1 Algoritmizácia úloh

Pod pojmom algoritmizácia riešenia problému (úlohy) si môžeme predstaviť definovanie pravidiel (činností) a ich postupnosti, ktoré treba v čase a priestore vykonať pre dosiahnutie požadovaných výsledkov riešenia úlohy. Pod pojmom algoritmizácia si teda môžeme rovnako predstaviť hľadanie a zostavovanie pravidiel na riešenie zadaného matematického problému, ako aj navrhovanie a zostavovanie technologických postupov, liniek pre výrobu ľubovoľného produktu (výrobku).

Algoritmus (konkrétneho problému) je potom súbor pravidiel, ktoré je potrebné aplikovať na vstupné údaje (vstupy), spĺňajúce vstupné podmienky, aby sa dosiahli výstupy (výsledky), spĺňajúce požadované vlastnosti.

Pod vstupmi si môžeme predstaviť nielen vstupujúce číselné údaje, ale aj vstupné suroviny do technológie, pre ktoré sa požaduje určitá kvalita (vstupné podmienky), aby bolo možné získať výrobok (výstup) požadovanej kvality a funkčnosti (podmienky pre výstup).

Ak chceme prehlásiť, že zostavený súbor pravidiel je algoritmom (konkrétneho problému), musí spĺňať nasledovné vlastnosti:

1. **konečnosť** – algoritmus pozostáva z konečného ' počtu jednotlivých krokov (pravidiel), t.j. po určitom počte dokázateľne končí.
2. **rezultatívnosť** – algoritmus dokázateľne vedie výpočtový proces od vstupných údajov k výsledku – riešeniu problému.
3. **determinovanosť** – po realizácii každého kroku algoritmu sa dá rozhodnúť, či sa výpočtový proces už skončil a ak nie, tak ktorý krok sa má vykonať ako nasledujúci.
4. **hromadnosť** – algoritmus zabezpečuje riešenie všetkých úloh toho istého typu.

Algoritmický problém možno schematicky znázorniť: $\{I\} \rightarrow \{A\} \rightarrow \{O\}$
kde {I} sú vstupné údaje a vstupné podmienky, {O} sú výstupné údaje a podmienky a {A} je hľadaný algoritmus.

Riešenie algoritmického problému spočíva v hľadaní algoritmu s vyššie uvedenými vlastnosťami.

Tak, ako na výrobu jedného výrobku môže existovať viac technologických postupov, ktoré vyžadujú rôzne materiálové, časové, energetické a iné náklady, potom aj na riešenie konkrétneho problému je možné vytvoriť viacero algoritmov. Ich väčšia, či menšia **efektívnosť** sa potom zvyčajne porovnáva z hľadiska časovej resp. pamäťovej náročnosti vzhľadom na nejaký konkrétny procesor. Pod pojmom **procesor** máme na mysli "realizátora" algoritmu (stroj alebo človeka).

Časovou výpočtovou zložitosťou algoritmu nazývame funkciu, ktorá vyjadruje závislosť skutočného počtu operácii (aritmetických, vyhodnotení podmienok a pod.) potrebných na realizáciu algoritmu, od veľkosti a množstva vstupných údajov.

Pamäťovou výpočtovou zložitosťou algoritmu nazývame funkciu, vyjadrujúcu závislosť počtu údajov, ktoré si pri realizácii algoritmu treba pamätať, od veľkosti a množstva vstupných údajov.

Na zápis algoritmov sa bežne používajú dva spôsoby:

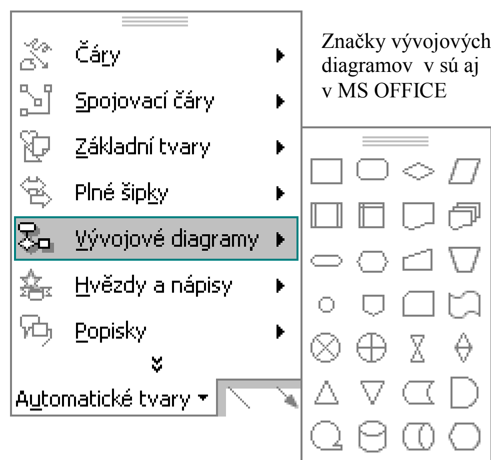
1. **slovný** – pomocou zvoleného algoritmického jazyka
2. **grafický** – pomocou vývojových diagramov.

Vývojové diagramy sú jedným z najstarších a najznámejších spôsobov zápisu algoritmov. Ich výhodou je názornejšie a presnejšie zobrazenie štruktúry procesov, ako aj vyjadrenie poradia vykonávania jednotlivých činností pri realizácii algoritmu. Sú preto vhodným prostriedkom pre zachytenie toku **riadenia** v algoritme. Na

druhej strane je však ich nevýhodou nepomer medzi veľkosťou kresby a množstvom informácie, ktorú poskytujú, čo sa okamžite prejaví pri zložitejších algoritmoch.

Vzhľadom na to, že sa v ďalšom sústredíme predovšetkým na slovný popis algoritmov, uvedieme len niektoré základné prvky vývojových diagramov, ktoré budú neskôr použité ako pomôcka na objasnenie niektorých štruktúr algoritmického jazyka. V prípade záujmu si čitateľ môže pozrieť napr. **ČSN 36 9001, ČSN 36 9030.**

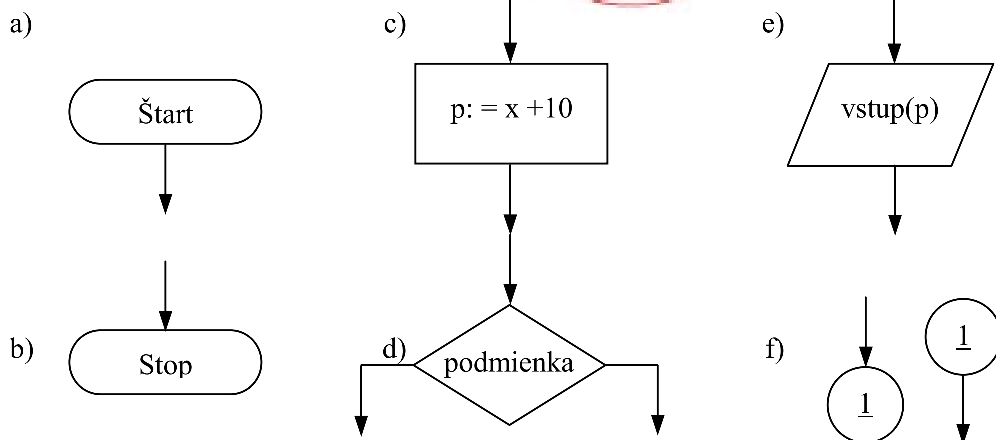
Vývojový diagram je obrazec pozostávajúci z blokov, do ktorých sa zapisujú jednotlivé výkonné a rozhodovacie kroky algoritmu. Bloky sú spájané orientovanými spojnicami. V prípade, že spojnice nie sú orientované, predpokladá sa postup zhora nadol, resp. zľava doprava.



Značky vývojových diagramov v sú aj v MS OFFICE

Každý vývojový diagram začína práve jedným štartovacím blokom (Obr. A.1a)), z ktorého vedie sled spojníc a blokov ku koncovému bloku (Obr. A.1b)), v ktorom sa realizácia algoritmu končí. Medzi týmito základnými blokmi sa môžu nachádzať operačné bloky (Obr. A.1.c)), obsahujúce operácie, ktorých výsledkom je transformácia informácie (napr. zmena hodnoty, umiestnenia, apod.) a rozhodovacie bloky (Obr. A.1d)), ktoré obsahujú podmienku určujúcu ďalší smer postupu. Vstup alebo výstup údajov sa zakresľuje pomocou vstupno/výstupných blokov (Obr. A.1e)).

