienu simbolu. Rūtiņas ekrānā izvietotas 24 rindās, pie kam katrā rindā ir pa 80 rūtiņām. Rūtiņu numerācija tiek veikta no kreisās uz labo pusi un no augšas uz leju. Tādējādi, ekrāna augšējās kreisās rūtiņas koordināte ir (1,1), bet labās apakšējās rūtiņas koordināte ir (80,24). Ekrāna kreisā stūra palielinājums ir:



(1;1)	(2;1)		
(1;2)	(2;2)	(3;2)	
	(2;3)		

program students; **uses crt**; **begin** **ClrScr**; **GotoXY** (20,10); **writeln** ('Es esmu RTU students!'); **end**.	Izmantojot šo programmu uz displeja ekrāna tiek izvadīts teksts ‘’Es esmu RTU students!’’ sākot ar displeja ekrānā 10. rindas 20. pozīciju.

Procedūra ClrScr. Procedūra **ClrScr** attīra monitora ekrānu un novieto kursoru ekrāna kreisajā augšējā stūrī. To parasti lieto programmas operatoru daļā, tūlīt aiz sākuma iekavas **begin.** Tā rezultātā pēc atkārtotas programmas aktivizēšanas nav redzami iepriekšējā aktivizācijas reizē ievadītie un izvadītie dati. Procedūras **ClrScr** izmantošana iespējama tikai gadījumos, kad programmas apraksta daļā **uses** ir uzrādīts Pascal bibliotēkas modulis **Crt**.

Tukšais operators. Šādi operatori nesatur simbolus un neizpilda darbības. Tie nereti tiek veidoti ar iezīmi, lai izietu no salikta operatora vai programmas. Piemēram:

Begin

Goto <iezīme>; {pāreja uz programmas beigām}

<iezīme>: {tukšais operators}

end.

Programmas operatoru daļā lieks semikols nerada kļūdainu situāciju. Tā, piemēram, x:=1**;;** y:=2; interpretējams sekojošā veidā: tiek izpildīts piešķires operators x:=1**;** pēc tam seko tukšais operators un tad vēl viens piešķires operators y:=2;

Apraksta daļā dubults semikols nav pieļaujams, tas izraisa kompilācijas kļūdu.

Vienkāršāko programmu piemēri

Vienkāršākās programmas piemērs Pascalā ir programma:

Programmas	**Komentāri**
program pirmaa; **begin** **end**.	{virsraksts} {operatoru bloks}

Šī ir visvienkāršākā iespējamā programma, bet reālas darbības tā neveic.Tajā nav paredzēts ne datu ievads, ne izvads. Nav arī aritmētisku izteiksmju. Programma vienīgi demonstrē programmas struktūru.

1. Aprēķināt divu brīvi izvēlētu skaitļu a un b summu un noteikt to vidējo vērtību!

Šādas operācijas veikšanai sastādīsim programmu vairākos variantos:

Gadījumā, ja a un b ir veseli skaitļi, piem., a=271 un b=329, iespējami šādi neatkarīgi programmu varianti:

program summa1; **begin** **write**(271+329,' ', (271+329)/2) **end**.	 {tiek izvadīta doto skaitļu summa un aiz atstarpes to vidējā vērtība}
program summa2; **const** a=271; b=329; **begin** **write**(a+b,' vid.vert=', a/2+b/2) **end**.	{cits pieraksta veids} {tiek izvadīta doto skaitļu summa, teksts un vidējā vērtība}
program summa3; **var** a,b,c:**integer**; **begin**	{programmas vispārinātāks variants} {skaitļi var tikt definēt arī kā **real**}

read(a); **read**(b); c:=a+b; **write**(c, ‘ v.v=’,c/2) **end**.	{no klaviatūras ievada divus patvaļīgus skaitļus **a** un **b**} {mainīgajam **c** tiek piešķirta vērtība} {izvada prasītos rezultātus}

Svarīgi ņemt vērā, ka ar šāda tipa programmām iespējams veikt virkni visai ietilpīgu aprēķinu. Tā, piem., divu skaitļu summas vietā ievietojot daudz sarežģītāku aritmētisku izteiksmi, iespējams noteikt tās vērtību. Programmu varam izmantot trigonometrisku funkciju vērtību noteikšanai patvaļīga argumenta gadījumā, kā arī grafisku un tekstisku simbolu attēlošanai.

2. Noteikt pirmo n skaitļu summu un pārbaudīt vai tā ir vienāda ar S = n(n+1)/2.

program summa_n; **begin** **write**(1+2+3+4+5+6+7= 7(7+1)/2) **end**.	{izvēlamies patvaļīgu skaitli, mūsu gadījumā tas ir 7} {tiks veiktas attiecīgās operācijas un izvadīts paziņojums par to, vai vienādība izpildās }

Pārbaudot izvirzīto apgalvojumu par to, ka $\sum n = n(n+1)/2$ pie dažādām n vērtībām varam izdarīt secinājumu, ka vienādības labā puse izmantojama patvaļīga skaita naturālu skaitļu summas noteikšanai. Analoga uzdevuma atrisināšanai turpmāk tiks izmantots arī cits variants.

3. Ievadīt divus gadījuma skaitļus a un b. Noskaidrot, vai skaitlis a ir mazāks par skaitli b, vai šie skaitļi ir pāra vai nepāra skaitļi un kāda ir to summa.. Izvadīt iegūtos skaitļus.

program gadijumskaitli;
var a,b:**integer;**
 	
begin
randomize;
a:= **random**(100);
b:= **random**(100);
write(a<b);
 	
writeln(a mod 2 = 0, ’ ‘,b mod 2 = 0, ‘ summa ir ‘,a + b);
Writhe(a,’ ,b)
end.
	{gadījuma skaitļus iespējams definēt tikai kā veselus}
	
	{operators **random** ievada divus gadījuma skaitļus no intervāla [0,100)
	{tiek veikta skaitļu salīdzināšana un izvadīts attiecīgs rezultāts – TRUE vai FALSE}
	{tiek izvadīti attiecīgie rezultāti}
	
	{ tiek izvadītas skaitļu **a** un **b** vērtības}

4. Atbilstoši simbolu kodiem izvadīt uz ekrāna pašus simbolus.

```
program charmak1;
var   x:byte;
       y:char;
begin
      readln(x);
```
{definē **x** no intervāla (0,255) un simbolus no simbolu tabulas)}

{ievada skaitli **x**}

```
      y:= chr(x);

   write('atbilst  ');
writeln(y)
end.
```
{operators **chr** piekārto mainīgajam **y** skaitlim **x** atbilstošo simbolu }

{tiek izvadīts teksts un iegūtais simbols}

```
program charmak2;
label 1;
var  x:byte;
      y:char;
begin
1: readln(x);
y:= chr(x);
writeln('skaitlim  ',x,
'  atbilst simbols     ',y);
goto 1
end.
```
{tiek definēta iezīme **1**}

{ievada skaitļus **x** un tiem piekārt0tie simboli tiek piešķirti parametram **y**}

{izvada skaitlim **x** atbilstošo simbolu}

{programmas izpilde tiek pārnesta uz iezīmi **1** jauna skaitļa ievadīšanai}

Izmantojot programmā *kvadrats* pielietoto konstrukciju darba turpināšanai papildināt programmu *charmak2* tā, lai būtu iespēja pārtraukt programmu jebkurā brīdī.

5. Izveidot taisnstūra figūru, izmantojot kādu no simboliem.

```
program taisnsturis;
var  x:byte;
      y:char;
begin
read(x); y:= chr(x);
writeln(y,y,y,y,y,y,y,y);
writeln(y,'        ',y);
writeln(y,'        ',y);
writeln(y,'        ',y);
writeln(y,y,y,y,y,y,y,y)
end.
```
{astoņas reizes tiek izvadīts simbols }

{simbols tiek izvadīts divas reizes ar atstarpi}

Šai programmā simbola veidu nosaka mainīgā x vērtība, kurai piekārtots konkrēts simbols no simbolu tabulas.

6. Noskaidrot, vai kvadrātvienādojumam $ax^2+bx+c=0$, kura koeficientus izvēlas lietotājs, ir reālas saknes.

program saknes; **var** a,b,c:**real**; **begin** **write**(‘a= ‘); **read**(a); **write**(‘b= ‘); **read**(b); **write**(‘c= ‘); **read**(c); **write**((b*b-4*a*c)>=0) **end**.	{vienādojumam ir reālas saknes, ja diskriminants $D=b^2-ac$ ir nenegatīvs} {tiek ievadīti no klaviatūras vienādojuma koeficientu vērtības} {tiek pārbaudīts, vai D nav negatīvs un izvadīts atbilstošs paziņojums}

Paškontroles uzdevumi

1. Veikt brīvi uzdotu skaitļu a un b dalīšanu (noformējot programmas veidā):

a) a/b= b) a **div** b= c) a **mod** b=

Decimāldaļskaitļus noformēt ar divām zīmēm aiz komata.

2. a) Izskaitļot izteiksmes $1,75/1,2 – 0,43^2\sin(37^o)(1-tg(2+0,3\sqrt{4,5})$

vērtību, to piešķirt identifikatoram **a** un izvadīt uz monitora ekrāna.

b) Noteikt par cik izmainās izteiksmes **y=a(c+d)/(be)** vērtība,

ja a=9, b=0,001, c= 2+x, d=2/x, e= 6-x, bet **x vērtības ir 2 vai 3**.

3. Noskaidrot, kuri no skaitļiem 31, 84, 1021, 2471, 4305. dalās bez atlikuma ar 7 vai 3 un kuri ar 3 un ar 7 vienlaicīgi. Noformēt atbildi teksta veidā.

4. Izveidot un aizpildīt tabulu

xi(grad)	x(rad)	sin(x)	cos(x)	tg(x)

pie sekojošām leņķu vērtībām: xi=0, 30, 60, 120, 240. (decimāldaļskaitļus noformēt ar divām zīmēm aiz komata)

5. Aprēķināt riņķa līnijas garumu un riņķa laukumu dotai rādiusa vērtībai. Noteikt laukumu gredzenam, kura ārējais diametrs ir **D**, bet iekšējais **d.** D un d lietotāja izvēlētas vērtības. Rezultātus izvadīt ar divām decimālzīmēm aiz komata.

6. Noteikt diennakts garumu sekundēs un atbildi noformēt teksta veidā.

7. Noskaidrot, kādi simboli atbilst kārtas numuriem **5, 15, 45, 135** un kādi kārtas numuri ir simboliem **5, D, ↑, ≠**.

8. Noskaidrot, vai uzdotiem simboliem atbilst pāru vai nepāru kārtas numuri (izmantot funkciju **odd(x)).**

```
        *          &&&&&&&&&&&&&&&&&&
       ***         &                &
      *****        &   Jānis Bērziņš &
     *******       &                &
    *********      &    II RBCB01    &
                   &                &
                   &                &
                   &&&&&&&&&&&&&&&&&&
```

9. Noteikt un pārbaudīt vienādojuma $0{,}47x^2 - 3{,}75x + 1{,}25 = 0$ saknes.

10. Sastādīt programmu, kura **monitora centrā** izvada: a) šādu piramīdu , b) tavu vizītkarti.

11. Noteikt trīszīmju vesela skaitļa ciparu summu.

Sazarojumu operatori

Līdz šim rakstītās programmas visus operatorus izpildīja tai secībā, kurā tie bija uzrakstīti. Tomēr bieži rodas nepieciešamība izmainīt programmas izpildes gaitu atkarībā no tā, vai uzdotais nosacījums izpildās vai nē. Gadījumos, kad programmas izpildes gaitā jāizvēlas viena no divām iespējamām operatoru grupām, programmā izmanto sazarojumu.

Sazarojuma konstrukcijas pieraksts teksta veidā ir:

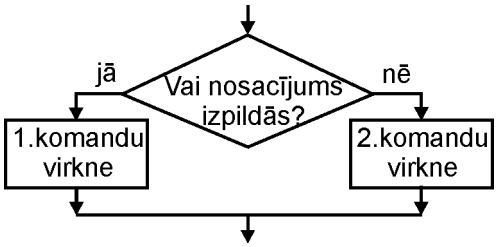
Ja izpildās <nosacījums> , **tad tiek veikta** < 1.operaturu grupa> , **pretējā gadījumā tiek veikta** < 2. operatoru grupa> .

Ar sazarojumu saprot situāciju, kad, izpildoties kādam nosacījumam, realizējas viens vai vairāki iepriekš noteikti uzdevumu risināšanas ceļi.

Pilnās sazarojuma konstrukcijas vispārīgais pieraksts **Pascal** vidē ir

If <loģiska izteiksme> **then** <operators_1> **else** <operators_2>;

Ar < **loģisko izteiksmi**> šeit domāta salīdzināšana: = (ir vienāds); <> (nav vienāds); < (ir mazāks); > (ir lielāks); <= (ir mazāks vai vienāds); >= (ir lielāks vai vienāds).

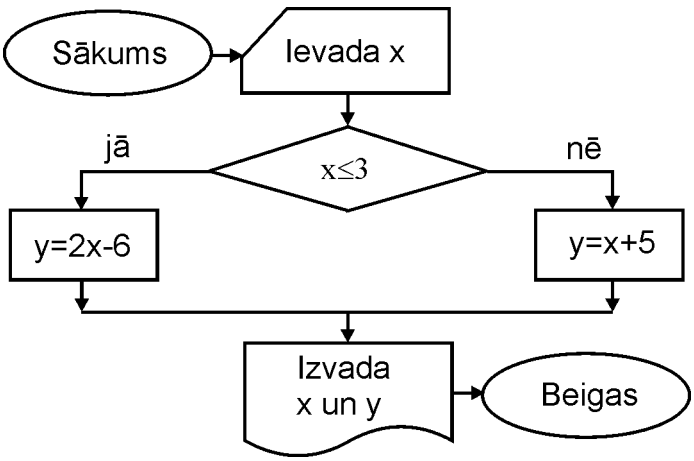


Konstrukcijas **if**..**then**..**else** blokshēma:

Uzdevuma - aprēķināt funkcijas y vērtību, ja

$$y = \begin{cases} 2x - 6, ja\ x \le 3 \\ x + 5, ja\ x > 3 \end{cases}$$

algoritma blokshēma ir:



bet algoritma izpildes programma:

program funkc2; **uses crt**; **var** x,y: **integer**; **begin** **ClrScr**; **writeln**(‘Ievadi x vertibu’); **readln**(x); **if** x<=3 **then** y:=2*x - 6 **else** y:= x +5; **writeln**(‘Ja x=’,x,’tad y= ’,y); **readln** **end**.	{attīra ekrānu} {izvada paziņojumu monitorā} {mainīgajam **y** tiek piešķirta vērtība, kura ir atkarīga no izvēlētās **x** vērtības} {iegūtā rezultāta izvadīšana } {pauze programmas izpildē}

Piemēram, ja tiek ievadīta mainīgā x vērtību 4, tad programma aprēķina y vērtību y=x+5=4+5=9 (jo 4 nav mazāks vai vienāds ar 3), bet, ja tiek ievadīta vērtību x = -4, tad programmas aprēķina rezultātā iegūstam vērtību y=-14.

Nepieciešams atcerēties, ka, ja aiz operatora **then** vai aiz operatora **else** seko vairāki operatori, tad tie jāraksta starp operatoru iekavām **begin** un **end**, t.i.,

if <nosacījums> **then**

begin <operators a1>; <operators a2>; ..... <operators an>; **end**

else

begin <operators b1>; <operators b2>; …. <operators bn>; **end**;

pie kam aiz pirmā operatoru bloka **end** semikols (;) **<u>nav jāliek</u>**.

Piemērs:

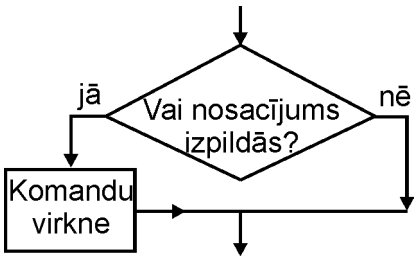
```
if X=0 then
    writeln(‘A=’, X)
        else
            begin
                A:=X - 1;
                writeln(‘A=’, A);
             end;
```

```
if Y=0 then
    begin
        A:=Y+1;
        writeln(‘A=’, A);
    end
else
writeln(‘A=’, A);
```

Daļējā sazarojuma konstrukcijas forma

if <nosacījumi> **then** < operatori>

ir izmantojama gadījumos, ja kāda programmas daļa jāizpilda tikai pie noteiktiem nosacījumiem. Gadījumos, kad šie nosacījumi neizpildās, operatori aiz atslēgvārda **then** tiek ignorēti un izpildās nākamie programmas operatori.



Šādas konstrukcijas **if**..**then** blokshēma ir:

Piemērs. Sastādīt programmu, kura dod iespēju aprēķināt divu patvaļīgu no klaviatūras ievadītu skaitļu A un B summu un izvadīt to uz ekrāna gadījumos, ja kāds no skaitļiem A vai B ir pāra skaitlis. Gadījumos, kad neviens no dotajiem skaitļiem nav pāru skaitlis, programma beidz darbu neaprēķinot summu.

```
program summa;
uses crt;
Var a, b, sum: integer;
begin
ClrScr;
write('Ievadi a :'); readln (a);
write('Ievadi b :'); readln (b);
if (a mod 2=0) or (b mod 2=0) then      {sazarojums ar dubultnosacījumu or (vai)}
   begin
      sum:=a+b;
      writeln('summa ir: ',sum);        {izvada summu}
   end;                                 {sazarojuma beigas}
writeln('Darbs pabeigts');              {izvada paziņojumu}
end.
```

Izvēles operators CASE ... OF ... END

Izmantojot nosacījuma operatoru **if...**, iespējams izvēlēties vienu no diviem iespējamiem variantiem. Nav ieteicams nosacījuma operatorus atkārtoti ieslēgt vienu otrā. Tas sarežģī programmas struktūru un dara to nepārskatāmu. Tiek rekomendēts veidot ne vairāk kā 2 – 3 ieslēgtos operatorus. Gadījumos, kad jāpārbauda vairāki nosacījumi, tiek lietots **izvēles**, jeb **variantu** operators **case**.