V prípade nedostatku kresliaceho priestoru možno na prenesenie toku riadenia použiť spojky (Obr. A.1f)) a pokračovať v kreslení na inom vhodnom mieste. Spojky, ktoré predstavujú pokračovanie tej istej spojnice, sú označené tými istými znakmi (číslom, písmenom, textom apod.).



Obr. A.1a–f) Základné prvky vývojových diagramov

Na slovný popis algoritmov sa bude v ďalšom používať algoritmický metajazyk. Predpokladá sa, že hypotetický procesor, ktorý bude algoritmy realizovať "pozná" základné algebraické operátory ako +, –, *, /, ^ (mocnina), vie vyhodnocovať výrokové formy vytvorené pomocou logických operátorov "a súčasne" (∩–konjunkcia), "alebo" (∪–disjunkcia), "negácia" (¬ – negácia), (⊕ – neekvivalencia) (≡ – ekvivalencia) a výrazy vytvorené pomocou relačných operátorov =, <, >, ≤ , ≥ ,< >, ≠ .

Základnými objektmi algoritmického jazyka, s ktorými procesor dokáže pracovať, sú **konštanty** a **premenné**. Premenné sa chápu ako objekty, ktoré počas realizácie algoritmu môžu nadobúdať isté hodnoty. Tieto hodnoty patria do množiny, ktorá sa nazýva typ **údajov** a jej prvky **hodnoty typu**.

Počet hodnôt typu sa nazýva kardinalita – **kardinálne číslo** typu. Typ, ktorý má konečne veľa hodnôt sa nazýva konečný typ (ordinálny), v opačnom prípade sa jedná o nekonečný typ. Poznamenávame, že v algoritmizácii ako aj pri realizácii procesorom sa aj niektoré nekonečné typy transformujú na spočitateľné typy.

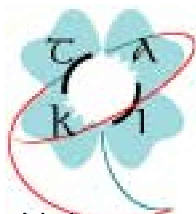
S typom údajov úzko súvisia operácie vykonávané nad jeho hodnotami, ktoré sú špecifické pre každý typ. Ich výsledkom je výraz fixne dopredu určeného typu.

Napr. M **div** N má výsledok celočíselný
M/N má výsledok reálny (desatinné číslo).

Typy údajov sa delia na základné a štruktúrované.
Základné typy údajov obsahujú elementárne, nedeliteľné objekty napr.: čísla, logické konštanty, znaky.

Patria k nim typy:
- celočíselný
- vymenovaný
- reálny
- logický {pravda, nepravda}
- znakový.

Štruktúrované typy sa vytvárajú z už existujúcich typov vyčlenením podtypu alebo kompozíciou – štruktúrovaním, napr.:
- interval
- reťazec
- pole
- záznam atď. (ako štruktúrované typy).



V rôznych časových okamihoch môže mať jedna premenná rôzne hodnoty. Konštanty naopak slúžia na označenie istého objektu, ktorý sa počas celého procesu (algoritmu) nemení (napr. Ludolfovo číslo π v goniometrických výpočtoch). Konštanty a premenné sa označujú písmenami alebo znakovými reťazcami, pozostávajúcimi z písmen a číslic (prvé musí byť vždy písmeno) napr. A, sucin, B5, atď. Diakritika sa obvykle nepoužíva.

Aritmetické výrazy sú výrazy vytvorené z konštánt, premenných, aritmetických operátorov, funkcií nadobúdajúcich číselné hodnoty a okrúhlych zátvoriek. Výrazmi sú napr. 10, sin(x+π/2), |x|+B, atď.

Štandardné aritmetické funkcie v metajazyku sú goniometrické funkcie sin (zápis: sin(x)), cos, tg, cotg, cyklometrické funkcie arcsin, arccos, arctg, arccotg, exponenciálna funkcia so základom e (e^x) , logaritmická funkcia (ln(x)), odmocnina (√x) a absolútna hodnota (|x|).

Algoritmické operátory, ktoré počas realizácie algoritmu manipulujú s údajmi alebo riadia tok algoritmu, sa nazývajú príkazy. Na ich zápis sa používajú tzv. **rezervované slová**, ktoré budú písané tučným typom písma, podčiarknuté napr. **pre**, ***pre**. Nesmú sa používať na označenie iných objektov algoritmického jazyka (konštánt, premenných atď.).

Príkazy algoritmického metajazyka sa rozdeľujú na základné a štruktúrované.

Základné príkazy:

1. prázdny príkaz 2. priraďovací príkaz 3. príkaz vstupu 4. príkaz výstupu

Štruktúrované príkazy

1. sekvencia
2. podmienený príkaz
3. príkazy cyklu
   - cyklus so známym počtom krokov (opakovaní)
   - cyklus s neznámym počtom krokov (opakovaní)

Najjednoduchším príkazom je prázdny príkaz (p–príkaz), ktorý nemení žiadne hodnoty premenných. Napr. majme algoritmický problém:

Dané sú celé čísla A, B : A<B **A.1**

{A}

{ A ≤ B }

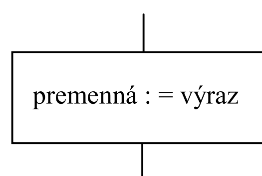
Je zrejmé, že miesto **{A}** môžeme dosadiť p–príkaz, pretože čísla spĺňajúce vstupnú podmienku logicky spĺňajú aj výstupnú podmienku.

Ďalším riešením daného problému môže byť zvýšenie hodnoty A o jedna, to znamená zmenu hodnoty premennej A. Na realizáciu slúži priraďovací príkaz, ktorý má tvar:

premenná := výraz

resp.

premenná ← výraz



kde výraz predstavuje aritmetický výraz. Priraďovací príkaz sa realizuje tak, že sa najskôr vypočíta hodnota výrazu na pravej strane a výsledok sa priradí premennej na ľavej strane príkazu ako jej nová hodnota. Algoritmus (A.1) by potom vyzeral nasledovne:

{ A < B }

A := A+1 **A.2**

{ A ≤ B }

V prípade, že sa tá istá premenná nachádza na oboch stranách priraďovacieho príkazu, najskôr sa vyhodnotí výraz s pôvodnou hodnotou premennej a až potom sa jeho hodnota priradí premennej na ľavej strane. Príkaz

A:=A+1

je teda v skutočnosti

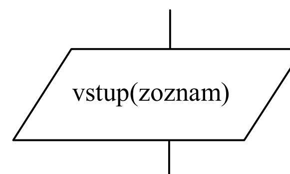
$A_{nové} := A_{staré} + 1$.

Vstupné premenné, ktoré sa vyskytujú vo vstupnej podmienke, musia mať pred ich prvým spracovaním priradené tzv. začiatočné hodnoty. Toto priradenie sa vykonáva pomocou **príkazu vstupu**, ktorý má tvar:

vstup(premenná)

resp.

vstup(premenná$_1$, ... , premenná $_n$)



Prípustný je aj zápis:

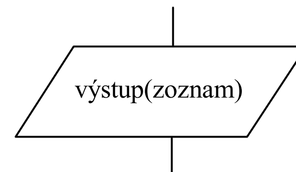
čítaj(premenná)
resp.
čítaj(premenná$_1$,..., premenná $_n$).

Postupnosť premenná$_1$,...,premenná $_n$ sa nazýva **zoznam**. Pretože je použitý v príkaze vstupu, dostáva prívlastok vstupný (zoznam). Pomocou príkazu vstupu sa premenným uvedeným vo vstupnom zozname priraďujú hodnoty zo vstupu, napr.: zadávaním z klávesnice alebo iného vstupného zariadenia.

Z dvoch posledne uvedených príkazov vyplýva, že **premenné môžu nadobúdať hodnoty pomocou príkazu vstupu alebo priraďovacieho príkazu**. Zároveň treba poznamenať, že **sú to jediné príkazy, slúžiace na túto činnosť**. Rozdiel medzi týmito dvoma príkazmi je v tom, že zatiaľ čo sa v priraďovacom príkaze najskôr vypočíta hodnota výrazu, ktorá sa potom uloží do premennej, v príkaze vstupu je hodnota, ktorú chceme uložiť do premennej známa až pri realizácii algoritmu (vstupuje z okolia procesora).

Výsledky realizácie algoritmu sa zapisujú pomocou **príkazu výstupu** v tvare:

výstup(premenná) resp.
výstup(výraz$_1$, ..., výraz $_n$)
alebo
píš(premenná) resp.
píš(výraz$_1$, ..., výraz $_n$).



Príkaz výstupu sa vykonáva tak, že sa najskôr vypočíta hodnota výrazu v zátvorkách a táto sa vypíše na výstupné zariadenie (na obrazovku, tlačiareň). Postupnosť výraz$_1$, ..., výraz $_n$ sa nazýva **výstupný zoznam**. V príkaze výstupu sa pod pojmom výraz rozumie nielen aritmetický výraz, ale aj text, t.j. postupnosť znakov medzi dvoma úvodzovkami (napr. "Výška = "). Ak má napr. premenná A v istom okamžiku realizácie algoritmu hodnotu 10 a práve nasleduje príkaz výstupu

výstup("A = ", A)

tak ten realizuje výpis textového reťazca medzi úvodzovkami (výraz$_1$) a výpis hodnoty A (výraz$_2$) nasledovne:

A = 10 .

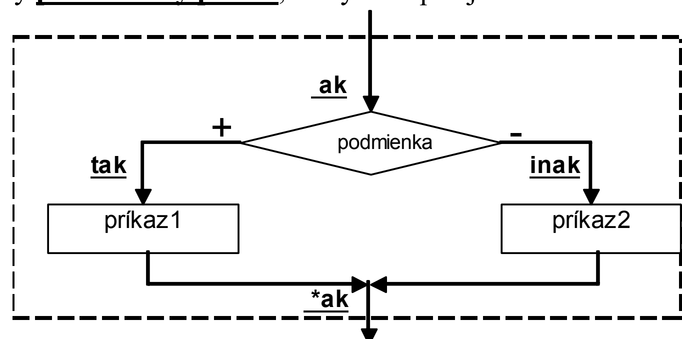
Uvažujme teraz algoritmický problém:
{ Dané sú celé čísla A, B, A≤ B }

{A} **A.3**

{ A<B }

Ak vstupné hodnoty A, B zároveň spĺňajú výstupnú podmienku, tak by stačilo použiť prázdny príkaz. Ak nie (t.j. platí vzťah A=B), treba napr. zmenšiť hodnotu A o jedna. Problém je teda možné rozložiť na dva disjunktné podproblémy (A=B alebo A<>B, z čoho vyplýva, že A<B) a ďalej pokračovať v závislosti od toho, ktoré tvrdenie je platné. V metajazyku je na to určený **podmienený príkaz**, ktorý sa zapisuje:

ak podmienka
tak príkaz_1
inak príkaz_2
***ak** ;



kde pod podmienkou je možné (aj v ďalších štruktúrovaných príkazoch) si predstaviť ľubovoľnú výrokovú formu, obsahujúcu konštanty, premenné, relačné symboly a logické operátory. Príkaz_1,

príkaz_2 predstavujú ľubovoľné základné alebo štruktúrované príkazy algoritmického jazyka.
Poznámka. Realizácia "vnútorných" príkazov (príkaz_1, príkaz_2) musí skončiť skôr ako "vonkajší" štruktúrovaný príkaz. Ide teda o vnorenie príkazov. V žiadnom prípade sa príkazy nesmú prekrývať.

Podmienený príkaz pracuje nasledovne:

vyhodnotí sa podmienka; ak platí, tak sa vykoná príkaz_1, ak podmienka neplatí, realizuje sa príkaz_2. Teda z dvoch príkazov, ktoré sú uvedené v podmienenom príkaze, sa vždy realizuje iba jeden podľa toho, či podmienka (výrok, tvrdenie) platí alebo neplatí.

Algoritmický problém A.3 potom môžeme riešiť algoritmom:

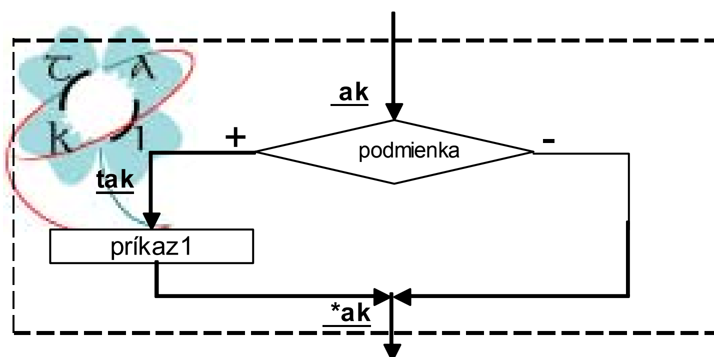
{ A ≤ B }
ak A = B **A.4**
tak A := A–1
inak p –príkaz
***ak ;**
{ A < B }

Pri praktickej tvorbe algoritmov sú dosť časté prípady, keď v prípade neplatnosti podmienky nerobíme nič (p--príkaz). Preto v takýchto situáciách je možné použiť aj tzv. **skrátený podmienený výraz** v tvare:

ak podmienka
tak príkaz_1
***ak;**

čo je ekvivalentné zápisu:

ak podmienka
tak príkaz_1
inak [p–príkaz]
***ak;**



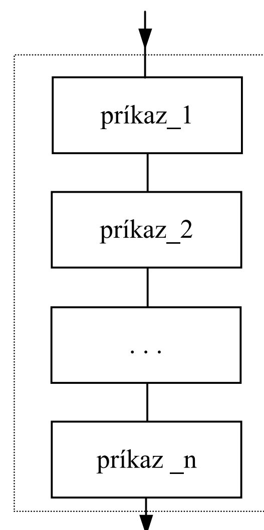
kde p–príkaz je možné, ale nie nutné písať.

Daný je ďalší algoritmický problém:
{ Dané sú celé čísla $A=a_0$, $B=b_0$ }
{A} **A.5**
{ $A=b_0$, $B=a_0$ }

čo je vlastne výmena dvoch hodnôt. Ak by sa premennej A priradila hodnota premennej B, tým by sa síce splnila prvá z výstupných podmienok, ale súčasne by sa nenávratne "stratila" pôvodná hodnota premennej A. Tú si teda na začiatku treba "odložiť" do nejakej pomocnej premennej napr. POM a nakoniec ju priradiť do premennej B. Je zrejmé, že na realizáciu tohoto algoritmu nevystačíme len s jedným príkazom, ale musíme použiť postupnosť troch príkazov, ktorú nazývame **sekvenciou**.