Operatora vispārīgā forma ir:

```
case <selektors> of
N1: begin   <operatoru grupa 1>  end;
N2: begin   <operatoru grupa 2>  end;
N3: begin   <operatoru grupa 3>  end;
..... ;
```

```
Nk: begin   <operatoru grupa k>  end
else
begin <operatoru grupa (k+1)>    end;
end;
```

Ar šo operatoru var izvēlēties vienu no vairākiem operatoriem vai operatoru grupām. Operators **case** sastāv no **selektora** (mainīgā) un operatoru saraksta, kur katram operatoram vai operatoru grupai tiek piešķirta sava konstante. Ja mainīgā vērtība sakrīt ar kādu no konstantēm, tiek izpildīts attiecīgais operators vai operatoru grupa. Ja attiecīgās konstantes nav, tad tiek uzrādīta kļūda un programmas darbība tiek pārtraukta vai arī tiek izpildīts tas operators, kurš seko tieši aiz izvēles operatora.

No visām operatoru grupām tiks izpildīta tikai viena - tā, kuras priekšā konstantes $\mathbf{N_i}$ vērtība sakrīt ar **selektora** vērtību. Operatorā **case** izmantojamās konstantes $\mathbf{N_i}$ netiek uzdotas apraksta daļā. Izpildāmo **operatoru grupas** var sastāvēt no viena vai vairākiem operatoriem. Selektora vērtībām var izmantot **veselu**, **loģisku** vai **simbolu** tipu. Tas nevar būt **real**.

Operatoru grupa pēc **else** tiek izpildīta gadījumā, ja neviena konstantes vērtība nesakrīt ar **selektora** vērtību. Gadījumos, ja vārds **else** nav iekļauts operatorā **case**, tiek izpildīts nākamais operators, kurš seko pēc **case**. Jāņem vērā, ka operatora **case** beigās stāv vārds **end**, kuram nav attiecīgā pāra vārda **begin**.

Piemērs. Gadījumā, ja nepieciešams dotu veselu skaitli N pārveidot atbilstoši pazīmei, kuru nosaka šī skaitļa dalījuma ar 17 atlikums sekojošā veidā:

case (N **mod** 17) **of**	
0: N:=0;	ja N **mod** 17=0, tad N=0;
1,6: N:=-N;	N **mod** 17=1 vai 6, tad N= - N;
2,3,5: N:=2*N;	N **mod** 17=2, 3 vai 5, tad N= 2N;
4: N:=3*N;	N **mod** 17=4, tad N= 3N,
else	bet visos citos gadījumos N=5N.
N:=5*N;	
end;	

Pascal vidē šo uzdevumu veic dotais operators.

Piemērs. Sastādīt programmu, kura pieprasa lietotājam piespiest kādu klaviatūras taustiņu un izdrukā piespiestā taustiņa grupas nosaukumu: Burts, Skaitlis, Operators, Speciāls simbols.

program simbols;	
uses crt;	
var x: **char**;	
Begin ClrScr;	
writeln(‘Nospied kādu	

klaviatūras taustiņu); **readln**(x); **case** x **of** 'A'..'Z','a'..'z': w**riteln**('Burts'); '0'..'9': **writeln**('Skaitlis'); '+','-','*','/': **writeln**('Operators'); **else** **writeln**('Specials simbols'); **end**; **readln** **end**.	{izvada paziņojumu monitorā} {mainīgo ievadīšana no klaviatūras} {izvēles operatora **case** sākums} {izvēles: nospiežot kādu taustiņu - simbolu, tiek izvadīts atbilstošais paziņojums, nospiežot taustiņu, kas neatbilst nevienam no uzrādītajiem simboliem, tiek izvadīts paziņojums, kas seko aiz **else**}

Paškontroles uzdevumi

1. Kāda būs mainīgā A vērtība pēc šādu operatoru izpildes:

A:=0;

if B > 0 **then if** C > 0 **then** A:=1 **else** A:=2;

pie dotajām mainīgo B un C vērtībām:

a) B:=1; C:=1; b) B:=1; C:=-1; c) B:= -1; C:=1;

2. Norādīt kļūdu un paskaidrot to:

a)	**b)**

if 3<A<7 **then**
A:=A+1; B:=-1;
else A:=0; B:=B+1;
	if 3<A **and** A<7 **then**
	begin A:=A+1; B:=-1; **end**;
	else
	begin A:=0; B:=B+1 **end**;

3. Doto algoritmu:

ja mainīgā A vērtība nav 0 un A kvadrātā ir mazāks par 25, tad mainīt A zīmi;

ja A ir 0, tad mainīgajam A piešķirt vērtību 1.

pārrakstīt programmēšanas valodā Pascal

4. Sastādīt programmu, kas pieprasa ievadīt skaitli no 1 līdz 10, un kura ievadīto skaitli izvada monitorā vārdiskā formā.

5. Sastādīt programmu, kura nosaka, cik un kādi atrisinājumu būs kvadrātvienādojumam $ax^2+bx+c=0$ (mainīgos a, b un c ievada lietotājs).

6. Sastādīt programmu, kas aprēķina funkcijas

$$y=\begin{cases} 2x+5,\ ja\, x \le 3 \\ 2x-\dfrac{3}{x-1}, jax > 3 \end{cases}$$

vērtību, ja mainīgo x nosaka sakarība x= 3sin(απ /180), bet α ievada lietotājs.

Programmā paredzēt iespēju izvadīt paziņojumu ‘’funkciju nav definēta’’.

7. Sastādīt programmu, kas pieprasa ievadīt 3 nogriežņu garumus un nosaka, vai no tiem var uzkonstruēt trīsstūri (viena trīsstūra mala ir mazāka par divu pārējo trīsstūra malu summu).

Cikla operatori

Cikls ir viens no programmēšanas pamatelementiem. Cikla rezultātā tiek daudzkārt atkārtota vienas operatoru grupas izpilde. Tas sevī ietver programmēšanas būtību – daudzkārtēju vienveidīgu operāciju izpildi. Pascal-ā tiek izmantoti cikli kā ar fiksētu to izpildes skaitu, tā arī ar iepriekš nezināmu ciklu skaitu. Pirmos sauc par kontrolējamiem cikliem, bet otros par cikliem ar nosacījumiem. Nemākulīga ciklu lietošana var radīt situāciju, kad programma nevar iziet no cikla (programma ieciklojusies). Tādā gadījumā Turbo Paskālā tiek izmantoti klaviatūras taustiņi **Ctrl+Break**. Ja tas nelīdz, Pascal programmu var aizvērt izmantojot taustiņu kombināciju **Ctrl+Alt+Delete** vai veikt datora pārstartēšanu ar taustiņu **Reset**. Tas uzskatāms par galēju līdzekli, jo tā rezultātā izpildāmās programmas dati tiks pazaudēti.

Operators FOR ... TO ... DO...

Fiksēta ciklu skaita operatoram **for** ir sekojoša konstrukcija:

for i:= a **to** b **do begin** <operatoru grupa> **end**;

(no **i** vērtības vienādas ar **a** līdz **i** vērtībai vienādai ar **b** tiek izpildīta **operatoru grupa**).

Tātad operatoru grupa tiek izpildīta (**b–a+1**) reizes. Svarīgi ņemt vērā, ka ciklu izpildes solis ir **+1**, bet parametriem **a** un **b** jāapmierina nosacījums **a<b**. Operatora parametri **a, b, i** tiek definēti kā naturāli skaitļi, vai simbolu tips, piem., a,b,i:**integer**; vai a,b,i:**char**;

Cikla operatoram soli var izvēlēties arī vienādu ar **- 1**. Tādā gadījumā operatora konstrukcija ir:

for i:=b **downto** a **do begin** <operatoru grupa> **end**;

Cikla operatora izpilde sākas ar nosacījuma **i<= b** pārbaudi. Ja nosacījums neizpildās, tad cikla operatoru grupa netiek izpildīta un tiek turpināta programma. Ja nosacījums izpildās, tad tiek izpildīta cikla operatoru grupa un cikla parametram i tiek piešķirta nākamā vērtība **i:=i -1**. Process atkārtojas.

Piemēri:

Izpildot programmas fragmentu – operatoru

for i:=15 **to** 19 **do write**(i:3); uz ekrāna tiks izvadīta ciparu virkne: **15 16 17 18 19,**

for i:=19 **downto** 15 **do write**(i:3); uz ekrāna tiks izvadīta ciparu virkne: **19 18 17 16 15**.

for ch:=‘a’ **to** ‘e’ **do write**(ch:2); uz ekrāna tiek izvadīta burtu virkne: **a b c d e.**

for i:= ‘e’ **downto** ‘a’ **do write**(ch:2); uz ekrāna tiek izvadīta burtu virkne: **e d c b a**.

Programmas datu izvadei ērti lietot cikla operatoru **for**.

Tā, piemēram, operators **for** i:=1 **to** 50 **do**

begin writeln(i);

if i **mod** 24 = 23 **then readln**; **end**;

apstādinās datu izvadi pie **i = 23 un 47**. Izvade atjaunosies nospiežot Enter.

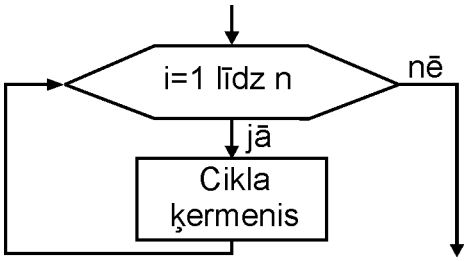
Priekšlaicīgu izeju no cikla var panākt izmantojot operatoru **goto** vai **break**.

Piemēram: **for** i:=1 **to** 45 **do**

begin f:=f+i;

if (f>100) **or** (i=39) **then break**; **end**;

Konstrukcijas **for...to...do** blokshēma ir:



Ja ciklā jāizpilda vairākas komandas, tad tās jāraksta starp atslēgvārdiem **begin** un **end**.

Uzdevums. Sastādīt programmu, kura no lietotāja ievadītā skaitļa vispirms atņem 1, tad 2, tad 3, ..., tad 10 un galīgo rezultātu izvada uz displeja ekrāna.

program atnem_1;	
uses Crt;	
var n,i:**integer**;	
begin	
ClrScr;	
write('Ievadi skaitli: '); **readln**(n);	{ievada n vērtību}
for i:=1 **to** 10 **do**	{cikls}
n:=n-i;	{n tiek samazināts par i}
writeln(‘Skaitlis ir: ',n);	{izvada iegūto n vērtību}
end.	

Modificējam programmu tā, lai no lietotāja ievadītā skaitļa n vispirms tiktu atņemts 10, tad 9, tad 8, ...,1. Visi rezultāti jāizvada uz monitora ekrāna.

program atnem_10;	
uses crt;	
var n,i:**integer**;	
begin ClrScr;	
writeln('Ievadi skaitli: ');	

```
readln(n);
for i:=10 downto 1 do
    begin
        n:=n-i;
        writeln(‘Skaitlis ir: ',n);
    end;
readln;
end.
```

{cikls}

{mainīgā **n** vērtība tiek samazināta par **i** un iegūtais rezultāts izvadīts uz monitora ekrāna}

Programmas izpildes rezultātā, gadījumā, ja n = 100, iegūstam naturālu skaitļu virkni **90 81 73 66 60 55 51 48 46 45.**

Piemēri.

1. Izveidot programmu, kura ļauj noteikt visu naturālo skaitļu summu intervālā (a,b). Izveidot programmas **atnem_10** variantu izmantojot iegūtās programmas operatoru bloku.

2. Izveidot programmu, kura ļauj noteikt uzdota naturāla skaitļa n faktoriāla vērtību (n!=1·2·3·4·...·(n-1)·n).

3. Izveidot programmu, kura ļauj noteikt no 100 ģenerētiem skaitļiem maksimālo skaitli.

4. Izveidot programmu, kura fiksētām naturāla skaitļa **n** vērtībām nosaka skaitļu summas S=1+1/2+1/3+1/4+....+1/n vērtību.

5. Izveidot programmu, kura ļauj noteikt 100 ģenerēto skaitļu pāra un nepāra skaitļu skaitu, summu un vidējo vērtību.

6. Izveidot programmu reizrēķina zināšanu testēšanai.

```
program Nr1;
var i,a,b,s:integer;
begin
writeln(‘ievada skaitli a= ‘);
read(a);
writeln(‘ievada skaitli b= ‘);
read(b);
s:=0;
for i:=a to b do
s:=s+i;
write(s)
end.
```

```
program Nr2;
var i,n,fakt:integer;
begin
writeln(‘ievada skaitli n= ‘);
readln(n);
fakt:=1;
for i:=1 to n do
fakt:=fakt*i;
write(fakt)
end.
```

```
program Nr3;
var
```

```
program Nr4;
var
```

```
i,m,a:integer;
begin
  m:=0;
  for i:=1 to 100 do
  begin
    a:=random(50);
    if a>m then m:=a;
  end;
  write(m)
end.
```

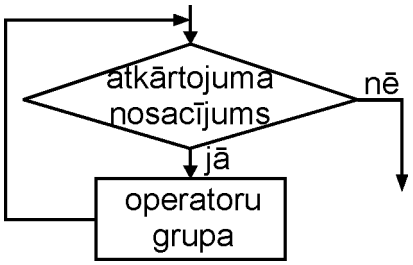
```
i,n:integer;
s:real;
begin
writeln('ievada skaitli n= ');
readln(n);
s:=0;
for i:=1 to n do
s:=s+1/i;
write('summa ir s=',s:6:3)
end.
```

```
program Nr5;
var
i,a,nsk,nsum,psk,psum:integer;
pvv,nvv: real;
begin
psk:=0; nsk:=0; psum:=0; nsum:=0;
for i:=1 to 100 do
begin  a:=random(50);
if a mod 2=0 then
begin psk:=psk+1; psum:=psum+a;
end
else
begin nsk:=nsk+1; nsum:=nsum+a;
end;
pvv:=psum/psk; nvv:=nsum/nsk;
write(psk:4,psum:6,nsk:4,nsum:6,
pvv:4:2,nvv:4:2)
end.
```

```
program Nr6;
var
i,a,b,atbilde,atziime:integer;
begin
randomize;
for i:=1 to 5 do
begin  a:=random(18)+2;
b:=random(18)+2;
write('cik ir',a,'*',b,'?');
readln(atbilde);
if atbilde=a*b then
begin  write('pareizi');
atziime:=atziime+1; end
else write('nepareizi'); end;
writeln('Juusu atziime -',atzi-
ime)
end.
```

Operators WHILE ... DO ...

Cikla operators **while** ... **do** ... organizē ciklus ar neierobežotu ciklu skaitu. Tas nozīmē, ka tiek atkārtota fiksēta operatoru grupa tik reizes, cik reizes ir spēkā uzdotais nosacījums. Cikla izpildes nosacījuma pārbaude notiek operatora sākumā (cikla priekšnosacījums). Operatora **while** ... **do** ...blokshēma redzama attēlā.



Jāņem vērā, ka cikla parametram (mainīgajam, kuru satur kā cikla nosacījums, tā <operatoru grupa>) pirms cikla izpildes uzsākšanas jāpiešķir noteikta vērtība, un vienam no cikla operatoriem šī vērtība ir jāmaina, jo pretējā gadījumā cikls turpināsies bezgalīgi.

Operatora pieraksta forma :

while <nosacījums> **do begin** <operatoru grupa>; **end**;

Nosacījums ir loģiskā izteiksme. Tas var tikt izteikts dažādos veidos. Piemēram: x>0; (a>1) and (b<>0); s='A'. Gadījumos, kad **nosacījuma** vērtība ir **true**, izpildās cikla **operatoru grupa**, pie **nosacījuma** vērtības **false** šī izpilde pārtraucas. Ja pirmā vērtība ir **false**, tad cikls neizpildās un otrādi, ja **nosacījuma** vērtība ir **true** un pēc **operatoru grupas** izpildes tā nemainās, cikls atkārtojas bezgalīgi. Šo cikla operatoru lietderīgi lietot gadījumos, kad iespējamas situācijas, kurās nosacījums neizpildās un līdz ar to cikla operatoru izpilde izpaliek.

Piemērs. Izmantojot cikla operatoru **while**...**do** sastādīt programmu, kura dod iespēju nodrukāt visus skaitļus, kuri nepārsniedz 100 un dalās ar 7.

```
program dalamie_2;
uses crt;
var y:integer;
begin ClrScr;
y:=0;                          {piešķir y sākumvērtību}
while y=<100 do                {cikla sākums, cikls tiek pildīts tik ilgi, kamēr y ir mazāks vai vienāds ar 100}
begin
    y:=y+7;                    {y vērtība tiek palielināta par 7}
    writeln(y);                { izvada parametra y vērtību }
end;                           {cikla beigas}
writeln('Programma             {paziņojums teksta veidā}
darbu pabeidza.');
readln;
end.
```

Piemēri.