Sekvencia má všeobecný tvar:

príkaz_1;
príkaz_2;
. . . ;
príkaz _n

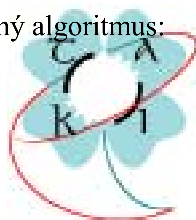


Poznámka. Za každým príkazom vo vnútri sekvencie treba dať bodko–čiarku ako oddeľovač jednotlivých príkazov.

Algoritmický problém (A.5) teda možno riešiť algoritmom

{ Dané sú celé čísla $A=a_0$, $B=b_0$ }
POM:=A:
A:=B:
B:=POM **A.6**
{ $A=b_0$, $B=a_0$ }

Iným riešením problému (A.5) je aj nasledovný algoritmus:



{ Dané sú celé čísla $A=a_0$, $B=b_0$ };
A:=A-B;
B:=A+B;
A:=B-A
{ $A=b_o$, $B=a_o$)

Presvedčíme sa o tom sledovaním resp. tzv. **trasovaním** výpočtového procesu, kedy si vytvoríme pomocnú tabuľku, kde budeme sledovať ako sa menia hodnoty premenných po realizácii jednotlivých príkazov.

{ $A=a_0$, $B=b_0$ }
A:=A–B;
B:=A+B;
A:=B–A
{ $A = b_o$, $B = a_o$ }

A	B	Výraz
a_0	b_0	
$a_0–b_0$		$a_0–b_0$
	a_0	$(a_0–b_0)+ b_0$
b_0		$a_0–(a_0–b_0)$
b_0	a_0	

Trasovanie algoritmu je užitočné najmä vtedy, ak algoritmus nedáva očakávané výsledky a je potrebné zistiť, kde sme pri jeho tvorbe algoritmu spravili chybu alebo aj vtedy, ak pri rozsiahlych výpočtoch potrebujeme dokázať správnosť postupu riešenia daného problému.

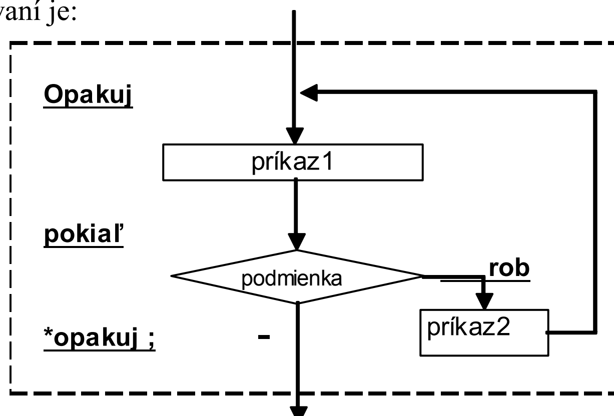
Uvažujme teraz nasledovný algoritmický problém:

{ Dané sú celé čísla A, B, $A \leq B$ }
{A} **A.8**
{ A=B }

ktorý sa pokúsime riešiť opakovaným zväčšovaním hodnoty premennej A o 1, až kým nedosiahne hodnotu premennej B. Toto zväčšovanie možno zrejme vykonávať len vtedy, pokiaľ platí vzťah A<B (inak by A prekročila hodnotu B). Proces, ktorý sa systematicky opakuje sa nazýva **cyklus**. Ak nevieme dopredu povedať, koľkokrát sa má istý proces opakovať, jedná sa o **cyklus s neznámym počtom opakovaní**.

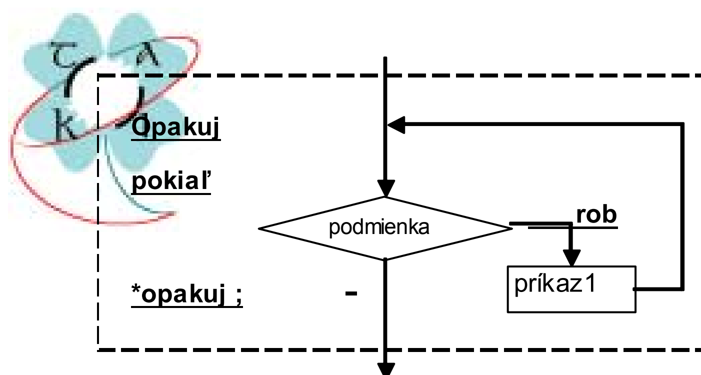
Tvar úplného cyklu s neznámym počtom opakovaní je:

opakuj príkaz_1
  pokiaľ podmienka
  rob príkaz_2
***opakuj ;**



Príkaz úplného cyklu pracuje nasledovne: vykoná sa príkaz_1 a testuje sa podmienka cyklu. Ak je podmienka platná, realizuje sa príkaz_2, po ňom opäť príkaz_1 a znova sa testuje podmienka cyklu. Táto činnosť sa opakuje dovtedy, kým po prvýkrát nebude splnená podmienka cyklu.
Redukciou úplného cyklu (ak chýba príkaz_1) dostávame tzv. **cyklus s podmienkou na začiatku**. Tento sa zapisuje:

opakuj [p–príkaz]
  pokiaľ podmienka
  rob príkaz1
***opakuj ;**



Príkaz sa vykoná nasledovne: najskôr sa vyhodnotí podmienka cyklu; ak platí, vykoná sa príkaz1, ktorý sa nazýva aj **telo cyklu**. Po vykonaní tela cyklu sa znovu testuje podmienka cyklu. Toto testovanie podmienky a vykonanie tela cyklu sa opakuje dovtedy, kým po prvý krát nebude platiť podmienka cyklu. V tomto prípade sa už telo cyklu nevykoná a realizácia celého príkazu cyklu končí. Počet realizácií tela cyklu sa nazýva **počtom krokov cyklu**. Pretože je podmienka až na začiatku cyklu, nemusí sa telo cyklu vykonať ani jedenkrát.

Riešenie problému (A.8) teda bude vyzerať:

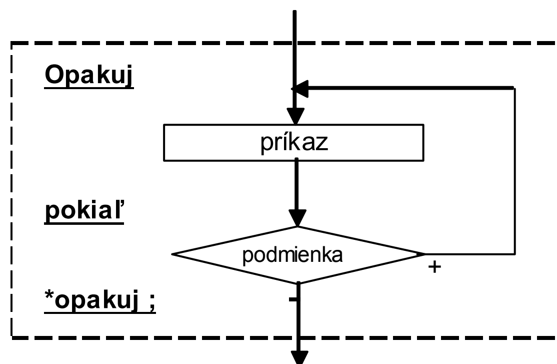
{ $A \leq B$}
  opakuj
    pokiaľ A < B
    rob A:=A+1 **A.9**
  ***opakuj ;**
{ A = B}

Vzhľadom na to, že sa jedná o celé čísla, cyklus logicky skončí práve vtedy, keď premenná A dosiahne hodnotu premennej B.

Čo sa stane, ak v (A.8) zmeníme vstupnú podmienku na A<B ? Zrejme prvé zväčšenie hodnoty A môžeme vykonať predtým, než testujeme podmienku cyklu a potom môžeme zisťovať, či ešte stá1e platí vzťah A<B. Ak tento ešte platí, vtedy sa vrátime k zvyšovaniu hodnoty A. To znamená, že sa bude opakovať telo cyklu

dovtedy, kým bude splnená podmienka cyklu. Takýto cyklus sa nazýva **cyklus s podmienkou na konci** a zapisuje sa v tvare:

opakuj príkaz
pokiaľ podmienka
[**rob** (p-príkaz)]
***opakuj ;**



Rozdiel medzi týmito dvoma cyklami je viditeľný aj z príslušných vývojových diagramov. Kým v cykle s podmienkou na konci sa telo cyklu vždy vykoná aspoň raz, v cykle s podmienkou na začiatku sa nemusí vykonať ani raz, ak podmienka cyklu nie je splnená hneď pri prvom testovaní.
Použitím cyklu s podmienkou na konci získame riešenie v tvare:

{ A<B }
opakuj A:=A+l
pokiaľ A<B **A.10**
***opakuj ;**
{ A=B }

Ostáva ešte spomenúť posledný typ cyklu, ktorým je **cyklus so známym počtom opakovaní**. Zapisuje sa nasledovne:

pre premenná := výraz$_1$ **po** výraz$_2$
rob príkaz
***pre ;**

kde premenná je tzv. **riadiaca premenná** cyklu (ordinálneho typu), výraz$_1$, výraz$_2$ sú výrazy nadobúdajúce hodnoty toho istého typu ako riadiaca premenná. Príkaz sa realizuje nasledovným spôsobom:
riadiacej premennej sa priradí hodnota výrazu$_1$ a vykoná sa telo cyklu. Potom sa premennej priradí hodnota nasledujúca (v príslušnom type) za hodnotou premennej a znova sa vykoná telo cyklu atď., až kým premenná nenadobudne hodnotu výrazu$_2$. Riadiaca premenná sa tiež môže vyskytnúť v niektorom príkaze tela cyklu, avšak jej hodnota by sa v tele cyklu nemala meniť (t.j. nemala by sa vyskytovať na ľavej strane priraďovacieho príkazu alebo v príkaze vstupu).

Pomocou už známych cyklov a za predpokladu, že premenná je ordinálneho typu, sa môže cyklus so známym počtom opakovaní prepísať s použitím funkcie succ (následník):

premenná:=výraz$_1$;
opakuj
pokiaľ premenná ≤ výraz$_2$
rob príkaz
premenná := succ (premenná)
***opakuj ;**

Napr. máme zostaviť algoritmus násobenia dvoch celých kladných čísel A, B pomocou sčítavania, t.j.

A * B = A + A + . . . + A

B-krát

{ Dané sú kladné čísla A, B }
{A} **A.11**
{ SÚČIN=A*B }

Problém (A.11) vyriešime pomocou cyklu so známym počtom opakovaní. Použijeme premennú SÚČIN, ktorá bude mať na začiatku hodnotu 0. K nej budeme postupne pripočítavať hodnotu premennej A, tento proces vykonáme B–krát. Dostaneme

{ Dané sú kladné čísla A, B }
SÚČIN := 0;
pre i:=1 **po** B **A.12**
rob SÚČIN:=SÚČIN+A
***pre ;**
{ SÚČIN=A*B }

Riadiaca premenná cyklu nemusí svoju hodnotu iba zvyšovať, ale môže ju aj znižovať (funkcia pred), čo sa zapisuje nasledovne:

pre premenná := výraz_1 **klesaním po** výraz_2
rob príkaz
***pre ;**

Pomocou už známych cyklov a za predpokladu, že premenná je ordinálneho typu, sa môže cyklus so známym počtom opakovaní prepísať s použitím funkcie pred (predchodca):

premenná := výraz$_1$;
opakuj
pokiaľ premenná ≥ výraz$_2$
rob príkaz
premenná := pred (premenná)
***opakuj ;**

Kompletný algoritmus v algoritmickom metajazyku pozostáva z názvu algoritmu, popisov premenných (deklarácií) a príkazov. Vo všeobecnosti má tvar:

Alg názov_algoritmu **;**
prem popis premenných **;**
začiatok
telo algoritmu
koniec .

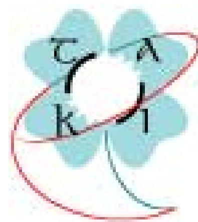
Názov algoritmu tvorí reťazec znakov, zostavený z písmen, číslic, (prípadne znaku "_" podčiarkovník). Popis premenných popisuje príslušnosť jednotlivých premenných vyskytujúcich sa v algoritme k určitým typom údajov. Príslušnosť premenných $p_1, p_2, ..., p_n$ k určitému typu údajov D, sa zapisuje v tvare

$p_1, p_2, . . . , p_n$ **:** D;

Napr.:

i, j ,POMOC	:	celočíselné **;**
SUMA	:	reálne **;**
NASIEL	:	logické **;**
ZNAK1, ZNAK2	:	znakové **;**
TABULKA	:	pole[1..100] reálnych {hodnôt}**;**

Telo algoritmu pozostáva z príkazov algoritmického jazyka oddelených bodkočiarkami. Okrem príkazov sa v tele algoritmu môžu vyskytovať **komentáre**, ktoré sa uzatvárajú do zložených zátvoriek. Tieto však realizáciu algoritmu žiadnym spôsobom neovplyvňujú, ale slúžia skôr na jeho sprehľadnenie.



2 Zobrazenie informácie v počítači

2.1 Číselné sústavy

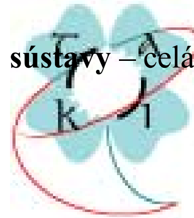
Najpoužívanejšou číselnou sústavou v bežnom živote je dekadická číselná sústava. Vychádza z arabského systému, ktorého súčasťou je aj číslica nula. **Základ číselnej sústavy** je tvorený počtom jedinečných neopakujúcich sa číslic (0, 1, 2,...,8, 9). Táto číselná sústava však nie je vhodná pre technickú realizáciu výpočtov na počítači (v aritmetickej jednotke), kde sa používa dvojková (binárna) číselná sústava (číslice základu 0, 1).V programovaní sú od nej odvodené číselné sústavy (osmičková – oktálová s číslicami základu 0–7, šestnástková – hexadecimálna s číslicami základu 0–9, A–F). Každá číselná sústava má teda svoj základ (jedinečné neopakujúce sa číslice), ktorých počet zároveň definuje základ pre **váhový faktor** – mocniteľa základu. Váhový faktor je určený poradím číslice sprava, počínajúc 0. Napr. hexadecimálne číslo $AF3_{16}$:

$AF3_{16} =$

$$3 * 16^0 = 3$$
$$15 * 16^1 = 240$$
$$10 * 16^2 = \underline{2560}$$
$$2803_{10} = 2 * 10^3 + 8 * 10^2 + 0 * 10^1 + 3 * 10^0 = 1010\ 1111\ 0011_2 =$$
$$= 101\ 011\ 110\ 011_2 = 5363_8$$

Na prevod medzi jednotlivými číselnými sústavami existuje mnoho metód. Medzi základné patria prevody medzi dekadickou a binárnou číselnou sústavou.

a) **Prevod z dekadickej do binárnej sústavy** – celá časť čísla (napr. 122):



122 / 2 = 61 zvyšok 0
61 / 2 = 30 zvyšok 1
30 / 2 = 15 zvyšok 0
15 / 2 = 7 zvyšok 1
7 / 2 = 3 zvyšok 1
3 / 2 = 1 zvyšok 1
1 / 2 = 0 zvyšok 1 $\Rightarrow 122_{10} = 1111010_2$

Postup je možné charakterizovať ako celočíselné delenie základom sústavy, do ktorej sa vykonáva prevod, pričom zvyšky delenia sa zapíšu v opačnom poradí.

b) **Prevod z dekadickej do binárnej sústavy** – desatinná časť čísla (napr. 0.120):

0.120 * 2 = **0**.240 (celá časť je 0)
0.240 * 2 = **0**.480 (celá časť je 0)
0.480 * 2 = **0**.960 (celá časť je 0)
0.960 * 2 = **1**.920 (celá časť je 1)
0.920 * 2 = **1**.840 (celá časť je 1)
0.840 * 2 = **1**.720 (celá časť je 1)
0.720 * 2 = **1**.480 (celá časť je 1)
0.480 * 2 = 0.960 (celá časť je 0, číslo je periodické) $\Rightarrow 0.120_{10} = 0.00\overline{01111}_2$

Postup je možné charakterizovať ako násobenie základom sústavy, do ktorej sa vykonáva prevod, pričom celá časť sa zanedbáva a zapisuje sa v poradí jej vzniku.