1. Nodrukāt n skaitļus (no 1 līdz n), izmantojot operatoru **while**.

2. Sastādīt programmu, kura nosaka to naturālo skaitli **k**, pie kuras izteiksmes $\mathbf{x^k/k}$ vērtība kļūst lielāka par uzdotu skaitli **A** (x>1, A>1).

```
program Nr1;
const n=10;
var m:integer;
begin
m:=0;
while m<n do
begin
    m:=m+1;
    write(m);
end
end.
```

```
program Nr2;
var x,A,p:real;
k:integer;
begin
read(x,A); k:=1; p:=x;
while p/k<=A do
begin
    k:=k+1;   p:=p*x;
end;
write(k)
end.
```

Piemērs. Noteikt funkcijas $y(x) = \lg(3) + x\sqrt{5\sin(\pi x/3)}$ vērtības intervālā (a, b) ar soli dx.

```
program funkcija;
var x,y,z,dx,a,b,lg3:real;
begin
write('ievadam a ='); readln(a);
write('ievadam b ='); readln(b);
write('ievadam dx ='); readln(dx);
lg3:=ln(3)/ln(10);
x:=a;
while x<=b do
    begin
        z:=sin(pi*x/3);
        if (z<0) then writeln('funkcija nav definēta')
    else
    begin
        y:= lg3 + x* sqrt(5*z);
        writeln('pie x=',x,'y=',y);
    end;
x:= x+dx;
    end
```

```
end.
```

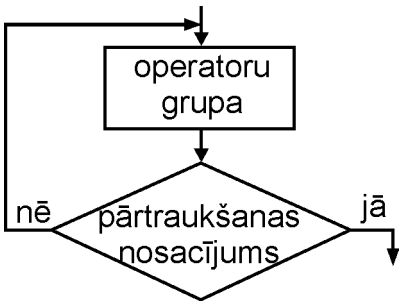
Operators REPEAT ... UNTIL...

Cikla operators **repeat** arī organizē ciklus ar neierobežotu ciklu skaitu. Atšķirībā no iepriekšējā operatora (**while**), ciklu atkārtotas izpildes nosacījums pārbaudās pēc cikla operatoru grupas izpildes. Tātad cikla operatoru grupa izpildās vismaz vienu reizi, pat gadījumā, ja nosacījums pie pārbaudes neizpildīsies. Operatoru grupa ir ieslēgta starp vārdiem **repeat** un **until** un līdz ar to operatoru iekavas begin un end nav nepieciešamas.

Operatora vispārējā pieraksta forma ir

```
Repeat <operatoru grupa> until <nosacījums>;
```

Cikls atkārto <operatoru grupas> izpildi līdz brīdim, kad izpildās nosacījums cikla beigās. Nosacījums aiz atslēgvārda **until** norāda, kad programmai pārtraukt cikla izpildi. Tātad cikls atkārtojas tik ilgi, kamēr nosacījuma vērtība ir **false**.



Šī operatora blokshēma ir

Jāievēro, ka cikla mainīgajam pirms cikla jāpiešķir kāda konkrēta vērtība, un vienam no cikla operatoriem šī vērtība ir jāizmaina, pretējā gadījumā cikls turpināsies bezgalīgi.

Piemērs. Sastādīsim programmu, kura izvada uz ekrāna visus veselos skaitļus, kuri nepārsniedz 100 un dalās ar 9.

```
program dal_ar_9;
uses crt;
var x:integer;
begin  ClrScr;
x:=0;
repeat                          {cikla sākums};
    x:=x+9;                     { x vērtība tiek palielināta par 9}
    write(x:5);                 {tiek izvadīta mainīgā x vērtība}
until x>=100;                   {cikla beigas un nosacījuma pārbaude}
writeln('Programma darbu pabeidza.');   {paziņojuma izvadīšana}
readln;
end.
```

Monitora ekrānā tiek izvadīti skaitļi:

9 18 27 36 45 54 63 72 81 90 99

Piemēri. Izmantojot operatoru **repeat: 1)** nodrukāt **n** naturālus skaitļus, **2)** . noteikt to naturālo skaitli **k**, pie kura izteiksmes $\mathbf{x^k/k}$ vērtība kļūst lielāka par uzdotu skaitli **A** (x>1, A>1).

```
program Nr_1r;
const n=10;
var m:integer;
begin
m:=0;
repeat
    m:=m+1;
    write(m)
until m>n
end.
```

```
program Nr_2r;
var x, a, p:real;
k:integer;
begin
read(x,A); k:=0; p:=1;
repeat
    k:=k+1;
    p:=p*x
until p/k<=A;
write(k)
end.
```

Paškontroles uzdevumi

1. Kas tiks izdrukāts monitora ekrānā pēc doto programmas fragmentu izpildes?

a) **for** i:=3 **to** 8 **do write**(2*i,‘ ‘);

b) **for** i:=1 **to** 5 **do write**(i:4, ' :', (100-i):5);

c) Skaitlis:= 2;

for i:= (5*2-4) **to** 5*Skaitlis **do write** ('**', i:5);

d) **for** i := 12 **downto** 4 **do writeln**(25-i: 5);

2. Izlabo dotās programmas tā, lai tās atbilstu uzdevumu nosacījumiem.

a) Sastādīt programmu, kas monitora ekrānā izdrukā skaitļus no 1 līdz 5 vienu zem otra.

for i := 1 **to** 5 **do**; **writeln**(i);

b) Sastādīt programmu, kas ļauj ievadīt 10 skaitļus, saskaita tos un rezultātu izdrukā monitora ekrānā.

Summa := 0;

for i:= 1 **to** 10 **do**

read(Skaitlis);

Summa:= Summa + Skaitlis;

writeln(Summa: 15);

3. Pārraksti doto programmas fragmentu, izmantojot operatoru **for** ..**to**..**do** tā, lai monitora ekrānā tiktu izdrukāts tāds pats rezultāts:

for i:= 10 **downto** 3 **do writeln**(i: i);

4. Pārraksti doto programmas fragmentu, izmantojot operatoru **for**..**downto**..**do** tā, lai monitora ekrānā tiktu izdrukāts tāds pats rezultāts.

Summa := 0;

for i := 1 **to** 4 **do begin**

writeln(‘*’: 10 + i);

Summa:= Summa + i;

writeln(Summa); **end**;

5. Izveidot programmu, kura ļauj noteikt visu intervāla (a,b) naturālo skaitļu kvadrātu summu.

6. Izveidot programmu, kura ļauj noteikt uzdota intervāla (a,b) naturālo skaitļu reizinājumu.

7. Izveidot programmu, kura ļauj noteikt no 100 ģenerētiem skaitļiem maksimālo vai minimālo skaitļu skaitu.

8. Izveidot programmu, kura ļauj noteikt 100 ģenerēto skaitļu skaita sadalījumu pa intervāliem (0, k/3), (k/3, 2k/3) un (2k/3, k).

9. Izveidot programmu, kura ļauj noteikt skaitļu summas S = 1 - 1/2 +1/3 – 1/4 + ...+ $(-1)^n$ /n vērtību fiksētām naturāla skaitļa n vērtībām.

10. Izveidot programmu, kura ļauj izveidot atbilstību starp temperatūru vērtībām Celsija un Fārenheita skalās. Izmantot sakarību $\mathbf{t_F=9t_C/5+32.}$

11. Sastādiet programmu, kas izvada uz ekrāna $Y=X^2+3X-2$ vērtību tabulu X vērtībām no 1 līdz 10.

12. Sastādīt programmu, kas aprēķina **n** pēc kārtas ņemtu skaitļu summu (intervāla sākumu un n ievada lietotājs). Piemēram, ja intervāla sākums ir 3 un n=4, tad programma saskaita 3+4+5+6=18.

13. Sastādīt programmu, kas izvada 7 gadījuma skaitļus, kuri atrodas intervālā [2,6] un aprēķina šo skaitļu kvadrātu summu.

14. Sastādīt programmu, kura aprēķina Tavu naudas daudzumu, tas ir, Tu esi ielicis bankā N latus uz P procentiem gadā un uz G gadiem, bet programma aprēķina, cik tev ir bankā nauda pēc G gadiem.

15. Sastādīt programmu, kura ļauj noteikt bezgalīgas skaitļu rindas locekļu summu ar precizitāti līdz saskaitāmajam, kurš mazāks par uzdoto skaitli p.

S=1+1/2+1/3+1/4+.....+1/n (p=0,01);

S=1-1/2+1/3-1/4+.....+1/n (p=0,01);

S=1-1/2-1/3-1/4+.....-1/n (p=0,01).

16. Sastādīt programmu, kura ļauj noteikt, cik dienām pietiks 200 tonnu cementa, ja pirmajā dienā patērē 5 tonnas, bet katrā nākamajā dienā par 20% vairāk nekā iepriekšējā.

17. Sastādīt programmu, kura nosaka visus Pitagora skaitļus gadījumam, kad 1<=a<20, 1<=b<20. Tātad jāatrod vesels skaitlis, kura kvadrāts ir divu skaitļu no dotā intervāla kvadrātu summa.

MASĪVI

Darbojoties ar garām skaitļu vai simbolu virknēm, līdz šim bija nepieciešams definēt katram skaitlim vai simbolam savu atsevišķu mainīgo. Tas rada pārblīvētības iespaidu un apgrūtina programmas pārskatāmību.

Lai atvieglotu liela apjoma datu apstrādi, **Pascal** un daudzās citās programmēšanas valodās ir iespēja veidot strukturētus mainīgos, kuru struktūra sastāv no vairākām viena tipa vērtībām. Katram strukturētajam mainīgajam ir viens nosaukums, bet pats mainīgais ir sadalīts mazākās vienībās un katra vienība satur vienu vērtību. Pie strukturētajiem mainīgajiem pieder **masīvi**, **ieraksti** un **faili**.

Vairākas viena tipa vērtības var apvienot masīvā. **Masīvs ir galīga skaita elementu kopums, kurā apvienoti viena tipa elementi.** Katram masīva elementam ir savs kārtas numurs. Masīvus iedala viendimensiju un daudzdimensiju masīvos. Tā, piemēram, skaitļu virkni var uzskatīt par viendimensiju masīvu. Bet datu tabulu (piemēram matricu) par divdimensiju masīvu. Pieeja masīva elementiem notiek pēc elementa kārtas numura masīvā jeb indeksa. Ar indeksa palīdzību tiek norādīts konkrētais masīva elements. Katram masīva elementam ir viens vai vairāki indeksi, kuri norāda elementa vietu masīvā. Elementi masīvā ir sakārtoti to indeksu augšanas secībā. Masīvu raksturojoši lielumi ir

- tips – visu masīva elementu tips;
- izmērs (rangs) – masīva indeksu skaits;
- indeksu izmaiņas diapazons – nosaka visu masīva elementu skaitu.

Vektors ir masīva piemērs, kura visi elementi tiek numurēti ar vienu indeksu. Līdz ar to vektora koordinātes ir viendimensiju masīva elementi.

Matrica (vērtību tabula) ir divdimensiju masīva piemērs. Tā elementi tiek numurēti ar diviem indeksiem – rindas numuru un stabiņa numuru. Augstāku kārtu masīvi praksē sastopami visai reti.

Datora atmiņā visi masīva elementi aizņem obligāti vienu nepārtrauktu apgabalu. Divdimensiju masīvi atmiņā tiek izvietoti pa rindām: vispirms visi pirmās rindas elementi, tad otras u.t.t. Masīva elementa numurs vispārīgā gadījumā ir kārtas tipa izteiksme. Visbiežāk indekss ir konstante vai **integer** tipa mainīgais, retāk – **char** tipa vai **boolean** tipa lielums.

Konkrēts masīva elements tiek aprakstīts aiz masīva nosaukuma kvadrātiekavās norādot tā indeksu. Piemēram, A[3], B[3,5].

Viena no masīvu priekšrocībām ir tā, ka to indeksi var būt gan mainīgie, gan izteiksmes. Tas dod iespēju vērsties pie noteiktas masīva elementu virknes. Tā, piemēram A[i] dod iespēju pārskatīt visus masīva elementus, bet pieraksts A[i*2] – elementus, kuri atrodas pāra vietās vai A[2*i-1] – nepāra vietās.

Vienkāršākais masīva apraksta veids - mainīgo apraksta daļā **var** tiek definēts attiecīgs apzīmējums izmantojot vārdu **array**.

Viendimensiju masīva gadījumā

var <masīva vārds>**array**[apakšējā robeža..augšējā robeža]

Piemēram:

a:**array**[1..100] of **integer**;	{masīvs sastāv no 100 elementiem veselu skaitļu veidā}
b:**array**[0..25] of **char**;	{26 elementi – simboli}
c:**array**[-3..4] of **boolean**;	{8 elementi – loģiskās vērtības}

Divdimensiju gadījumā

var <masīva vārds>**array**[r1..rn, s1..sm] **of** <elementu tips>

kur, r1 – pirmās rindas kārtas numurs;

rn - pēdējās rindas kārtas numurs;

s1 - pirmā stabiņa kārtas numurs;

sm – pēdējā stabiņa kārtas numurs.

Tātad definējot tabulu

1	3	5	7
2	4	6	8
9	8	7	6

masīva veidā aprakstam jālieto

var tabula:**array**[1..3,1..4] of **integer**;

Svarīgi ievērot, ka ne vienmēr visi masīva elementi ir aizpildīti, tas nozīmē, ka reālais masīva elementu skaits var būt mazāks par definēto, bet **nekādā gadījumā nedrīkst rasties situācija**, kad reālais masīva elementu skaits ir lielāks par masīva definīcijā paredzēto.

Svarīgi ņemt vērā, ka Turbo Pascal-ā ir visai stingrs ierobežojums attiecībā uz operatīvo atmiņu, kas pieejama mainīgo aprakstam. Tās apjoms ir 64 kbaiti. Tātad izmantojot definīcijā **array**[1..105,1..105] of **real** tiek pārsniegta programmēšanas vidē pieejamā atmiņa (**viens real** tipa elements atmiņā aizņem **6 baitus**). Šādā situācijā tiek uzrādīta kļūda **Error 22 (structure too large)**. Operatīvās atmiņas ekonomijai lietderīgi iespēju robežās izmantot vienbaitīgus tipus **byte** un **shortint**.

Definējot masīvus to indeksu robežas **nedrīkst** uzdot ar mainīgajiem. Šim nolūkam ieteicams izmantot konstantes, kuras nosaka elementu skaitu. Tātad divdimensiju masīvu ērti definēt sekojošā veidā:

const r=10; k=15;

var matrica:**array**[1..r,1..k] **of real**;

Masīvu definēšanu var realizēt arī citos veidos – **const** daļā kā tipizētu konstanti vai **type** daļā definējot masīvu kā atsevišķu tipu.

Iespējamās **tipiskās kļūdas** aprakstot masīvus:

- nav noteikts masīvu izmērs un diapazona robežas - a:**array**[ ] **of real**;
- masīva apakšējā robeža lielāka par augšējo - a:**array**[10..1] **of integer**;
- masīva robežas jāuzrāda ar konstantēm, bet ne ar izteiksmēm - a:**array**[1..x+y] **of real**;

- indeksu robežas nedrīkst uzdot ar decimālskaitļiem - a:**array**[1.0..20.0] **of integer**.

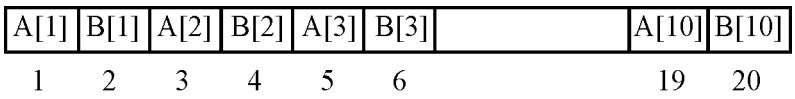
Izejot ārpus masīva indeksu robežām var tikt bojātas ar masīva elementiem aizņemtās atmiņas daļas blakus šūnas.

Ja vairākiem masīviem ir vienāds elementu skaits un viens un tas pats tips, tad tos var definēt, piemēram, šādi:

var A, B, C**:array**[1..10] **of real;**

Te katram no masīviem A, B, C var būt līdz 10 elementiem, kuri ir reāli skaitļi. Vienāda tipa un izmēra masīviem vērtību piešķiršanu var veikt šādi: A:=B; {masīvam A piešķir masīva B vērtības}.

Uzdevums. Doti trīs masīvi: A[1..10], B[1..10], C[1..20]. Masīvs A aizpildīts ar gadījuma skaitļiem. Masīvam B tiek piešķirtas masīva A vērtības. Jāsastāda programma, kura dod iespēju aizpildīt masīvu C atbilstoši sekojošai shēmai:



program masivi_3;	
uses crt;	
var A,B:**array**[1..10] **of integer**;	{definē divus 10 **integer** tipa elementu masīvus}
C:**array**[1..20] of **integer**;	{definē 20 **integer** tipa elementu masīvu}
i:**integer**;	
begin clrscr;	
randomize;	{masīva A aizpildīšana ar gadījuma skaitļiem
for i:=1 **to** 10 **do**	
A[i]:=**random**(10);	
writeln('Masivs A:');	{masīva A vērtību izvade ekrānā, izmantojot ciklu}
for i:=1 **to** 10 **do**	
write(A[i]:3);	{izvada tukšu rindu}
writeln;	{masīvam B piešķir masīva A vērtības}
B:=A;	
writeln('Masivs B:');	{masīva B vērtību izvade ekrānā, izmantojot ciklu}
for i:=1 **to** 10 **do**	
write(B[i]:3);	{masīva C elementiem, kuru indeksi ir nepāra skaitļi piešķir masīva A vērtības}
writeln;	
for i:=1 **to** 10 **do**	
begin	{masīva C elementiem, kuru indeksi ir pāra skaitļi piešķir

C[2*i-1]:=A[i]; C[2*i]:=B[i]; **end**; **writeln**('Masivs C:'); **for** i:=1 **to** 20 **do** **write**(C[i]:3); **readln**; **end**.	masīva B vērtības} {Masīva C vērtību izvade ekrānā, izmantojot ciklu}

Darbības ar masīviem

Viendimensiju masīvu var stādīties priekšā kā galīgu lentu, kura sadalīta rūtiņās. Katrai rūtiņai ir kārtas numurs un attiecīgajā rūtiņā var kaut ko ierakstīt. Ar masīvu nevar rīkoties kā vienu veselu. Lai veiktu kādu operāciju ar masīvu kopumā, šī operācija ir jāveic ar katru masīva elementu atsevišķi, vēršoties pie konkrētā masīva elementa, izmantojot tā indeksu.

Ja programmā nepieciešams izvadīt ekrānā visu masīva elementu vērtības, tad visērtāk ir lietot galīga skaita ciklu operatoru **for**, kurā cikla mainīgo izmanto kā masīva indeksa vērtību.

Masīvu aizpildīšana.

Masīva elementu vērtības var uzdot sekojošos veidos:

- datu ievads no klaviatūras (programmas piemērs 1);
- ar gadījuma skaitļu ģeneratoru (programmas piemērs 2);
- ar piešķires operatoru;
- nolasot elementu vērtības no faila.