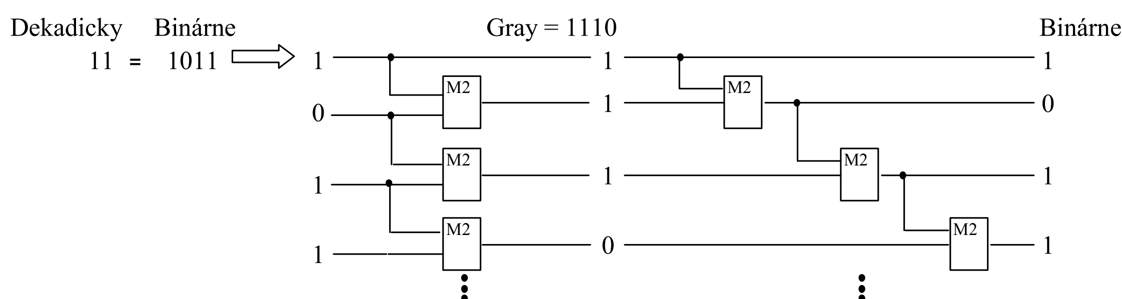
Pretože $8 = 2^3$ a $16 = 2^4$, je možné realizovať prevody medzi binárnou, oktálovou alebo hexadecimálnou sústavou združovaním (rozpisom) trojíc alebo štvoríc binárnych číslic.

Tab. 2.1 Prevody medzi číselnými sústavami

Dekadicky	Binárne	Oktálovo	Hexadecimálne
0	0 000	0 0	0
1	0 001	0 1	1
2	0 010	0 2	2
3	0 011	0 3	3
4	0 100	0 4	4
5	0 101	0 5	5
6	0 110	0 6	6
7	0 111	0 7	7
8	1 000	1 0	8
9	1 001	1 1	9
10	1 010	1 2	A
11	1 011	1 3	B
12	1 100	1 4	C
13	1 101	1 5	D
14	1 110	1 6	E
15	1 111	1 7	F

2.2 Číselné kódovanie informácie

Pre dve **binárne čísla** môžeme zaviesť pojem **Hammingova vzdialenosť**. Tá je daná počtom bitov, v ktorých sa porovnávané čísla líšia. Napr. pre susedné čísla 0 a 1 je Hammingova vzdialenosť 1, ale pre susedné čísla 7 a 8 je to už 4. Čísla je možné kódovať aj tak, aby medzi susednými číslami (v dekadickej sústave) bola vždy Hammingova vzdialenosť 1 (v binárnej sústave). Zabezpečí to tzv. **Grayov kód** (pozri Tab 2.2). Prevod z binárnej sústavy do Grayovho kódu a naopak je možné vyjadriť schémami uvedenými v Obr. 2.1.



Obr. 2.1 Schémy transformácií Grayovho kódu na príklade čísla 11

Obvod M2 predstavuje realizáciu operácie súčet modulo 2 (neekvivalencia, xor, $1 \otimes 1 = 0 \otimes 0 = 0$, resp. $1 \otimes 0 = 0 \otimes 1 = 1$). Zo schémy je zrejmé, že prevod z Grayovho kódu je výpočtovo náročnejší.

Tab. 2.2 Kódovanie čísel

Dekadicky	Binárne	Gray	BCD	2 z 5
0	0 000	0 000	0 000	0 0000
1	0 001	0 001	0 001	0 0011
2	0 010	0 011	0 010	0 0110
3	0 011	0 010	0 011	0 0101
4	0 100	0 110	0 100	0 1100
5	0 101	0 111	0 101	0 1001
6	0 110	0 101	0 110	0 1010
7	0 111	0 100	0 111	1 1000
8	1 000	1 100	1 000	1 0001
9	1 001	1 101	1 001	1 0010
10	1 010	1 111	–	–
11	1 011	1 110	–	–
12	1 100	1 010	–	–
13	1 101	1 011	–	–
14	1 110	1 001	–	–
15	1 111	1 000	–	–

Kódovanie znakov používa kód *ASCII* (*The* ***A****merican* ***S****tandard* ***C****ode for* ***I****nformation* ***I****nter-change*). Kód je 7 bitový a umožňuje kódovať 128 znakov uvedených v Tab. 2.3. Kódy sú obvykle vyjadrované v hexadecimálnej sústave (napr. kód znaku ‘A‘ je 41_{16}. Kódovacia tabuľka obsahuje:

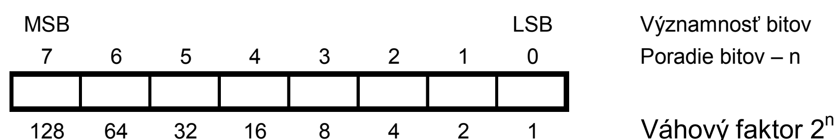
- špeciálne riadiace (ďalekopisné) znaky (NUL, SOH,..., DEL)
- znaky latinskej abecedy A ... Z, a ...z
- číslice 0 ... 9
- špeciálne znaky, ako medzera ‘ ‘ s kódom 20_{16}, ! “, atď.

Tab. 2.3 Kódovanie ASCII

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	NUL	SOH	STX	EXT	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&		(	)	*	+	,	–	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Y	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Každý znak je teda uložený v 1 byte. Pretože byte má 8 bitov využíva sa zvyšných 128 kódov pre národné abecedy (ktoré sa navzájom prekrývajú) a pre ďalšie špeciálne znaky.

Zobrazenie záporných čísel je možné realizovať viacerými spôsobmi. Buď sa využije znamienkový bit, alebo sa použije doplnkový inverzný kód.



Obr. 2.2 Schematické zobrazenie bytu

Byt je charakterizovaný ako osmica bitov, ktorých hodnotový význam je daný váhovým faktorom jeho poradia. Najvýznamnejší bit je označovaný MSB (Most Significant Bit) a najmenej významný bit LSB (Least Significant Bit). Pri zobrazení pomocou znamienkového bytu sa využíva MSB bit (1 = záporné číslo). Pri doplnkovom inverznom kóde sa záporné číslo vypočíta tak, že ku inverzii jeho absolútnej hodnoty (1 ... 128) sa pripočíta 1. Aj v tomto zobrazení bude MSB bit signalizovať záporné číslo. Napr.:

$-3 = \overline{0000\ 0011} + 1 = 1111\ 1100 + 1 = 1111\ 1101$

Celé čísla je teda možné ukladať priamo v základnom tvare alebo inom binárnom kóde. Využíva sa *n* bytov (slová s *n**8 bitov) a to najmä 8, 16, 32 prípadne 64 bitové slová (pozri Tab. 2.4).

Čísla s pohyblivou rádovou čiarkou slúžia na uloženie reálnych čísel. Obsahujú tri časti: znamienko, exponent a mantisu. Rozloženie bitov je definované normou IEEE Standard 754.

Obr. 2.3 Schéma zobrazenia čísla v pohyblivej rádovej čiarke

Pretože norma IEEE 754 pripúšťa dvojakú presnosť, je v Tab. 1.4 uvedené rozloženie bitov pre 32 a 64 bitové slovo.

Tab. 2.4 Rozloženie bitov podľa normy IEEE Std. 754–1985

Presnosť	Dĺžka [bit]	Znamienko [bit]	Exponent [bity]	Mantisa [bity]	Posun v mantise
Jednoduchá	32	1 [31]	8 [30–23]	23 [22–0]	127
Dvojnásobná	64	1 [63]	11 [62–52]	52 [51–0]	1023

Znamienkový bit (MSB) hodnotou 0 určuje kladné číslo a hodnotou 1 záporné číslo.
Exponent svojim rozsahom musí zobrazovať záporný aj kladný exponent. Preto je zavedený tzv. **posun**, t.j. číslo, ktoré sa ku skutočnému exponentu pripočíta. Exponent sa vypočíta v procese úpravy čísla do normalizovanej formy z intervalu (1, 0.5>. Napr. číslo 125_{10} sa transformuje nasledovne:
125 /2 → 62.5 /2 → 31.25 /2 → 15.625 /2 → 7.8125 /2 →3.90625 /2 → 1.953125 /2 → 0.9765625
teda 125 = 0.9765625 * 2^7. Potom bude exponent 7+posun = 7+127 =134_{10}= $1000\ 0110_2$.

Je zrejmé, že číslo v absolútnej hodnote menšie ako 0.5 sa bude násobiť. **Normalizovaná** dekadická **mantisa** (v predchádzajúcom príklade 0.9765625) sa transformuje do binárneho tvaru.
$0.9765625_{10} = 0.\mathbf{1111101}_2$.

Pretože normalizovaná časť binárnej mantisy začína vždy číslicou 1 netreba ju zobrazovať a stačí teda zobraziť ďalších 23 (52) bitov normalizovanej binárnej mantisy (teda $0.\mathbf{111101}_2$). V uvedenom príklade nie sú zobrazené pravostranné nulové bity mantisy.
Záverom poznamenávame, že norma IEEE 754 zavádza tzv. špeciálne čísla, ktoré umožňujú kódovať čísla, ktoré predchádzajúcim spôsobom nie je možné vytvoriť (+0, –0, $+\infty, -\infty$), ako aj kódy ktoré nepredstavujú žiadne číslo NaN (Not a Number). Kódy NaN je možné rozdeliť do dvoch skupín: SNaN (Signaling Not a Number) a QNaN (Quiet Not a Number). Kombinácie SNaN sa vyskytujú pri neplatných operáciách a QNaN pri operáciách, ktorých výsledky nie sú definované. Obe sú pri výskyte signalizované procesorom.

2.3 Neriešené príklady

Pr. 2.1 Transformujte celé číslo 37 110 do binárnej, oktálovej a hexadecimálnej sústavy.

Pr. 2.2 Transformujte desatinné číslo 121.25_{10} do binárnej, oktálovej a hexadecimálnej sústavy.

Pr. 2.3 Transformujte číslo 124_{10} do Grayovho kódu.

Pr. 2.4 Transformujte číslo 101101_2 z Grayovho kódu do dekadickej sústavy.

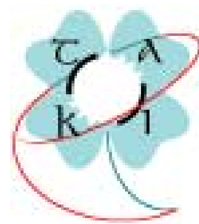
Pr. 2.5 Transformujte číslo –1024 do 16 bitového zobrazenia v doplnkovom inverznom kóde.

Pr. 2.6 Zistite, aká je Hammingova vzdialenosť znakov ‘L‘ a ‘S‘.

Pr. 2.7 Transformujte znaky ‘L‘ a ‘S‘ do Grayovho kódu a určite ich Hammingovu vzdialenosť.

Pr. 2.8 Vyriešte zobrazenie čísla –124.25 podľa normy IEEE 754 v jednoduchej presnosti.

Pr. 2.9 Vyriešte zobrazenie čísla –124.25 podľa normy IEEE 754 v dvojnásobnej presnosti.



3 Príklady a cvičenia z algoritmizácie

Príklad 1. Zostavte algoritmus, ktorý pre prirodzené číslo n ∈ N, zisti, či je párne alebo nepárne.

Riešenie. Ukážeme tri z možných spôsobov riešenia. Čitateľ iste pozná aritmetický operátor **mod** (čítaj modulo), ktorý predstavuje zvyšok po celočíselnom delení: X mod Y = Z, kde X je celé číslo, Y je prirodzené číslo, Z (zvyšok) je nezáporné celé číslo.

Napr. 7 mod 3 = 1 (pretože 7 = 3*2+1)
–7 mod 3 = 2 (pretože -7 = 3*(–3)+2 a zvyšok musí byť nezáporný).

Z definície operátora je zrejmé, že Z nadobúda hodnoty z množiny (0,1,2, ..., Y-1}. Teda vieme povedať, či je číslo párne alebo nepárne podľa toho, či jeho zvyšok po delení dvoma je jedna alebo nula. Prvým riešením problému potom môže byť algoritmus:

```
{ n ∈ N }
vstup(n);                          {načíta sa prirodzené číslo n}
ak n mod 2 = 0                     {ak je zvyšok 0, tak sa vypíše}
   tak výstup(n,“ je párne“)       {hodnota n a reťazec v apostrofoch}
   inak výstup(n,“ je nepárne“)    {inak je zvyšok 1}
*ak ;
```

V prípade, že procesor nepozná operátor mod, možno postupovať nasledovne: pokiaľ to bude možné, budeme odpočítavať od čísla n dvojku, až kým neostane číslo 1 alebo 0 a na základe zostatku rozhodneme, či pôvodné číslo bolo párne alebo nepárne.

```
{ n ∈ N }
vstup(n);                          {načíta sa prirodzené číslo n}
opakuj
   pokiaľ n > 1                    {opakovane sa zmenšuje hodnota}
   rob n := n – 2                  {čísla n o 2}
*opakuj ;                          {n=0 alebo n=1}
ak n = 0
   tak výstup(“číslo je párne“)
   inak výstup(“číslo je nepárne“)
*ak ;
```

Táto verzia však môže použiť len v tom prípade, ak nie je pôvodná hodnota čísla n ďalej potrebná (pripomíname, že opakovaným "prepisovaním" premennej n sa jej počiatočná hodnota stráca). Aby sme sa vyhli tomuto nedostatku (princíp neustáleho zlepšovania algoritmu), použijeme v algoritme pomocnú premennú, do ktorej sa na začiatku uloží pôvodná hodnota n a ďalej sa bude pracovať s touto premennou. Pôvodná hodnota premennej n zostáva počas celej realizácie algoritmu nezmenená.

```
{ n ∈ N }
vstup(n);
pom := n;
opakuj
   pokiaľ pom > 1
```

rob pom := pom – 2
***opakuj** ;
ak pom = 0 **tak** výstup(n, " je párne")
inak výstup(n, " je nepárne")
***ak** ;

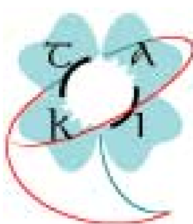
Príklad 2.

Zostavte algoritmus pre výpočet funkcie SIGN(x) definovanej pre všetky reálne čísla x ∈ R predpisom:

$$SIGN(x) = \begin{cases} -1, & \text{ak } x < 0 \\ 0, & \text{ak } x = 0 \\ 1, & \text{ak } x > 0 \end{cases}$$

Riešenie. Ako bolo spomenuté vyššie, príkazy vo vnútri štruktúrovaných príkazov môžu byť nahradené ďalšími štruktúrovanými príkazmi. Takže, ak x<0, premennej SIGN sa priradí hodnota –1. Ak nie, zostávajú ešte dve možnosti (x=0 alebo x>0), čo vyriešime ďalším podmieneným príkazom.