Katrā no šiem gadījumiem masīva aizpildīšanai tiek izmantots cikls.

Tā, piemēram, ievads no klaviatūras realizējas ar

for i:=1 **to** 5 **do readln**(a[i]); {vektors no 5 elementiem}

vai

for i:=1 **to** 3 **do**

for j:=1 **to** 2 **do readln**([b[i,j]); {matrica ar izmēru 3x2, t.i. 6 elementiem aizpildās pa kolonām}.

Aizpildot masīvu c ar n gadījuma skaitļiem no diapazona 0–99 izmantojam

randomize; {gadījuma skaitļu devēja inicializācija}

for i:=1 **to** n **do** c[i]:=**random**(100);

Masīvu aizpildīšana ar gadījuma skaitļiem bieži tiek lietota veicot matemātisko modelēšanu un spēles gadījuma situāciju realizācijai.

Gadījumos, kad masīvs „jāattīra” no iepriekšējām elementu vērtībām, to aizpilda ar nullēm:

for i:=1 **to** n **do** a[i]:=0;

Masīva elementa vērtību izvade realizējas arī ar cikla operatoru for izmantojot operatorus **write** un **writeln**.

Vektora izvadi stabiņu veidā realizē

for i:=1 **to** n **do writeln**(a[i]);

bet rindas veidā

for i:=1 **to** n **do write**(a[i],‘ ‘);

vai **for** i:=1 **to** n **do write**(a[i]:4);

Matricu izvads standarta formā (pa rindām un kolonnām) realizējas izmantojot operatoru writeln bez parametra:

for i:=1 **to** n **do**

begin for j:=1 **to** m **do**

write(a[i,j]:4); **writeln**; **end**;

(skaitļu formāts palīdz nolīdzināt kolonnas).

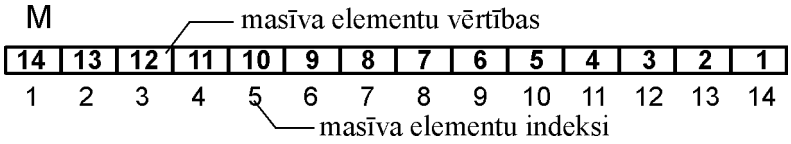
Masīvu aizpildīšanas piemēri

1. Katram masīva elementam ar piešķires operatoru tiek piešķirta konkrēta vērtības. Sastādīsim programmu, kura aizpilda masīvu A[1..7] ar nedēļas dienu nosaukumiem un izvada tos uz ekrāna.

program diena; **uses crt**; **var** A:**array**[1..7] of **string**; i :**integer**; **Begin clrscr**; A[1]:=’Pirmdiena’; A[2]:=’Otrdiena’; A[3]:=’Tresdiena’; A[4]:=’Ceturtdiena’; A[5]:=’Piektdiena’; A[6]:=’Sestdiena’; A[7]:=’Svetdiena’; **for** i:=1 **to** 7 **do writeln**(A[i]); **readln**; **end**.	{definē masīvu ar 7 **string** tipa elementiem} {Masīva elementus aizpilda ar nedēļas dienu nosaukumiem} {Masīva elementu vērtību izvadīšana uz ekrāna}

2. Masīva aizpildīšana ar cikla palīdzību Sastādīsim programmu, kura izveido skaitļu virkni no veseliem skaitļiem no **1** līdz **n** dilstošā secībā un izvada to uz ekrāna.

Gadījumā, ja n=14, tad masīvu grafiski var attēlot šādi:



Protams, šo uzdevumu var atrisināt, neizmantojot masīvu, bet katru skaitļu virknes elementu piešķirot konkrētam masīva elementam. Šajā gadījumā ir jāizveido 14 mainīgie.

Veidojot šāda tipa programmu tās apjoms pieaug atbilstoši virknes elementu skaitam. Tādēļ tiek izmantots masīvu uzpildīšanas paņēmiens izmantojot ciklu.

Varam atzīmēt, ka programmas **masivs_1** apjoms nemainīsies, ja elementu skaits pieaugs. Turpretī izmantojot iepriekšējo masīva uzpildes paņēmienu, programma būtiski pagarinās.

Sastādot programmu, kura paredzēta lielu masīvu apstrādei, ieteicams, vismaz programmas veidošanas sākumposmā, masīva elementiem piešķirt gadījuma skaitļu ģeneratora piedāvātās vērtības.

Pēc tam, kad masīva elementu apstrāde ir noprogrammēta, atrastas pieļautās kļūdas un konstatēts, ka programma strādā atbilstoši uzdevuma izvirzītajām prasībām, masīva elementu vērtību piešķiršanu gadījuma skaitļu veidā var aizstāt ar **read**, **readln** vai citiem operatoriem.

program masivs_1; **uses crt**; **var** M:a**rray**[1..14] of **Integer**; n,i:**Integer**; **begin ClrScr**;	{definē masīvu ar 20 **integer** tipa elementiem}

write(‘Cik ir masiva elementu? ‘); r**eadln**(n);
for i:=n **downto** 1 **do**
begin
write(‘Ievadi skaitļu virknes ‘,i,’
elementu: ‘);
readln(M[i]);
end;
for i:=1 **to** n **do** w**rite**(M[i]:4);
readln;
end.
	{Masīva vērtību ievade izmantojot ciklu}
	{Masīva vērtību izvade ekrānā, izmantojot ciklu}

3. Sastādīt programmu, kura aizpilda masīvu A[1..10] ar gadījuma skaitļiem no intervāla (3,10).

program masivs_2; **uses crt**; **var** A:**array**[1..10] of **integer**; i:**integer**; **begin clrscr**; **randomize**; **for** i:=1 **to** 10 **do** A[i]:=**random**(8)+3; **for** i:=1 **to** 10 **do** **write**(A[i]:4); **readln**; **end**.	{definē masīvu ar 10 **integer** tipa elementiem} {masīva aizpildīšana ar gadījuma skaitļiem*} {masīva elementu vērtību izvade uz ekrāna}

Piezīme.*{dators "iedomājas" skaitli no intervāla (3;10), jo **random**(8) iniciē skaitli no intervāla [0,7), bet pieskaitot 3, A[i] vērtību intervāls ir [3,10).

Masīvu apstrāde

Bieži nākas aprēķināt masīva elementu summu, to vidējo vērtību, noteikt maksimālās vai minimālās elementu vērtības un to kārtas numurus, nomainīt atsevišķu elementu vērtības ar citām u.t.t.

Viendimensiju masīva elementu summu nosaka:

```
s: =0
for i:= 1 to n do S:=S+a[i];
```

bet to reizinājumu:

```
r:=1;
for i:=1 to n do r:=r*a[i];
```

Piemērs: Meteoroloģiskā stacija ik pēc divām stundām reģistrē gaisa temperatūru. Sastādīt programmu, kura ļauj ievadīt iegūtos mērījumus un aprēķina diennakts vidējo gaisa temperatūru. Tātad diennaktī tiek veikti 12 neatkarīgi mērījumi.

program Meteo_stacija;
uses crt;
const n=12;
var sum,vid:**real**;
i: **integer**;
M:**array**[1..n] of **real**;
begin clrscr;
for i:=1 **to** n **do**
begin
write(‘Ievadi ’,i,’. merijumu: ');
readln(M[i]); **end**;
sum:=0;
for i:=1 **to** n **do** sum:=sum+M[i];
vid:=sum/n;
writeln(’Dienakts vidējā temperatūra ir ’,vid:3:1, ’deg.’);
readln;
end.
	{definē **n real** tipa elementu masīvu}
	{aizpilda masīva elementus ar mērījumu vērtībām}
	{nosaka mērījumu summu}
	{aprēķina diennakts vidējo temperatūru un izvada uz ekrāna}

Lai noteiktu veselu skaitļa masīva elementu skaitu, kuri ir pāru skaitļi, izmantojam programmas fragmentu:

k:=0;

for i:=1 **to** n **do**

if a[i] **mod** 2=0 **then** k:=k+1; {tiek pārbaudīts katrs masīva elements, vai tas dalās ar 2 bez atlikuma un fiksēts šādu elementu skaits}.

Bieži ir nepieciešama elementu meklēšana ar īpašu vērtību. Atrast elementu nozīmē noteikt tā kārtas numuru masīvā.

Tā, piemēram, lai noteiktu pirmo masīva elementu, kura vērtība ir 0, varam izmantot:

i:=0; {masīva elementa numurs}

repeat

i:=i+1;

until(a[i]=0) **or** (i=n) {vai nu tāds elements tiek atrasts vai arī nē (n atbilst pēdējam masīva elementam)}

if a[i]=0 **then writeln**(‘pirmais elements ar vērtību 0 ir ‘, i)

else writeln (‘tāda elementa nav’}.

Gadījumā, ja jāatrod visi elementi, kuru vērtības ir 0, būs jālieto cikla operators **for**. Šai nolūkā varam izmantot sekojošus operatorus:

Writeln(‘elementu, kuru vērtības ir 0, kārtas numuri ir ‘);

```
for i:=1 to n do
if a[i]=0 then write(i,' ');
```

Elementu maksimālās vērtības un tā kārtas numura noteikšanu realizē operatori:

max:=a[1]; k:=1	{meklēšanu sāk ar pirmo elementu}
for i:=2 **to** n **do**	{pārskata visus pārējos elementus sākot ar otro}
if a[i]>max **then**	
begin max:=a[i]; k:=i;	{fiksē katru atrasto vērtību, kura lielāka par visām iepriekšējām}
end;	

Šis programmas fragments nodrošina pirmā maksimālā elementa noteikšanu. Operatorā **if** izmantojot nosacījuma formu a[i]>=max, nosakām **pēdējo elementu ar maksimālu vērtību** (pie nosacījuma, ka ir vairāki elementi ar maksimālu vērtību).

Studentam ieteicams izveidot programmas fragmentu, kurš dod iespēju fiksēt **visus** elementus ar maksimālo vērtību.

Atsevišķu masīva elementu **vērtības nomaiņa** ar uzdoto vērtību realizējas ar operatoriem:

```
min:=80;
for i:=1 to n do
if a[i]<min then a[i]:=min;
```

Šāds algoritms atbilst situācijai, kad pārskatot n darbinieku algas tiek konstatēts, kuriem no darbiniekiem alga ir zem pieļaujamā minimuma (80) un tās tiek nomainītas ar šo fiksēto vērtību min.

Studentam ieteicams papildināt šo programmas fragmentu ar operatoriem, kas ļautu fiksēt cik šādu darbinieku ir kopskaitā un par cik būs jāpalielina kopējais algu fonds.

Masīva elementu pārvietošanas mērķis ir elementu vērtību apmaiņa vietām. Tātad elementu vērtības nemainās, bet mainās to atrašanās vieta. Lai realizētu šādu elementu vērtību pārvietošanu, tiek izmantots papildus mainīgais (**apmaiņas buferis**), kurā uz laiku tiek saglabāta viena no elementu vērtībām.

Lai realizētu masīva pirmā elementa vērtības apmaiņu ar piektā elementa vērtību, izmanto operatorus:

```
buf:=a[1];  a[1]:=a[5];  a[5]:=buf;
```

Masīvu sakārtošana

Kārtošana un **meklēšana** ir svarīgākie informātikas jēdzieni. Kārtošanas būtība ir viena tipa datu kopuma sakārtošanas process augošā vai dilstošā kārtībā atbilstoši kādai no pazīmēm.

Kārtošanas procesā masīva elementi tiek mainīti vietām tā, lai gala rezultātā to vērtības būtu sakārtotas augošā vai dilstošā kārtībā. Gadījumos, kad starp masīva elementiem ir elementi ar vienādām vērtībām, lieto jēdzienus nedilstoši un neaugoši masīvi.

Sakārtotos masīvos informācija atrodama daudz ērtāk un ātrāk. Masīvu kārtošanai ir izstrādāti daudzi algoritmi, kuri atšķiras, galvenokārt, ar to realizācijas ātrumu. Liela apjoma (1000 un vairāk elementu) masīviem racionāli izmantot „ātrās" kārtošanas metodes. Mazāka apjoma masīviem var tikt pielietoti paši vienkāršākie kārtošanas algoritmi.

Par vienu no vienkāršākajām kārtošanas metodēm uzskatāma **lineārā sakārtošanas metode**.

Šīs metodes būtība atbilstoši veicot masīva sakārtojumu nosacījumam par tā elementu vērtību neaugšanu, ir pārskatot visu masīvu atrast lielāko vērtību un apmainīt to ar pirmā elementa vērtību. Algoritms tiek atkārtots sākot ar otro masīva elementu, tā noteiktā maksimālā vērtība tiek apmainīta ar otrā masīva elementa vērtību.

Šādu procesu realizē programma:

```
program sakarto_1;
const  n=10;
M:array[1..n] of byte
=(9,11,12,3,19,1,5,17,19,3);                {masīvs tiek definēts tipizētas konstantes veidā}
var  i, j, buf, k:byte;
                a:integer;
begin
writeln('dotais masīvs');
for i:=1 to n do write(M[i]);
writeln;
a:=0;                                        { -  iterāciju skaitītājs}
for i:=1 to n-1 do
for j:=i+1 to n do                           {nepārbaudītās masīva daļas izmēra izmaiņa}
a:=a+1;
if M[i]<M[j] then                            {salīdzina i-to elementu ar nepārbaudītās masīva daļas elementiem}
begin
buf:=M[i];                                   {ja atrasts elements, kurš ir lielāks par i-to elementu, tos maina vietām}
M[i]:=M[j];
M[j]:= buf;  end;
for k:=1 to n do
write('  ',M[k]);
writeln(' ir veiktas',a,' iterācijas');  end;
end.
```

Konkrētā masīvā sakārtošanai neaugošā kārtībā bija nepieciešamas 190 iterācijas.

Studentam ieteicams patstāvīgi pārbaudīt cik iterācijas nepieciešamas, lai masīvu sakārtotu nedilstošā kārtībā.

„Burbuļu" metode. Tā ir viena no populārākajām masīvu sakārtošana metodēm un tās būtību raksturo algoritms, kura izpildes rezultātā „vieglākie" masīva elementi pakāpeniski „uzpeld". Metodes īpatnība ir tā, ka tiek salīdzināti blakus esošie elementi un, ja nepieciešams, samainīti vietām.

Šo algoritmu realizē programma:

program burbulis; **const** n =10; M:**array**[1..n] **of byte** =(9,11,12,3,19,1,5,17,19,3);	{masīvs tiek definēts tipizētas konstantes veidā}

begin
writeln (dotais masivs');
for i:=1 **to** n **do**
write(' ',M[i]);
writeln;
a:=0;
for i:=1 **to** n **do begin**
for j:=n **downto** i **do**
begin a:=a+1;
	{nepārbaudītās masīva daļas izmēra izmaiņa}

if M[j-1]<M[j] **then**
begin
buf:=M[i];
M[i]:=M[j];
M[j]:=buf;
for k:=1 **to** n **do**
write(' ',M[k]);
writeln(' ir veiktas',a,' iteracijas'); **end; end; end.**
	{salīdzina i-to elementu ar nepārbaudītās masīva daļas elementiem}
	{ja atrasts elements, kurš ir lielāks par i-to elementu, tos maina vietām}

Programmas algoritms paredz ciklisku visu elementu no pēdējā līdz otram caurskati un pārvietošanu pa kreisi, ja „kaimiņam" ir mazāka vērtība. Pēc katra cikla soļa caurskatāmās masīva daļas garums samazinās par vienu.

Dotā masīva sakārtošanai ar „burbuļu" metodi ir nepieciešams veikt 170 iterācijas.

Abas iepriekš analizētās masīvu sakārtošanas metodes uzskatāmas par vienkāršām, bet arī maz efektīvām. Daudz efektīgāka ir metode, kuras pamatā ir masīva sadalīšanas princips atsevišķās daļās un elementu apmaiņa starp šīm daļām. Metodes algoritms ir visai sarežģīts un tādēļ attiecīgo programmu šī kursa apjomā nerekomendējam.

Meklēšana nesakārtotā masīvā ir visai darbietilpīga, jo pamatojas uz visu elementu caurskati un salīdzināšanu ar uzdoto vērtību („atslēgu") līdz situācijai, kad salīdzināšanas rezultāts dod vērtību **true**. Apstrādājot liela apjoma informāciju parasti vispirms attiecīgo informācijas masīvu sakārto un pēc tam veic nepieciešamā elementa meklēšanu. Viena no meklēšanas efektīvām metodēm ir „dalīšanas uz pusēm" metode.

Metodes ideja – sakārtotu masīvu dala uz pusēm un pēc masīva vidējā elementa vērtības nosaka, kurā masīva daļā atrodas meklējamais elements. Nevajadzīgo masīva daļu atmet un attiecīgo algoritmu atkārto ar atlikušo masīva daļu. Dalīšanu atkārto tik ilgi, līdz masīva daļa sastāvēs tikai no viena elementa.

Šādu algoritmu realizē programmas:

```
program dala;
const n=20;
M:array[1..n] of byte
=(20,20,19,19,19,18,17,17,12,12,11,10,9,9,5,5, 3, 3, 2, 1);
var  atsl,i,pirm,ped:byte;
                    a: integer;
                  rez: boolean;
begin
writeln('dotais masīvs');
for i:=1 to n do
write('  ',M[i]);
```

```
writeln;
write('ievadām meklēšanas "atslēgu" '); readln(atsl);
a:=0; pirm:=1;  ped:=n; rez:=false;
repeat
i:=(pirm+ped) div 2;
if M[i]=atsl then rez:=true
else
if M[i]>atsl then pirm:=i+1;
else  ped:=i-1;
a:=a+1;
until (rez) or (pirm>ped);
if rez then
Writeln(' meklējamais elements atrodas',i,'-jā vietā');
else writeln('masīvā nav meklētā elementa');
writeln('meklēšana veikta ar ', a, ' iterācijām')
```

```
end.
```

Elementu izslēgšana un ievietošana masīvā.

Tā kā masīva elementi izvietoti atmiņas blakusesošās šūnās, tad nepieciešamības gadījumā bez papildus grūtībām likvidēt vai arī pievienot elementus var tikai masīva beigās. Visos citos gadījumos nāksies masīva elementus pārbīdīt.

Sekojošā programma veic masīva elementu ar konkrētu vērtību meklēšanu, to izslēgšanu no masīva un masīva sabīdīšanu.

```
program izslegt;
const n=10;
Mas:array[1..n] of byte=(15, 35, 27, 6, 3, 74, 42, 21, 43, 91);
var  x,                                   {izslēdzamais elements}
     j,                                   {cikla parametrs pārbīdot elementus}{skaitītājs}
     i,
     m: byte;                             {masīva elementu skaits pēc to sabīdīšanas}
begin
begin
writeln('sakotnejais masivs');
for i:=1 to n do
write('  ',Mas[i]);
writeln;
write('ievadiet izslēdzamo elementu'); readln(x);
```

m:=n; j:=0; i:=0;
repeat
i:=i+1;
while Mas[i]=x **do**
begin j:=i;
repeat
Mas[j]:=Mas[j+1];
j:=j+1; **until** j>=m;
m:=m-1;
end;
until i>=m;
	
	{caurskatīt visus masīva elementus}
	[kamēr kārtējais masīva elements ir vienāds ar **x,** nobīdām atlikušos elementus par **1** pozīciju pa kreisi}
	
	{samazinām masīva elementu skaitu}

for i:=m+1 **to** n **do** Mas[i]:=0; **writeln**(‘iegūtais masīvs ir’); **for** i:=1 **to** n **do write**(‘ ‘,Mas[i]); **writeln**; **end**.	{masīva ”astē” ievieto nulli} {ja ’astes’ izvadīšana nav vajadzīga, tad ciklā **n** jānomaina ar **m**}

Paškontroles uzdevumi

1. Dots programmas fragments:

```
var   A : array[1..30] of real;
      B : array[-5…5] of integer;
      C : array[11..25] of char;
```

Noteikt katram masīvam:

a) elementu skaitu;

b) kā pirmajam un pēdējam elementam ar operatora **readln** palīdzību piešķirt vērtības?

2. Nodefinēt atbilstošus masīvus, kuros var ierakstīt:

a) 35 kontroldarba atzīmes;

b) 20 automobiļu cenas;

c) 50 atbildes (ar “jā” vai “nē”) uz vienu jautājumu.

3. Pieņemsim, ka masīvs Nauda ir definēts šādi:

```
var   Nauda: array[1..3] of real;
```

Tabulā redzamas masīva Nauda elementu vērtības:

25,31	Nauda[1]
43,27	Nauda[2]
17,52	Nauda[3]

Kādas ir masīva Nauda elementu vērtības pēc doto programmas fragmentu izpildes?

a) A:= 59.32;

```
B:= A + Nauda[3];
                    Nauda[1]:= B;
```

b) **if** Nauda[3] < Nauda[1] **then begin**

Pag:= Nauda[3]; Nauda[3]:= Nauda[1]; Nauda[1]:= Pag;

end;

4. Dots viendimensiju masīvs, kurā ir 7 elementi. Sastādīt programmu, kura aprēķina pirmo četru elementu summu un pēdējo 3 elementu reizinājumu.

5. Sastādīt programmu, kura ar gadījuma skaitļu ģeneratora palīdzību aizpilda masīvu A[1..10], tad piešķir masīva A vērtības masīvam B[1..10] un aizpilda masīvu C[1..10] ar masīva A un masīva B attiecīgo elementu summām.

6. Sastādīt programmu, kura lietotāja ievadīto vārdu izdrukā apgrieztā secībā. Piemēram: Alberts – streblA.

7. Sastādīt programmu, kura aizpilda masīvu A[1..20] tikai ar 0, 1 vai 2. Sakārtot elementus masīvā tā, lai sākumā būtu visas nulles, tad visi vieninieki, un beigās divnieki.

FUNKCIJAS UN PROCEDŪRAS

Mūsdienu programmēšanas būtiska iezīme ir moduļveida principa izmantošana. Atbilstoši šim principam programmas atsevišķas daļas tiek veidotas kā neatkarīgi moduļi - apakšprogrammas. Tas ļauj izmantot divas būtiskas priekšrocības – konkrētu fragmentu var izmantot vairākkārt kā vienā, tā arī vairākās programmās un programmu var izveidot kā nelielu neatkarīgu fragmentu kopumu. Tādas programmas viegli lasīt, testēt un kompilēt. Apakšprogrammu pielietošana dod arī iespēju taupīt atmiņu. Apakšprogrammās izmantojamo mainīgo saglabāšanai atmiņā tiek rezervēta atmiņas daļa tikai uz šīs apakšprogrammas izpildes laiku. Pārējā laikā tā ir atbrīvota. Pascal-ā moduļveida princips tiek realizēts izmantojot **procedūras** un **funkcijas**. Kaut arī tām ir analoga nozīme un struktūra, tās atšķiras ar pielietošanas mērķi un veidu. Visas procedūras un funkcijas dalās divās grupās – **iebūvētās** un programmētāja **sastādītās**. Iebūvētās, jeb standarta procedūras un funkcijās ir daļa no programmēšanas valodas un var tikt izsauktas bez papildus apraksta. Biežāk lietojamās standartfunkcijas, to argumentu un rezultātu tipi tika dotas tabulā (8. lpp). Šo funkciju izmantošana būtiski atvieglo programista darbu. Diemžēl šādu funkciju skaits ir visai ierobežots un daudzas plaši lietojamas funkcijas nav atrodamas Pascal bibliotēkā (kaut vai trigonometriskā funkcija tgx). Šādos gadījumos programmētājam nākas izstrādāt individuālas nestandarta procedūras un funkcijas.

Procedūru un funkciju struktūra tiek veidota ar saviem individuāliem mainīgiem, kurus sauc par **lokālajiem mainīgajiem**. Lokālos mainīgos definē procedūru vai funkciju iekšienē, un tie ir izmantojami tikai tajās procedūrās un funkcijās, kurās tie ir definēti.

Pārējie mainīgie (**globālie mainīgie**) tiek definēti programmas galvenajā daļā un tie ir izmantojami kā programmas galvenajā daļā, tā arī visās procedūrās un funkcijās. Ikreiz, vēršoties pie procedūras vai funkcijas, lokālie mainīgie tiek izveidoti no jauna. Tas nozīmē, ka iepriekšējos aprēķinos iegūtās vērtības netiek saglabātas. Gadījumos, ja procedūrā vai funkcijā kādam no mainīgajam piešķirts tāds pats nosaukums kā kādam no globālajiem mainīgajiem, procedūras vai funkcijas iekšienē

tiek izmantots šis lokālais mainīgais.

Procedūra ir neatkarīga īpašā veidā noformēta programmas daļa, kurai tiek piešķirts atsevišķs vārds. Šim vārdam jābūt unikālam, tas nozīmē, ka to nedrīkst lietot citu programmas procedūru vai mainīgo apzīmēšanai. Procedūra var tikt daudzkārt pēc tās vārda izsaukta dažādās programmas vietās konkrētu darbību izpildei. Procedūru nevar lietot **kā operandu izteiksmē**. Uzrakstot šo vārdu programmas tekstā, tiek izsaukta jeb aktivizēta attiecīgā procedūra. Līdz ar procedūras izsaukšanu sākas tās izpilde - tiek izpildīti procedūrā ietilpstošie operatori. Pēc pēdējā operatora izpildes, tiek turpināta programmas izpilde no tās vietu, kurā tika izsaukta procedūra. Ja ir nepieciešams nodot informāciju no kādas programmas daļas uz procedūru, izmanto vienu vai vairākus parametrus. Tos norāda iekavās aiz procedūras nosaukuma. Šo parametru vērtības jānorāda izsaucot procedūru.

Procedūras vispārīgais pieraksts analogs programmas pierakstam. To veido nosaukums, apraksta un izpildāmā daļa. **Procedūrām un funkcijām** (atšķirībā no programmas) **nosaukums ir obligāta sastāvdaļa**. Tas sastāv no rezervētā vārda **procedure** vai **function** un to identifikatora (vārda), aiz kura apaļajās iekavās ieslēgts formālo parametru saraksts ar katra parametra tipa norādi. Piemēram:

procedure pakape(a:**real**, n:**integer**).

Izpildāmā daļa tiek ieslēgta operatoru iekavās **begin** un **end**, pie kam pēc **end** ir jālieto **semikols** nevis punkts kā tas ir programmas beigās.

Lai vērstos pie procedūras, tiek izmantots procedūras izsaukšanas operators. Tas sastāv no procedūras vārda un faktisko parametru saraksta apaļajās iekavās. Parametrus vienu no otra atdala ar komatu. **Parametru saraksts nav obligāts.**

Informācijas nodošana procedūrai realizējas trīs veidos:

1) ja uz procedūru nav nepieciešams nodot informāciju, tad definējot procedūru, aiz tās nosaukuma mainīgos nenorāda:

procedure <vārds>;

2) ja uz procedūru ir nepieciešams nodot tikai mainīgā vērtību (šī mainīgā vērtību procedūra neizmaina), tad aiz procedūras nosaukuma iekavās jānorāda jauns lokāls mainīgais un tā tips, kuram tiks piešķirta mainīgā vērtība:

procedure <vārds> (<mainīgais>:<tips>);

3) ja procedūrai jānodod pats mainīgais, ļaujot procedūras operatoriem mainīt šī mainīgā vērtību, tad aiz procedūras nosaukuma iekavās jāraksta **var** un jānorāda jauns lokāls mainīgais, kuram tiks piešķirta galvenajā programmas daļā izmantotā mainīgā vērtība:

procedure <vārds> (**var** <mainīgais>:<tips>).

Pēc procedūras izpildes nodotajam mainīgajam tiks piešķirta procedūras mainīgā vērtība.

Paskaidrosim to ar piemēru:

Jānis (mainīgais ar vērtību, kuru nodod procedūrai) aizbrauc uz ārzemēm (procedūra) mācīties. Tur Jāni jaunie draugi sauc par Džoniju (mainīgais iegūst jaunu pagaidu nosaukumu, pēc kura procedūrā tiek veikta mainīgā vērtību maiņa), bet atgriežoties mājās viņu atkal sauc par Jāni, un viss, ko viņš ir iemācījies ārzemēs, ir Jāņa rīcībā (programmas galvenās daļas mainīgajam saglabājas vērtība, kas iegūta

procedūras darbības gaitā).

Funkcija līdzīgi kā procedūra veido atsevišķu noslēgtu programmas daļu, bet atšķiras no pēdējās ar to, ka funkcijas darbības rezultāts galvenajai programmas daļai tiek nodots kā mainīgais. Tādēļ, definējot funkciju, ir jānosaka, kāds būs tās darbības gala rezultāta tips.

function <vārds> (<mainīgie>): <funkcijas tips>;

Piemērs. Sastādīsim programmu, kura dod iespēju izveidot tabulu collu pārveidošanai centimetros.

Tabulas izveidošanai izmantosim vienkāršotu procedūru (bez parametriem) horizontālas līnijas izveidei.

```
program col_cm;
var col: integer;
    cm: real;
procedure linija ;                {horizontālas līnijas veidošanas procedūra}
var i: integer;
begin
for i:= 1 to 20 do write ('–');
writeln;  end;
begin                             {pamatprogrammas sākums}
linija;                           {novelk augšējo līniju}
writeln(' │ collas │ cm │');      {tabulas galva}
for col:= 1 to 10 do              {izveido divas kolonnas ar desmit vērtībām katrā}
begin  cm:= 2.54*col;
writeln('│', col: 8,'│', cm:9:3,'│');
end;
linija
end.                              {tabulu noslēdzošā līnija}
```

Vēlams programmu papildināt tā, lai katru colla – centimetri pāri atdalītu horizontāla līnija.

Minētā piemērā uzskatāmi redzama viena no procedūras priekšrocībām - ērta līnijas izmēru un to veidojošo simbolu nomaiņa. Attiecīgie labojumi jāizdara tikai vienu reizi procedūrā, bet ne vairākkārt programmā.

Sastādīsim procedūru, ar kuru nomainot iepriekšējā piemērā izmantoto, iegūstam iespēju izveidot dažāda garuma horizontālas līnijas, pie kam šīs līnijas veidojot no atšķirīgiem simboliem.

```
procedure linija (gar:integer; simb :char) ;
var i: integer;
begin
for i:= 1 to gar do write( simb);
writeln;  end;
linija (20,' _ ');
linija (10,' # ');
linija (15,' ○ ');
```

{**gar** – līnijas garums simbolos, **simb** – simboli no kuriem veidojas līnija}

{Izmantojot galvenajā programmā šos trīs neatkarīgos procedūras izsaukumus, iegūstam trīs atšķirīgas līnijas}

Daudzu uzdevumu algoritmos nākas veikt skaitļu kāpināšanu vesela pozitīva skaitļa pakāpē (Pascal- ā nav šādas standartoperācijas). Kāpināšanu pozitīvā pakāpē iespējams veikt izmantojot procedūru:

```
procedure pakape (x: real;
n:byte; var rez: real);
var i: integer;
begin      rez:= 1:
for i:= 1 to n do rez:= rez*x;
end;
var p1,p2,p3: real;
begin
pakape(2,10,p1);
pakape(2,20,p2);
pakape(2,30,p3);
writeln('210 = ', p1: 4 );
writeln('220 = ', p2: 7 );
writeln('230 = ', 31: 10)
end.
```

{**x** – skaitlis, kurš tiek kāpināts **n** – tajā pakāpē, rezultāts būs parametrs – mainīgais **rez**}

{pamatprogramma}

{mainīgajiem **p1, p2** un **p3** tiek piešķirtas vērtības atbilstoši pakāpēm **10, 20** un **30**}

Ņemot vērā to, ka šī procedūra pamatprogrammai nodod tikai vienu rezultātu, šo procedūru var noformēt arī kā funkciju. Tas izdarāms sekojošā veidā:

function pak (x:**real**; n:**byte**): real;
i, p: **integer**;
begin p:= 1:
for i:= 1 **to** n **do begin** p:=p*x; pak:=p; **end;**
var x,n,y: **real**;
begin
writeln(‘ ievadiet kāpināmo skaitli un kāpinātāju’);
readln(x,n);
y:= pak(x,n);
writeln (x:4:2,’ pakāpē’, n:2, ‘ir’,y)
end.
	{**x** – skaitlis, kurš tiek kāpināts **n** – tajā pakāpē, rezultāts būs identifikatorā **pak**}
	
	{pamatprogramma}
	
	{mainīgajam y tiek piešķirta attiecīgā funkcijas vērtība}

Piemērs. Definēt skaitļa **n** faktoriāla (n!) funkciju.

```
function  fakt(n): real;
          fak,i: integer;
begin  fak:= 1;
     for i:= 1 to n do
     fak:= fak*i;
     fakt:=fak;  end;
```

Piemērs. Sastādīsim programmu, kura aizpilda masīvus **A[1..n]** un **B[1..m]** ar gadījuma skaitļiem, saskaita katra masīva elementu vērtības un nosaka, kura masīva elementu summa ir lielāka.

Sastādot šo programmu, izmantosim procedūras un funkcijas, kurām no programmas galvenās daļas ir jānodod viens mainīgais (masīvs **A** vai **B**) un viena mainīgā vērtība (masīva elementu skaits **n** vai **m**).

program salidzini_summu; **uses** Crt; **var** A,B:**array**[1..20] of **integer**; i,n,m:**integer**;	{globālie mainīgie }
procedure mas_uzpilde (**var** x: **array** of **integer**; y:**integer**); **begin** **for** i:=1 **to** y **do** x[i]:=**random**(10)+1; **end**;	{procedūra, kura aizpilda viendimensiju masīvu ar gadījuma skaitļiem} {no programmas galvenās daļas uz procedūru tiek nodoti divi mainīgie - **masīvs** un tā **elementu skaits**.}
procedure mas_izvade(**var** x:	{procedūra, kura izvada viendimensiju masīva elementu

array of **integer**; y: **integer**); **begin** **for** i:= 1 **to** y **do write**(x[i]:4); **writeln**; **end**;	vērtības: no programmas galvenās daļas uz procedūru tiek nodoti divi mainīgie - **masīvs** un tā **elementu skaits**}

function summa(**var** x: **array** of **integer**; y: **integer**): **integer**;
var sum:**integer**;
begin sum:=0;
for i:=1 **to** y **do**
sum:= sum+x[i];
summa:=sum;
end;
	{viendimensiju masīva elementu vērtību aprēķina funkcija }
	{no programmas galvenās daļas uz funkciju tiek nodoti divi mainīgie - masīvs un tā elementu skaits, bet pēc funkcijas izpildes programmas galvenajai daļai tiek nodots vesels skaitlis (**summa) -**
	funkcijas lokālais mainīgais, kura vērtība nav pieejama programmas galvenajai daļai}

begin ClrScr; **randomize**; **write**('Ievadi masiva A elementu skaitu n= '); **readln** (n); mas_aizpilde (A,n); mas_izvade(A,n); **writeln** ('Masiva A elementu summa ir:', Summa(A,n)); **write** ('Ievadi masiva B elementu skaitu: '); readln(m); Mas_aizpilde (B,m); Mas_izvade (B,m);	{**sākas pamatprogramma**} { izvēlas parametra **n** vērtību} {izsauc procedūru, nododot tai divus mainīgos –masīvu **A** nodod kā mainīgo ar vērtību, bet masīva elementu skaitu **n** nodod tikai kā mainīgā vērtību. Masīva **A** vērtības piešķirs procedūras mainīgajam **x**, bet masīva elementu skaita **n** vērtību piešķirs mainīgajam **y**. Pēc procedūras izpildes masīva **x** vērtības tiks piešķirtas masīvam **A**, bet mainīgā **n** vērtība netiks izmainīta. {izsauc procedūru, kas izvada uz ekrāna masīva **A** vērtības} {izsauc funkciju **summa**, kas atgriež masīva **A** elementu summu, kuru **writeln** operators izvada uz monitora ekrāna.} {analogas manipulācijas tiek veiktas arī ar masīvu **B**}

```
writeln ('Masiva B elementu summa ir: ', Summa(B,m));
if summa (A,n) > summa (B,m) then
writeln('Masiva A elementu summa ir lielaka par masiva B elementu summu')
else
if summa (A,n)<Summa (B,m) then
writeln('Masiva A elementu summa ir mazaka par masiva B elementu summu')
else
writeln('Masiva A elementu summa ir vienada ar masiva B elementu summu')
end.
```

{salīdzina masīvu elementu summas (funkcijas summa(A,n) un summa(B,m)}

Paškontroles uzdevumi

1. Kuri dotajā programmā ir lokālie un kuri globālie mainīgie?

```
program druka;
uses crt;
var  j: integer;
  A,B: char;
 Burti: array[1..100] of char;
procedure Rekins(Skaits: integer);
var  i,Burts: char;
begin     Burts:= '*';
for i:=A to B do
if Burti[Ord(i)] > Burts then
Skaits:= Skaits + 1;
writeln('Skaits = ', Skaits);
end;
begin  ClrScr;
randomize;
A:='A'; B:='z';
for j:=1 to 100 do
```

```
Burti[j]:= chr(random(35)+30);
Rekins(0);
end.
```

2. Kas tiks izvadīts monitora ekrānā pēc dotās programmas izpildes?

```
program Druka;
uses crt;
var    A,B: char;
procedure Raksta(A: integer);
const  B= 5;
begin   writeln(A:3, B:3); end;
procedure Skaita;
var A: char;
begin  A:= B;     B:= '*';
writeln(A: 3, B:3);  end;
begin  ClrScr;
A:= 'a';  B:= 'b';
Raksta(7); writeln(A:3, B:3);
Skaita;   writeln(A:3, B:3);
readln;
end.
```

3. Atrast kļūdas programmas fragmentos un paskaidrot tās.

a) **function** Liels(A,B: **integer**): **integer**;

begin if A> B **then** Liels:= A; **end**;

b) **function** Summa(Skaits: **integer**): **integer**;

var L: **integer**;

begin Summa:= 0;

for L:=1 **to** Skaits **do** Summa:= Summa + L;

end;

4. Noteikt summas 1!+2!+3!+4!+5! **vērtību.**

5. Noteikt summas $1/a + 1/a^2 + 1/a^3 + 1/a^4 + ... + 1/a^n$ vērtību patvaļīgai naturāla skaitļa **n** vērtībai.

6. Noteikt un **izvadīt tabulas veidā** summas 1 +1/2+1/3 +1/4 + ... +1/n vērtības naturāla skaitļa **n** vērtībām intervālā (3, 10).

$1 + 1/2^2 + 1/3^2 + 1/4^2 + ... + 1/n^2$

$1/3^2 + 1/5^2 + 1/7^2 + ... + 1/(2n+1)^2$

1! +2!/(1+1/2) +... +n!/(1 +1/2 + ... 1/n)

7. Izveidot funkcijas y = $3x^5 + 2x^3 - 7x^2 + x - 8$ **vērtību tabulu** intervālā (1,15) ar soli

8. Izveidot dotās programmas, kura nosaka ar virsotņu koordinātēm uzdota **četrstūra ABCD laukumu,** komentārus. Papildināt programmu tā, lai tiktu izvadīti atsevišķu trīsstūru laukumi. Pārbaudīt risinājumu izvēloties citu četrstūra dalījumu. Pārveidot programmu tā, lai tiktu izmantotas funkcija un procedūra.

```
program laukums4stur;
uses crt;
var xA,yA,xB,yB,xC,yC,xD,yD:integer;
a,b,c,d,e,AB,BC,CD,AD,DB,L,p1,p2,T1,T2:real;
begin
clrscr;
writeln (Ievadi koordinates xA, yA, xB, yB, xC, yC, xD, yD ');
writeln;
write('  xA= '); readln(xA);
write('     yA= '); readln(yA);
write('          xB= '); readln(xB);
write('               yB= '); readln(yB);
write('               xC= '); readln(xC);
write('          xC= '); readln(yC);
write('     xD= '); readln(xD);
write('  yD= '); readln(yD);
AB:=sqrt(sqr(xB-xA)+sqr(yB-yA));
BC:=sqrt(sqr(xC-xB)+sqr(yC-yB));
CD:=sqrt(sqr(xD-xC)+sqr(yD-yC));
AD:=sqrt(sqr(xD-xA)+sqr(yD-yA));
DB:=sqrt(sqr(xB-xD)+sqr(yB-yD));
p1:=(AB+DB+AD)/2;
     T1:=sqrt(p1*(p1-AB)*(p1-DB)*(p1-AD));
p2:=(DB+CD+BC)/2;
     T2:=sqrt(p2*(p2-DB)*(p2-CD)*(p2-BC));
L:=(T1+T2);
writeln;
write('Jusu ievadita 4-stura laukums ir ',L:4:2);
end.
```

9. Noteikt vienādojuma f(x)= 0 reālās saknes norādītajā intervālā (a, b). Izveidot programmas komentāru. Veikt rezultātu pārbaudi.

```
program saknes;
     var  a,b,c,eps,fa,fc: real;
function f(x:real): real;
     begin   f:=sqr(x) - 2;   end;
begin
     read(a,b,eps);
fa:= f(a);
while abs(b-a) > eps do
begin
c:= (a+b)/2;
fc:= f(c);
if fa*f c< 0 then b:= c
    else  begin
a:= c;  fa:= f(c);  end;
end;
write(c);
end.
```

10. Noteikt laukumu figūrai, kuru ierobežo līkne y(x), abscisu ass un taisnes x=a un x=b ar uzdotu precizitāti p. Izveidot programmas komentāru.

```
program laukums_y;
label 1;
var  a,b,s1,s2,x,d,p: real;
                      n: integer;
function y(x: real): real;
begin
      y:= Sqrt (100-Sqr(8-x))-6;
end;
begin
write('apaksseejaa robezza a=  ');
readln(a);
write(' augsseejaa  robezza b=  ');
readln(b);
write(' precizitaate   p=  ');
readln(p);
n:= 1;
```

```
s2:= 0;
1: n:= n+1;
x:= a;
d:= (b-a)/n;
s1:= 0;
while x < (b+d/2) do
begin
s1:= s1+y(x);
x:= x+d;
end;
s1 := s1*d;
if abs(((s1-s2)*100)/s1) > p then
begin
s2:= s1;
goto 1
end;
write('  laukums s=');
writeln(s1);
write('  dalijumu skaits n=  ');
writeln(n);
end.
```

11. Noteikt vienādojuma F(x)= 0 **saknes** un aprēķināt funkcijas F(x) un abscisu ass ierobežoto pozitīvo **laukumu,** ja $F(x)=x^5-5x^4+3x^3-$

$-x^2+x+5$.

GRAFISKAIS REŽĪMS GRAPH

Grafisku attēlu veidošanu jeb "zīmēšanu ekrānā" nodrošina speciālas procedūras un funkcijas. Displeja ekrānā attēli veidojas no atsevišķiem viens otram tuvu izvietotiem spīdīgiem punktiem. Katra punkta krāsu nosaka programma. Iekrāsotie punkti ekrānā izvietojas rindās un kolonnās. Iepriekšējos piemēros programmas darbības rezultāti ekrānā tika izvadīti burtu, ciparu vai cita veida simbolu formā. Arī burtu un ciparu formas ekrānā tiek izveidotas ar punktiem. Operatori **Write** un **Writeln** izvada ekrānā rezultātu simbolu formā tā, ka programmētājam nav jādomā, kāds izskatīsies burts vai cipars. Minētie operatori darbojas tā saucamajā ekrāna teksta režīmā. Šajā režīmā ekrāns sadalīts neredzamās rūtiņās, kur katrā rūtiņā var ierakstīt vienu simbolu. Lai veidotu grafiskus attēlus ("zīmējumus"), programmai ekrāns vispirms jāpārslēdz grafiskajā režīmā. Zīmēšanai var izmantot vairāk nekā 50 procedūru un funkciju, kuras iekļautas grafiskajā standartbibliotēkā **Graph.** Šīs bibliotēkas pievienošanu TurboPascal programmai veic ar komandu **Uses Graph.**

Programmas pāriešana grafiskajā režīmā

Dators pēc programmas Turbo Pascal aktivizēšanas strādā teksta režīmā, tāpēc jebkurai programmai, kura izmanto Turbo Pascal grafiskās iespējas, ir jānodrošina sadarbība ar monitora grafisko adapteri. Pēc programmas darba beigšanas Turbo Pascal pārslēdz atpakaļ teksta režīmu. Grafisko procedūru darba uzturēšana ar konkrēto adapteri tiek panākta ar attiecīgā grafiskā draivera pieslēgšanu. Draiveris ir speciāla programma, kas vada dažādu procesu norisi datorā. Grafiskais draiveris vada displeja adapteri grafiskajā režīmā. Grafiskajā režīmā monitora ekrāns tiek aplūkots, kā ļoti tuvu viens otram novietotu punktu (pikseļu) kopums, kuru krāsu var ieregulēt ar programmas palīdzību. Konkrētā adaptera grafiskās iespējas ir atkarīgas no monitora, t.i., pikseļu skaits ekrānā, kā arī krāsu un nokrāsu skaits, ko tie var iegūt.

Procedūra InitGraph

Programmā adaptera pārslēgšanu grafiskajā režīma nodrošina operators **InitGraph**.

Tā vispārīgais pieraksts:

InitGraph(var <Draiveris>, <Tips>:integer; <Taka>:string);

Mainīgais <Draiveris>, kurš ir Integer tipa, nosaka grafiskā draivera tipu. Mainīgais <Tips>, kurš arī ir Integer tipa, nosaka grafiskā adaptera darba režīmu, bet String tipa mainīgais <Taka> satur draivera faila nosaukumu vai adresi, kur to atrast.

Procedūras izsaukšanas momentā uz viena no diskiem jāatrodas failam (parasti katalogā BGI), kas satur vajadzīgo grafisko draiveri. Procedūra šo draiveri ielādē operatīvajā atmiņā un pārslēdz adapteri grafiskajā darba režīmā. Draivera tipam jāatbilst grafiskā adaptera tipam. Lai norādītu draivera tipu ir izdalītas speciālas konstantes (skat. pielikumā). Lai programmu varētu izmantot uz dažādu modeļu IBM tipa datoriem, ieteicams mainīgajam <Draiveris> piešķirt konstantu vērtību **Detect**. **Detect** norāda, ka programma atbilstošo adapteri piemeklē automātiski. Ja adaptera tips nav zināms vai arī programma domāta darbam ar jebkuru adapteru, procedūrā **InitGraph** izmanto automātisko draivera tipa noteikšanu.

Piemēram:

```
Uses Graph;   {programmai pieslēdz grafisko standartbibliotēku Graph}
var
Draiveris, Tips: Integer;
begin
Draiveris:=Detect;    {automātiskais draivera tipa meklēšanas režīms}
InitGraph (Draiveris, Tips, 'C:\TP\BGI');
```

Pēc šo operatoru izpildes tiek uzstādīts ekrāna grafiskais darba režīms un mainīgajiem <Draiveris> un <Tips> tirk piešķirtas vērtības (veseli skaitļi), kas nosaka draivera un adaptera darba režīma tipus.

Tā, piemēram, ja draiveris CGA.BGI atrodas katalogā TP\BGI uz diska C un tiek uzstādīts darba režīms 320x200 pikseļi, tad vēršanās pie procedūras būs šāda:

```
uses Graph;          {programmai pieslēdz grafisko standartbibliotēku Graph}
var
```

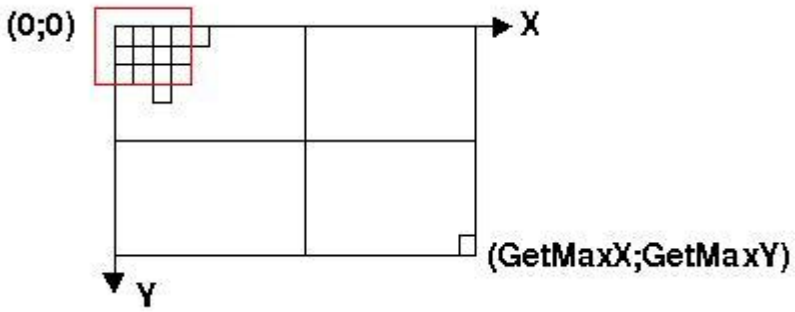
```
        Draiveris, Tips: Integer;
    begin
        Draiveris:=CGA;        {draivera tips}
        Režīms:=CGAC2;         {adaptera darba režīms}
    InitGraph (Draiveris, Tips, 'C:\TP\BGI');
```

Lai izbeigtu adaptera darbu grafiskajā režīmā un atjaunotu ekrānā teksta režīmu, tiek lietota procedūra **CloseGraph**

Šīs procedūras vispārīgais pieraksts ir

CloseGraph;

Daudzas grafiskās funkcijas un procedūras izmanto **aktīvās pozīcijas norādīšanu ekrānā**, kura atšķirībā no teksta kursora nav redzama. Aktīvās pozīcijas norāde, kā arī jebkura citas koordinātes ekrānā tiek uzdotas saskaņā ar nosacījumu ka kreisā augšējā stūra koordinātes ir 0,0. Tādā veidā horizontālā ekrāna koordināte palielinās no ekrāna kreisā stūra pa labi, bet vertikālā – no augšas uz leju.



Funkcija GetMaxX un GetMaxY

Šīs funkcijas nosaka maksimālo horizontālo (**GetMaxX**) un maksimālo vertikālo (**GetMaxY**) koordināti.

Piemēram:

```
    Program max_koord;
    Uses Graph;
    var a, b:Integer;
    begin
        a:=Detect;
    InitGraph(a, b, ' c:\tp\bgi  ');
    writeln(GetMaxX,' ', GetMaxY);   {Izvada monitorā ekrāna maksimālo X koordināti un maksimālo Y koordināti}
    readln;
    CloseGraph;
    end.
```

Maksimālās koordinātes dažādos adaptera režīmos var redzēt pielikumā.

Procedūra PutPixel. Šī procedūra izvada norādītās krāsas punktu norādītajās monotora ekrāna koordinātēs.

Tās vispārīgais pieraksts:

PutPixel(<X>, <Y>,<Krāsa>:**Integer**);

Šeit <X>, <Y>- punkta koordinātes, <Krāsa> – punkta krāsa. Koordinātes tiek uzdotas vadoties pēc ekrāna kreisā augšējā stūra.

Krāsu konstantes. Visas krāsas ir numurētas ar veseliem skaitļiem no 0 līdz 15:

0 (**Black**) – melns

1 (**Blue**) – zils

2 (**Green**) – zaļš

3 (**Cyan**) – debeszils

4 (**Red**) – sarkans

5 (**Magenta**) – violets

6 (**Brown**) – brūns

7 (**LightGray**) – gaiši pelēks

8 (**DarkGray**) – tumši pelēks

9 (**LightBlue**) – gaiši zils

10 (**LightGreen**) – gaiši zaļš

11 (**LightCyan**) – gaiši debeszils

12 (**LightRed**) – gaiši sarkans

13 (**LightMagenta**) – gaiši violets

14 (**Yellow**) – dzeltens

15 (**White**) – balts

Katrai krāsai Turbo Pascal ir izveidota konstante, kuras nosaukums ir krāsas angliskais nosaukums (skat. iekavās). Lai atvieglotu darbu ar krāsām, ieteicams programmas sākumā nodefinēt savas konstantes ar latviskiem krāsu nosaukumiem.

Piemēram:

```
Const
    melns = 0;     {black}
    balts = 15;    {white}
    sarkans = 4;   {red}
```

Procedūra Line.

Šī procedūra izvada ekrānā līniju ar norādītajām sākuma un beigu punktu koordinātēm.

Tās vispārīgais pieraksts:

Line(<X1>, <Y1>, <X2>, <Y2>:**Integer**);

Šeit (<X1>,<Y1>) līnijas sākuma punkta koordinātes, bet (<X2>,<Y2>) – līnijas beigu punkta koordinātes.

Procedūra MoveTo.

Procedūra **MoveTo** novieto kursoru punktā (x,y).

Vispārīgais pieraksts:

MoveTo(<X>, <Y>:**Integer**);

Procedūra LineTo

Procedūra **LineTo** izvada ekrānā līniju, kuras sākuma punkta koordinātes ir vienādas ar ekrāna aktīvās pozīcijas koordinātēm, bet beigu punkta koordinātes tiek uzdotas.

Vispārīgais pieraksts:

LineTo(<X>, <Y>:**Integer**);

Šeit <X>, <Y> – jaunās aktīvās pozīcijas koordinātes, kas vienlaicīgi ir arī līnijas beigu koordinātes.

Piemērs. Sastādīsim programmu, kura ekrānā uzzīmē trīsstūri.

```
Program Trissturis;  {piešķir programmai nosaukumu Trissturis}
Uses Graph;          {pieslēdz grafisko standartbibliotēku Graph}
var a, b:integer;    {definē mainīgos a un b}
begin
a:=Detect;     {mainīgajam a piešķir automātisko draivera tipa meklētāju}
InitGraph(a, b, 'c:\tp\bgi'); {uzstāda grafisko darba režīmu}
Line(20, 20, 60, 60);          {uzzīmē līniju}
MoveTo(60,60);                 {novieto kursoru punktā (60,60)}
LineTo(100, 20);               {zīmē līniju no punkta (60,60)}
LineTo(20,20);                 {zīmē līniju no punkta (100,20)}
readln;
CloseGraph;                    {izslēdz grafisko darba režīmu}
end.
```

Šī programma uzzīmē trīsstūri, kura virsotņu koordinātes attiecīgi ir (20,20), (60,60) un (100,20).

Procedūra Rectangle izvada uz ekrāna taisnstūri ar norādītajām koordinātēm.

Vispārīgais pieraksts:

Rectangle(<X1>, <Y1>, <X2>, <Y2>:**Integer**);

Šeit (<X1>,<Y1>) ir taisnstūra kreisā augšējā stūra koordinātes,

(<X2>,<Y2>) – taisnstūra labā apakšējā stūra koordinātes.

Procedūra Circle izvada uz ekrāna riņķa līniju.

Tās vispārīgais pieraksts:

Circle(<X>, <Y>, <R>:**Integer**);

kur <X> un <Y> – riņķa centra koordinātes, bet <R> – riņķa līnijas rādiuss pikseļos.

Procedūra SetLineStyle.

Šī procedūra norāda jaunu līnijas zīmēšanas veidu.

Vispārīgais pieraksts:

SetLineStyle(<Tips>, <Raksts>, <Biezums>:**Integer**);

Šeit <Tips>, <Raksts>, <Biezums> – attiecīgi ir līnijas tips, līnijas raksts un līnijas biezums. Līnijas tips var tikt uzdots ar attiecīgajām konstantēm:

SolidLn =0; {Nepārtraukta līnija}

DottedLn =1; {Punktota līnija}

CenterLn =2; {Svītr - punktu līnija}

DasheLn =3; {Pārtraukta līnija}

UserBitLn =4; {Līnijas rakstu nosaka lietotājs}

Parametrs <Raksts> ir paredzēts tikai līnijām, kuras rakstu nosaka lietotājs. Lietojot standarta līniju tipus, mainīgā <Raksts> vērtība ir nulle.

Parametrs <Biezums> var pieņemt tikai vienu no divām vērtībām:

NormWidth = 1; {Līnija ir 1 pikseli bieza}

vai

ThickWidth = 3; { Līnija ir 1 pikseli bieza}

Procedūra SetColor.

Šī procedūra uzstāda krāsu nākošajām izvadāmajām līnijām un simboliem.

Vispārīgais pieraksts:

SetColor(<Krasa>:**Integer**);

Procedūra SetBkColor

Šī procedūra nosaka ekrāna fona krāsu.

Vispārējais pieraksts:

SetBkColor(<Krasa>:**Integer**);

Piemērs:

```
Program linijas;        {piešķir programmai nosaukumu linijas}
Uses Graph;             {pieslēdz grafisko standartbibliotēku Graph}
Const                   {nodefinē krāsu konstantes}
        peleks=7;
        zals=2;
        sarkans=4;
        dzeltens=14;
        balts=15;
var a, b, i :integer;   {nodefinē mainīgos a, b un i}
begin
        a:=Detect;      {mainīgajam a piešķir automātisko draivera tipa meklētāju}
        InitGraph(a,b,'c:\tp\bgi '); {uzstāda grafisko darba režīmu}
        {pārbauda, vai ir izdevies sekmīgi pārslēgties grafiskajā režīmā, ja nē – programma
        tiek apturēta un izvadīts paziņojums}
        if GraphResult <> grOk then begin
        writeln('Kluda grafiskaja rezima …');
        readln;
        end;
        ClearDevice;                    {notīra ekrānu}
        SetBkColor(peleks);             {nomaina fona krāsu}
        SetColor(balts);                {nosaka sekojošo līniju krāsu}
        SetLineStyle(1,0,1);            {nosaka sekojošās līnijas stilu}
        Rectangle(10,10,GetMaxX-10,GetMaxY-10);     {zīmē taisnstūri}
        SetColor(dzeltens);             {nosaka sekojošo līniju krāsu}
        SetLineStyle(0,0,3);            {nosaka sekojošās līnijas stilu}
        Rectangle(20,20,GetMaxX-20,GetMaxY-20);     {zīmē taisnstūri}
        SetColor(balts);                {nosaka sekojošo līniju krāsu}
        SetLineStyle(1,0,1)             {nosaka sekojošās līnijas stilu}
        Rectangle(30,30,GetMaxX-30,GetMaxY-30);     {zīmē taisnstūri}
        SetColor(sarkans);              {nosaka sekojošo līniju krāsu}
for i:=0 to 3 do begin                  {cikla sākums}
        SetLineStyle(i,0,3);            {nosaka sekojošās līnijas stilu}
        Line(70,100*(i+1),GetMaxX-70,100*(i+1));    {zīmē līniju}
        end;            {cikla beigas}
readln;
```

CloseGraph; {izslēdz grafisko darba režīmu}

end.

Studentam ieteicams pārrakstīt šo programmu Turbo Pascal vidē, pievienojot jaunas krāsu konstantes, izmainīt programmā līniju krāsas un stilus un, iedarbinot programmu, izsekot izmaiņām ekrānā.

Procedūra SetFillStyle.

Šī procedūra uzstāda noslēgtas figūras aizkrāsojuma stilu, t.i., tā rakstu un krāsu.

Vispārīgais pieraksts:

SetFillStyle(<Pildījums>, <Krasa>:**Integer**);

Šeit <Pildījums> – aizpildījuma raksts, <Krasa> – aizpildījuma krāsa.

Aizpildījuma rakstam var izmantot iebūvētās konstantes:

EmptyFill =0;	{aizpilda ar fona krāsu}
SolidFill =1;	{blīvs pildījums}
Line Fill =2;	{aizpildīts ar ----------}
LtSlashFill =3;	{aizpildīts ar /////////////// }
SlashFill =4;	{aizpildīts ar biezām ////////// }
BkSlashFill =5;	{aizpildīts ar biezām \\\\\\\\\\ }
LtBkSlashFill =6;	{aizpildīts ar \\\\\\\\\\\\\\ }
HatchFill =7;	{aizpildīts ar ++++++}
XhatchFill =8;	{aizpildīts ar xxxxxxxxxx }
InterleaveFill =9;	{rūtains pildījums}
WideDotFill =10;	{pildīts ar retiem punktiņiem}
CloseDotFill =11;	{pildīts ar blīviem punktiņiem}

Procedūra FloodFill

Procedūra **FloodFill** aizkrāso patvaļīgu noslēgtu figūru, iepriekš noteiktajā stilā.

Vispārīgais pieraksts:

FloodFill(<X>, <Y>, <Mala>:**Integer**);

kur <X>, <Y> – koordinātes jebkuram punktam, kas atrodas iekš patvaļīgas noslēgtas figūras. <Mala> – kontūrlīnijas krāsa..

Gadījumos, kad figūra nebūs noslēgta ar uzdoto krāsu, tiks aizpildīts viss ekrāns.

Procedūra Bar

Šī procedūra zīmē aizkrāsotu taisnstūri, iepriekš norādītajā stilā.

Vispārīgais pieraksts:

Bar(<X1>, <Y1>, <X2>, <Y2>:**Integer**);

Šeit (<X1>,<Y1>) taisnstūra kreisā augšējā stūra koordinātes, bet (<X2>,<Y2>) – labā apakšējā stūra koordinātes. Procedūra **Bar** aizkrāso tikai pašu taisnstūra veida laukumu, bet ne tā kontūrlīniju.

Procedūra FillEllipse

Šī procedūra zīmē aizkrāsotu elipsi, iepriekš norādītajā stilā.

Vispārīgais pieraksts:

FillEllipse(<X>, <Y>, <RX>, <RY>:**Integer**);

Šeit <X> un <Y> elipses centra koordinātes, bet <RX> un <RY> –elipses horizontālais un vertikālais rādiuss pikseļos.

Ja <RX>=<RY>, tad procedūra **FillEllipse** zīmē aizkrāsotu riņķi.

Procedūra ClearDevice

Šī procedūra notīra ekrānu.

Vispārējais pieraksts:

ClearDevice;

Procedūra OutTextXY izvada uz ekrāna teksta rindu norādītajās koordinātēs.

Vispārīgais pieraksts:

OutTextXY(<X>, <Y>:**Integer**; <Teksts>:**String**);

Šeit <X>,<Y> – norāda koordinātes, no kuras vietas sākt tekstu izvadīt, bet <Teksts> ir izvadāmais teksts.

Piemērs. Sastādīt programmu, kura ekrānā izvada 30 rūtainus aplīšus.

```
Program Grafika;              {piešķir programmai nosaukumu Grafika}
uses Graph;                   {pieslēdz grafisko standartbibliotēku Graph}
var a, b:integer;
    k, x, y:integer;          {nodefinē mainīgos}
begin
a:=Detect;                    {mainīgajam a piešķir automātisko draivera tipa  meklētāju}
InitGraph(a, b, 'c:\tp\bgi'); {uzstāda grafisko darba režīmu}
{pārbauda, vai ir izdevies sekmīgi pārslēgties grafiskajā režīmā, ja nē – programma …}
                              {…tiek apturēta un izvadīts paziņojums}
if GraphResult <> grOk then begin
writeln('Kluda grafiskaja rezima …');
readln;
end;
ClearDevice;                  {notīra ekrānu}
Randomize;                    {skat. 3. nodarbībā}
SetBkColor(8);                {fonu nokrāso tumši pelēku}
```

```
for k:=1 to30 do                    {cikla sākums “Izvadīt monitorā 30 aplīšus”}
        begin
        x:=Random(GetMaxX);         {x piešķir gadījuma vērtību}
        y:=Random(GetMaxY);         {y piešķir gadījuma vērtību}
        SetColor(2);                {uzzīmē zaļu apli}
        Circle(x, y, 20);
        SetFillStyle(7, 14);        {laukumu, kuru ierobežo zaļa līnija aizkrāso …}
        FloodFill(x,y,2);           {… dzeltenu un sarūto}
        end;                        {cikla beigas}
SetColor(4);                        {Izvada tekstu sarkanā krāsā}
OutTextXY(275, 250, ‘Mani aplīši !!!’);
readln;
CloseGraph;                         {Izslēdz grafisko darba režīmu}
end.                                {programmas beigas}
```

Uzdevums patstāvīgajam darbam:

nomainiet aplīšiem krāsu un aizpildījuma veidu;

pārveido programmu tā, lai tā zīmētu 20 aplīšus;

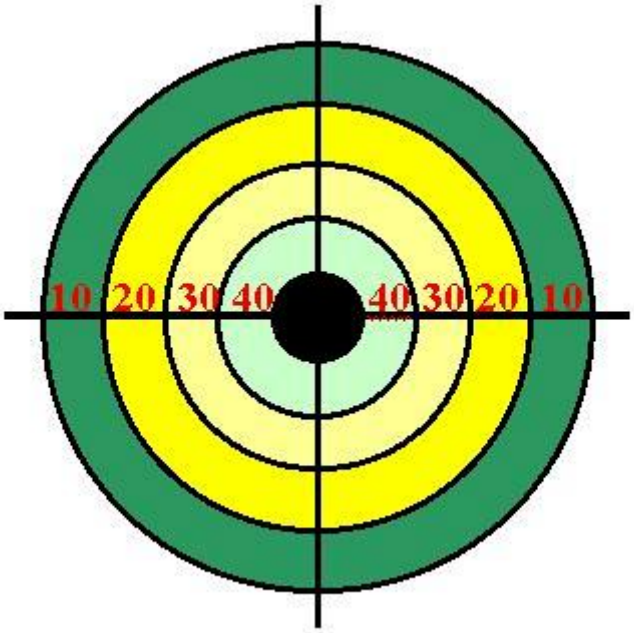
pārveido programmu tā, lai tā zīmētu 30 kvadrātiņus.

Uzdevumi

1. Sastādīt programmu, kas uzzīmē un izvada monitorā klaunu.

2. Sastādīt programmu, kas monitorā izvada 20 dažāda lieluma un dažāda stila kvadrātus.

3. Sastādīt programmu, kas monitorā izvada šaušanas mērķi, zīmējot 3 pikseļus biezas līnijas.



4. Sastādīt programmu, kas ļauj dialoga formā veidot zīmējumus. Piemēram ievadot 2, tiek zīmēta 20 pikseļus gara vertikāla līnija uz leju, ievadot 8 – uz augšu, 4 – pa kreisi, 6 – pa labi. Nospiežot 5, programma savu darbu beidz.

Piezīmes:

1) Pa horizontāli pikseļi izvietoti tuvāk viens otram, nekā pa vertikāli, tāpēc pa labi vai pa kreisi var zīmēt īsāku līniju (“spert” īsāku “soli”) nekā uz augšu vai leju.

2) Pirms zīmēt līniju, jāpārbauda (ar **if**.. operatoru) vai netiks zīmēts “ārpus” ekrāna.

5. Sastādīt programmu, kas ekrānā izvada n aplīšus (n ievada lietotājs), pēc tam līdzīgi kā 4. uzdevumā dialoga formā lietotājam ir iespējams apstaigāt (“savākt”) visus aplīšus.

PIELIKUMI

Pielikums I. Paškontroles uzdevumu atbildes.

Pk1-1. Gadījumā, ja d=11 un e=3, iegūstam a) **3.67** b) **3** c) **2**.

Pk1-2. a) **v= 1.2855146294E+00** b) **y izmainas par 5.7500000000E+03**

Pk1-3. Uzdevuma atrisināšanai ieteicams izmantot skaitļu dalīšanu **mod** ar 7 un ar 3.

Pk1-4. Veicot uzdevumu svarīgi atcerēties, ka lenķu vērtības grādoj jāpārvērš radiānos un ka trigonometriskās funkcijas tgx Pascal standartfunkciju bibliotēkā nav un tās aprēķins jāorganizē programmētājam.

Pk1-5. Aprēķinu veikšanai jāizmanto sakarības: rinķa līnijas garums - $\mathbf{2\pi r}$, rinķa laukums - $\mathbf{\pi r^2}$, kur **r** ir rinķa radiuss.

Pk1-6. Diennaktī ir 86400 sekundes. Svarīgi ņemt vērā, ka iegūtais skaitlis pārsniedz **integer** tipa skaitļu diapozonu**.**

Pk1-7. Iegūstam simbolus **♣, ♀ , ▲, ■, ╚.**

Pk1-9. Izmantojam kvadrātvienādojuma sakņu noteikšanas formulas. Pārbaudi veicam ievietojot iegūtās sakņu vērtības vienādojumā un nosakot cik precīzi tās apmierina vienādojumu.

Pk1-11. Trīsciparu skaitļa pirmo ciparu varam noteikt veicot dalīšanu **div** ar **100**, otro ciparu veicot ši dalījuma atlikuma dalīšanu **div** ar **10.**

Pk2-1. a) 1; b) 2; c) 0.

Pk2-2. a) trūkst operatoru iekavu **begin** un **end**;
b) lieks ir semikols pirms **else**.

Pk2-3.
```
if (A<>0) and (sqr(A)<25) then A:=-A
else A:=1;
```

Pk2-4.
```
var sk:byte;
begin  sk:=random(10)+1;
case sk of
1: write('viens');
2: write('divi');
---------------
10:write('desmit');
end.
```

Pk2-5.
```
Jāizmanto operātori:
D:= sqrt(sqr(b)-4*a*c);
if D>0 then write('ir divas realas saknes');
if D=0 then write('ir viena reala sakne')
else  write('realu saknu nav')
```

Pk2-6.	X:= 3*sin(a*pi/180); **if** x>3 **then** write('y=',2*x-3/(x-1); **if** x<=3 **then** write('y=',2*x+5 **else** write('funkcija nav defineta')

Pk2-7. Jāizmanto operātors:

if ((a+b)>c) and ((a+c)>b) and ((c+b)>a) **then**
write('var uzkonstruet trissturi').

Pk3-1. a) **6 8 10 12 14 16**

b) **1; 99 2; 98 3; 97 4; 96 5; 95**

c) **** 6** 7** 8** 9** 10**

d) **13 14 15 16 17 18 19 20 21**

Pk3-2. a) **for** i:=1 **to** 5 **do writeln**(i);

b) Summa:=0;

```
for i:=1 to 10 do     begin
                            read(Skaitlis);
                            Summa:=Summa+Skaitlis;
                      end;
writeln (Summa:15);
```

Pk3-3. **for** i:=3 **to** 10 **do writeln**(13-i:13-i);

Pk3-4.

```
Summa:=0;
for i:=4 downto 1 do      begin
            writeln('*':15-i);
            Summa:=Summa+(5-i);
            writeln(Summa);
                      end;
```

Pk3-5. Var izmantot operātorus: sum:=0;

for i:= a **to** b **do** sum:=sum+sqr(i);

write(sum);

Pk3-6. Var izmantot operātorus: r:=1;

for i:= a **to** b **do** r:=r*i;

write(r);

Svarīgi izsekot tam, lai reizinājums r nepārsniegtu tā definīcijas apgabalu.

Pk3-7. Var izmantot operātorus: b:=0;

for i:= 1 **to** 100 **do begin** a:= random(n);

if a>b **then** b:=a; end;

write(b);

Pk3-8. Var izmantot operātorus:

```
for i:= 1 to 100 do begin a:= random(n);
if (a<=n/3)  then n1:=n1+1;
if (a>n/3) and  (a<=2*n/3)  then  n2:=n2+1
else  n3:=n3+1 end;
write('n1=',n1, 'n2=',n2, 'n3=',n3)
```

Pk3-9. Var izmantot operātorus: s:=0;

```
for i:= 1 to n do begin if (i mod 2=0)  then
 s:=-+1/i;
else  s:=s+1/i end;
write('n1=',n1, 'n2=',n2, 'n3=',n3);
```

Pk3-14. Var izmantot operātorus: read(N);

```
for i:= 1 to G do N:=N+N*P/100;
 write(' pec G gadiem summa summa bus',N);
```

Pk3-15. Var izmantot operātorus: s1:=1; s2:=1; i:=2;

```
repeat
s1:=s1+1;   s:= abs(s1-s2); i;=i+1;
s2:=s1;
until s>p;
```

Pk3-16. Var izmantot operātorus: p:=5; C:=200; n:=0;

```
while p<C do
begin  C:=C-p;  n:=n+1;  p:=p*1.2;  end;  write(n);
```

Pk3-17.

```
program Pitagors;
var a,b,c,cx: integer;
begin
      for a:=1 to 20 do
      for b:=a to 20 do
begin
      cx:=sqr(a)+sqr(b);
      c:=1;
while sqr(c)<=cx do
begin
      if sqr(c)=cx then
       write(a,b,c);
       c:= c+1;
```

{b mainās no a nevis no 1, lai izslēgtu pāru atkārtošanos}

{pārbauda nosacījumu visiem skaitļiem no 1 līdz pareizajam}

{pareizā skaitļa gadījumā izdrukā visus trīs skaitļus}

{pretējā gadījumā palielina c par 1 un

	end; **end** **end**.	atgriežas pie nosacījuma pārbaudes} Iegūstam rezultātu: 3 4 5 ;5 12 13; 6 8 10; 8 15 17; 9 12 15; 12 16 20; 15 20 25.

Pk4-1. a) masīva A elementu skaits ir 30, masīva B elementu skaits ir 11,

Masīva C elementu skaits ir 15.

b) **readln(**A[1]); **readln**(A[30]); **readln**(B[-5]); **readln**(B[5]);

readln(C[11]); **readln**(C[25]).

Pk4-2. KONTR:**array**[1..35]**of byte**; AUTO: **array**[1..20]**of real;**

ATBILDE:**array**[1..50]**of char.**

Pk4-3. a) Nauda[1] = 76,84 Nauda[2] = 43,27 Nauda[3] = 17,52

Nauda[1] = 17,52 Nauda[2] = 43,27 Nauda[3] = 25,31

Pk4-4. Varam izmantot operātorus**: for** i:=1 **to** 4 **do** s:= s+M[i];

for j:=5 **to** 7 **do** r:=r*A[j];

Pk4-5. Varam izmantot operātorus**: for** i:= 1 **to** n **do begin**

A[i]:=random(n); B[i]:=A[i]; C[i]:=A[i] + B[i]:end;

Pk4-6. **var** A:**array**[1..n] **of char**; B: **array**[1..n] **of char**;

begin for i:= 1 **to** n **do begin** read (A[i]); j:=n+1-i;

B[j]:=A[i];**end; for** i:= 1 **to** n **do** write(A[i]);

for i:= 1 **to** n **do** write(B[i]);**end.**

Pk5-2. **7 5**

a b

b *

a *

Pk5-4. Summa iegūstama ar **for** operātoru ar parametru vērtībām a=1 un b=5 un izmantojot funkciju skaitļa faktoriāla aprēķināšanai.

Pk5-5. Līdzīgi kā iepriekšējā uzdevumā ar operātora for palīdzību tiek aprēķināta summa, kuras saskaitāmie ir funkcijas $1/a^n$ vērtības skaitļiem n no 1 līdz izvēlētajai vērtībai.

Pk5-6. Katra no summām noformējama funkcijas veidā. Tabulas izveidošanai ērti izmantot operātoru **for**.

Pk5-7. Ieteicams izmantot pakāpes funkciju x^n. Funkcija y tiek noteikta pie nosacījumiem x=1, x=3 u.t.t.

Pk5-11 Izmantojot programmu **saknes,** ievadām intervāla, kurā gribam noteikt saknes, galapunktus a un b un uzrādām dalījumu skaitu intervālā n. Programma pieprasa lietotālam nepieciešamo rezultātu precizitāti, kuru nosaka parametrs p. Tā, piem., ievadot a=-6, b=6, n=100 un p=0,01, monitora ekrānā iegūstam: x = - 3.06 x = - 0.06 x = 0.96 x = 2.94 x = 4.98 un paziņojumu ‘’intervals a – b parbaudits!’’.

Ievadot a = -6, b = 6, n =1000 un p = 0,001, monitora ekrānā iegūstam: x = - 3.00 x = 0.00 x = 1.00 x = 3.00 x = 5.00 un paziņojumu ‘’intervals a – b parbaudits!’’

Pielikums 2. Speciālas konstantes draivera tipu norādei.

Detect =0; {automātiskais tipa meklēšanas režīms}

CGA =1;

MCGA =2;

EGA =3;

EGA64 =4;

EGAMono =5;

IBM8514 =6;

HercMono =7;

ATT400 =8;

VGA =9;

PC3270 =10;

Lielākā daļa adapteru var strādāt dažādos režīmos. Lai norādītu adapterim nepieciešamo darba režīmu, izmanto mainīgo <Tips>, kura vērtības var būt konstantes:

{Adapteris CGA:}

CGAC0 =0;

CGAC1 =1;

CGAC2 =2;

CGAC3 =3;

CGAHi =4;

{Adapteris MCGA:}

MCGA0 =0;

MCGA1 =1;

MCGA2 =2;

MCGA3 =3;

MCGAMed =4;

MCGAHi =5; {640x480 pikseļi}

{Adapteris EGA:}

EGALo =0; {640x200 pikseļi, 16 krāsas}

EGAHi =1; {640x350 pikseļi, 16 krāsas}

EGAMonoHi =3; {640x350 pikseļi, 2 krāsas} {Adapteris HGC un HGC+:}

HercMonoHi =0; {720x348 pikseļi}

{Adapteris ATT400:}

ATT400C0 =0; {analoģisks režīmam CGAC0}

ATT400C1 =1; {analoģisks režīmam CGAC1}

ATT400C2 =2; {analoģisks režīmam CGAC2}

ATT400C3 =3; {analoģisks režīmam CGAC3}

ATT400CMed=4; {analoģisks režīmam CGACHi}

ATT400CHi =5; {640x400 pikseļi, 2 krāsas}

{Adapteris VGA:}

VGALo =0; {640x200 pikseļi}

VGAMed =1; {640x350 pikseļi}

VGAHi =2; {640x480 pikseļi}

PC3270Hi =0; {analoģisks HercMonoHi}

{Adapteris IBM8514}

IBM8514Lo =0; {640x480 pikseļi, 256 krāsas}

IBM8514Hi =1; {1024x

Pielikums 3. Darbs ar programmu

Palaižot Turbo Pascal 7.0 (fails turbo, exe), atveras ekrāns, kura augšējā daļā izvietotā komandu rinda.

Šai rindā ietilpst

File (darbs ar failiem), kurā iekļautas komandas: New, Open, Save, Save as, Save all, Change dir, Printer setup, Print, Dos shell, Exit.

Exit (teksta redaktors) – iekļautas komanda: Undo, Redo, Cut, Paste, Clear, Show clipboard

Search (teksta meklēšana un aizstāvēšana) – komandas: Find Replace, Search again, Go to line number

Run (programmas izpilde) – **Run, Step Over** (izpildot programmu pa soļiem)**, Trace into** (izpildīt pa operatoriem), **Go to cursor** (izpildīt līdz rindai, kurā atrodas kursors), **Program reset** (pārtraukt skaņošanu, atbrīvot atmiņu, aizvērt visus failus, kuri tika izmantoti programmā), **Parametrs** (uzdot komandu rindas argumentus).

Compile (programmas kompilācija) un tās komandas **Compile, Make, Build, Information.**

Debug (programmas skaņošana) – **Breakpoints, Call stack, Watch, Output** (rezultātu logs), **User screen**, Evaluate/modify

Tools (komandu izpilde neizejot no Pascal vides)

Options (kompilatora parametru uzstādījums)

Windows (operācijas ar logiem)

Help (informācijas iegūšana).

Programmas ievade un korekcija. Ekrāna darba zonā (pamatlaukumā) tiek rakstīta programma. Svarīgi ņemt vērā, ka vienā ekrāna logā var rakstīt **tikai** vienu programmu. Kopumā ir 20 logi. Tie aizveras ar peles klikšķi uz izslēgšanas pogas ekrāna augšējā kreisā stūrī (maza - zaļa).

Pulsējošais kursors darba zonā norāda ekrāna vietu, kurā tiks ievadīts programmas teksts. Lai dzēstu kļūdaini ievadītu simbolu, kursors jānovieto zem šī simbola un jānospiež taustiņš **Delete** vai arī kursors jānovieto pa labi no attiecīgā simbola un jānospiež taustiņš **Back Space**. Uz nākamo rindu kursors tiek pārcelts nospiežot **Enter**.

Saglabāšana uz diska. Uzrakstīto programmu pirms tās izpildīšanas ieteicams **saglabāt** uz diska. Šai nolūkā tiek atvērta pozicija **File**, aktivizēta komanda Save vai Save as, ievadīts programmas nosaukums un nospiests **Enter**. Tā rezultātā fails tiks saglabāts kārtējā kataloga vinčesterā. Gadījumos, ja fails jāsaglabā kādā citā katalogā, pirms ieraksta, izmantojot komandu **Change dir** (no File) jāsameklē nepieciešamais katalogs.

Programmas izpilde. Šai nolūkā tiek aktivizēta komanda **Run**. Gadījumos, kas programmā pielaistas sintaktiskas kļūdas, Turbo Pascal translators norādīs uz to atrašanās vietu, un dos atbilstošu paziņojumu (Error...). pēc tam, kad kļūda ir novērsta un programma saglabāta uz diska, to var palaist atkārtoti.

Apskatīt programmas darba rezultātus varam ieslēdzot komandu **User screen** (no **Debug**). Nospiežot jebkuru klaviatūras taustiņu, atgriežamies pie programmas teksta. Varam izmantot arī komandu **Output** (no **Debug**), kura atvērs risinājuma rezultātu izvades logu. Lai atstātu šo logu atvērtu visā programmas izpildes laikā jārealizē komandu **Cascade** (no Window).

Beidzot darbu Pascal vidē jānospiež komandpoga **Exit** (no **File**).

Programmas skaņošana

Programmas skaņošanas procesā tiek atklātas un novērstas kļūdas, kuras ieviesušās visos programmas sagatavošanas etapos. Pēc sava rakstura kļūdas iedalāmas trīs grupās:

sintaktiskās kļūdas – tās kuras rodas Pascal valodas sintakses likuma neievērošanas dēļ. Šai grupā tipiskas kļūdas ir:

- punkturācijas zīmju trūkums;
- apraksta daļā nedefinētu konstanšu, mainīgo vai masīvu izmantošana programmas operatora daļā;
- nepareizu datu piekārtošana, u.c.

šāda tipa kļūdas tiek atrastas kompilācijas procesā.

semantiskās kļūdas – saistītas ar Pascal valodas semantisko likumu neievērošanu (dalīšana ar nulli, mēģinājums izvilkt sakni no negatīva skaitļa u.c.). Šī tipa kļūdas tiek atklātas programmas izpildes gaitā (etapā). Tā rezultātā programmas izpilde tiek pārtraukta, kursors nostājas kļūdas vietā un uz ekrāna tiek izvadīts paziņojums:

Run time error <Nr.> at < x: y>,

kur Nr. – kļūdas numurs, x:y – kļūdas adrese.

algoritmiskās kļūdas – rodas nepareizu algoritmisku konstrukciju veidošanas

rezultātā. Šī veida kļūdas ne vienmēr ir vienkārši atrodamas.

Pascalā nav automātiska loģisko kļūdu meklētāja. Šādu kļūdu atklāšanai tiek rekomendēts veikt atsevišķu apakšprogrammas skaņošanu sākot ar zemāko hierarhijas līmeni. Programmas darba rezultātus obligāti jāpārbauda ar testa uzdevumiem, kuru rezultāti ir zināmi.

Turbo Pascal integrētajā rīku aprīkojumā atrodams iebūvēts skaņotājs (**Debugger**), ar kura palīdzību var izsekot programmu izpildei. Skaņotājs dod iespēju:

- veikt programmas skaņošanu un trasēšanu pa soļiem;
- izpildīt programmu līdz noteiktai vietai;
- realizēt programmas pārstartēšanu;
- caurskatīt un modificēt mainīgos.

Skaņošana un trasēšana pa soļiem. Tiek aktivizēta komanda **Step Over** (no **Run**). Lai veiktu programmas trasēšanu, jāizmanto komanda **Trace Into** (no **Run**). Atšķirība starp abiem šiem procesiem (skaņošanu un trasēšanu) ir tā, ka veicot programmas skaņošanu pa soļiem nav iespējams izsekot to operatoru izpildei, kuri iekļauti procedūrās un funkcijās. Toties veicot trasēšanu, tāda iespēja ir.

Programmas izpilde līdz noteiktai vietai. Gadījumos, kad nepieciešams noskaņot tikai daļu no programmas, var izmantot iespēju apturēt programmas izpildi noteiktā vietā. Lai to veiktu ir nepieciešams:

novietot kursoru uz rindas, līdz kurai jāizpilda programma;

aktivizēt komandu **Go to cursor** (no **Run**). Fiksēt programmas izpildes pieturpunktu (**Breakpoint**) iespējams vēl vairākos veidos: ar komandu **Toggle Breakpoint** (no Local menu), Breakpoint (no **Debug**), aktivizējot komandu **Add breakpoint** (no Debug) un nospiežot **Enter.**

Lai likvidētu apstāšanās punktu, pietiek nospiest klaviatūras taustiņu kombināciju **Ctrl+F8**.

Programmas pārstartēšana. Lai pārstartētu programmu pirms tās darbības beigām, pietiek aktivizēt komandu **Program reset** (no **Run**) vai nospiest kombināciju **Ctrl+F2**. Tā rezultātā programma tiks izpildīta no sākuma.

Mainīgo caurskate un modifikācija. Nepieciešams izpildīt programmu līdz konkrētai vietai, aktivizējot komandu **Watch** (no **Debug**). Tā rezultātā atvērsies logs Watches. Nospiežot **Ctrl+F7** vai taustiņu **Insert**, atvērsies logs **Add Watch.** Pēc tam nepieciešams ievadīt mainīgā nosaukumu un nospiest Enter. Logā Watches parādīsies mainīgā nosaukums un tekošā vērtība. Mainīgā dzēšana logā Watches notiek nospiežot taustiņu **Del** vai **Ctrl+Y**.

Gadījumā, ja vēlamies nomainīt mainīgā vērtību, nepieciešams izpildīt programmu līdz pieturpunktam, aktivizēt komandu **Eveluate /modify** (no **Debug**). Atvērsies dialoga logs **Evaluate and Modify**. Laukā Expression jāievada mainīgā nosaukums, laukā **New Value** mainīgā jaunā vērtība un ar peles kreiso taustiņu nospiest taustiņu Modify. Tā rezultātā laukā **Result** parādīsies mainīgā jaunā vērtība. Pēc tam nospiežam taustiņu **Evaluate** un taustiņu **Cancel**. Izdarīto manipulāciju rezultātā programmas izpildi varam turpināt jau ar jauno mainīgā vērtību.

Pielikums 4. Paziņojumu kodi par kļūdām Turbo Pascal valodas programmās

Iepazīsimies ar tipiskākajiem paziņosumiem par kļūdām , ar kurām nākas sastapties programmu kompilācijas un izpildes gaitā.

Paziņojumi par kompilācijas kļūdām.

1. **Out of memory** (iziets ārpus atmiņas robežām). Kļūda parādās gadījumā, ja kompilātors ir izmantojis visu tam atvēlēto atmiņu. To var novērst izmainot apgabala *Destination* no *Memory* vērtību.
2. **Identifier expected** (Tiek gaidīts identifikātors)**.** Šai vietā jāatrodas identifikātoram.
3. **Uncnown identifier** (Nepazīstams identifikātors)**.**
4. **Duplicate identifier** (identifikātora atkārtojums).
5. **Syntax error** (Sintaktiska kļūda). Tekstā atrasts nepieļaujams simbols. Iespējams, ka trūkst simbols - iekava.
8. **String constant exceeds line** (Rindiņas konstante pārsniedz rindiņas garumu).
21. **Error in type** (kļūda tipa noteikšanā0. Tipa noteikšana nevar sākties ar šo simbolu.
26. **Type mismatch** (Tipu neatbilstība). Paziņojuma cēloņi var būt: mainīgā un izteiksmes tipu neatbilstība piešķires operātorā; faktisko un formālo parametru tipu neatbilstība vēršoties pie procedūrām vai funkcijām; izteikmju operandu tipu neatbilstība.
30. **Integer constant expected** (Tiek prasīta vesela tipa konstante)
31. **BEGIN expected** (Nepieciešams operators BEGIN)

Liktenīgās kļūdas.

200. **Division to zero** (Dalīšana ar nulli).

205 **Floating point overflow** (Pārpildīšana veicot operācijas ar peldošu punktu).

207. **Invalid floating point operation** (Neiespējama darbība ar reālu skaitli). Kļūdas iemesli: reālais skaitlis, kurš tiek nodots funkcijai *Trunc* vai *Round* nevar tikt pārveidots par veselu tipa *Longint* pieļaujamā intervāla robežās; funkcijas sqrt arguments ir negatīvs; funkcijas ln arguments ir negatīvs vai nulle.

215. **Arithmetic overflow error** (Kļūda veicot matemātisku operāciju). Šāda kļūda rodas, piemēram, ja izteiksmes vērtība pārsniedz definēto vērtību diapozonu.

Literatūra.

- G SPALIS. Turbo Pascal for Windows ikvienam. Rīga, Datorzinību Centrs. 1998., 126 lpp.
- L. KUZMINA, J. KUZMINS. Pascal valoda skolēniem un skolotājiem. Lielvārds. 2001, 96 lpp.
- Ю.А. АЛЯЕВ, О.А. КОЗЛОВ. Алгоритмизация и языки программирования .Москва, Финансы и статистика, 2002, 319 стр.

- Г.РАПАКОВ, С. РЖЕУЦКАЯ. Turbo Pascal для студентов и школьников. БХВ – Петербург, 2002, 349 стр.
- С. А. НЕМНЮГИН. Turbo Pascal. Питер, 2003, 491 стр.