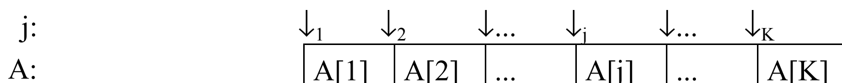
{ x ∈ R }
vstup(x); {načíta sa hodnota x}
ak x < 0 **tak** SIGN := –1 {podmienka je splnená, do premennej SIGN uloží hodnota –1}
inak
ak x = 0
tak SIGN := 0
inak SIGN := 1
***ak**
***ak** ;
výstup ("SIGN(", x, ") = ", SIGN) {výpis výsledku}
{ SIGN=SIGN(x) }



Príklad 3.

Na vstupe je postupnosť nenulových reálnych čísel zakončená nulou (t.j. čísla a_1, a_2, ..., 0). Napíšte algoritmus pre načítanie nenulových hodnôt tejto postupnosti do vektora A, ktorý zároveň vypíše ich počet.

Riešenie. Vektor v algoritmizácii (ale aj v programovaní) je možné interpretovať ako postupnosť pamäťových miest (buniek), do ktorých možno zapisovať. určité hodnoty. Na označenie polohy buniek použijeme riadiacu premennú j (index), ktorá bude postupne ukazovať na jednotlivé pamäťové miesta (v poradí od prvého po posledné), do ktorých sa budú zapisovať hodnoty načítané zo vstupu. Obsah j–tej bunky bude uložený v tzv. **indexovanej premennej** A[j].



j:	↓1	↓2	↓...	↓j	↓...	↓K
A:	A[1]	A[2]	...	A[j]	...	A[K]

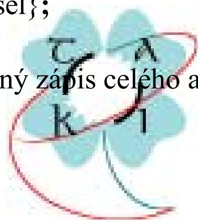
Pri načítaní postupnosti budeme postupovať nasledovne: Načíta sa prvá hodnota zo vstupu do pomocnej premennej napr. ČÍSLO (aby sme do vektora nezapísali aj nulu). Ak platí ČÍSLO>0, tak sa (index) premenná j nastaví na prvú voľnú bunku vektora A, do nej sa zapíše hodnota premennej ČÍSLO. Postup sa opakuje, pokiaľ sa do premennej ČÍSLO nenačíta 0. Teda na vstup postupnosti je vhodné použiť úplný cyklus.
Všimnime si, že indexová premenná obsahuje vždy na posledný zapísaný prvok vektora A, teda počet prvkov vektora A sa rovná hodnote premennej j (indexu).

```
j := 0 ;                              {počiatočné nastavenie indexu}
opakuj
   vstup(ČÍSLO)                       {načítanie hodnoty zo vstupu}
  pokiaľ ČÍSLO > 0 rob j := j +l ;    {nastavenie premennej s indexom na prvú voľnú bunku}
         A[ j ] := ČÍSLO              {zápis hodnoty do vektora}
*opakuj ;
výstup("Počet načítaných čísel je : ", j );
```

Pozrime sa, aké premenné vystupujú v algoritme. Je to premenná j, ktorá je zrejme celočíselná, reálna premenná ČÍSLO a okrem nich sa stretávame s novým typom premennej A, ktorý sa nazýva **pole**.

Pole si je možné predstaviť ako konečnú tabuľku pozostávajúcu z pevného počtu buniek, ktoré sú očíslované za sebou idúcimi celými číslami (alebo označené za sebou idúcimi hodnotami niektorého iného ordinálneho typu) – indexmi. Do buniek sa môžu vkladať hodnoty určitého (rovnakého) typu. Interva1 indexov sa zapisuje v lomených zátvorkách (viď nižšie). V našom prípade, ak predpokladáme, že na vstupe nebude viac ako 50 čísel, očíslujeme bunky indexmi 1 až 50 a do nich budeme zapisovať reálne čísla. Takže do deklarácie zapíšeme:

```
   prem j        : celočíselné;
        ČÍSLO    : reálne;
        A        : pole [1..50] reálnych {čísel};
```



Pre úplnosť uvádzame na záver nekomentovaný zápis celého algoritmu.

```
Alg Vstup_postupnosti ;
prem  j        : celočíselné ;
      ČÍSLO    : reálne ;
      A        : pole [1..50] reálnych ;
začiatok
j := 0 ;
opakuj
   vstup(ČÍSLO)
   pokiaľ ČÍSLO>0 rob   j := j +l ;
                        A[j] := ČÍSLO
*opakuj ;
výstup("Počet načítaných čísel je : ", j)
koniec .
```

Neriešené príklady

1. Riešte príklad 3 za predpokladu, že do vektora treba zapísať všetky prvky postupnosti na vstupe aj s nulou.
2. Doplňte algoritmus z príkladu 3 o výpis prvkov vektora A.
3. Na vstupe je daná postupnosť reálnych čísel $\{a_1, a_2, \dots, a_k\}$, kde $a_i <> a_k$ pre všetky i. Zostavte algoritmus pre načítanie hodnôt a_i do vektora A.

Príklad 4. Zostavte algoritmus na vstup vektora B, ktorý má K prvkov.

Riešenie. Najskôr je potrebné zistiť počet prvkov vektora B. Ak vieme koľko prvkov bude vektor obsahovať, môže sa na jeho vstup použiť cyklus so známym počtom opakovaní. V algoritme sa bude postupne čítať K+1 hodnôt. Aby sme vedeli, ktorú hodnotu práve ideme načítať, dáme pred každý príkaz vstupu príslušnú výzvu vo forme príkazu výstupu.

```
Alg Vstup_vektora ;
prem j, K    : celočíselné ;
       B     : pole [1..50] reálnych ;
začiatok
výstup("Zadaj počet prvkov vektora: ");
vstup(K);
pre j : =1 po K rob výstup( "Zadaj hodnotu B[ ", j, " ]: " ) ;
                    vstup(B[ j ])
*pre
koniec .
```

Pre porovnanie uvádzame aj riešenie pomocou cyklu s podmienkou na začiatku.

```
Alg Vstup_vektora ;
prem j, K    : celočíselné ;
       B     : pole [1..50] reálnych ;
začiatok
vstup("Zadaj počet prvkov vektora: ") ;
vstup(K);
j : =1;                         {nastavenie premennej – indexu j na prvú voľnú bunku}
opakuj
   pokiaľ j ≤ K
   rob  výstup("Zadaj hodnotu B[ ", j, " ]: ");
        vstup(B[ j ]);
        j := j +l               {nastavenie nasledujúcej hodnoty indexu}
*opakuj
koniec .
```

Neriešené príklady

1. Na vstupe je pripravených n hodnôt. Zostavte algoritmus, ktorý načíta nenulové hodnoty tejto postupnosti do poľa B, vypíše ich počet a nakoniec ich vypíše nenulové hodnoty.

Príklad 5. Zostavte algoritmus pre vstup matice A[M,N] po riadkoch.

Riešenie. Z predchádzajúceho príkladu vieme načítať vektor o K (alebo N) prvkoch. Maticu typu M x N možno chápať ako M riadkových vektorov (s N prvkami) pod sebou s tým rozdielom, že pozícia každého prvku matice je určená dvoma indexmi – indexom riadku a indexom stĺpca. To vlastne znamená, že treba M–krát realizovať algoritmus pre vstup vektora, ktorý má N prvkov.
Matica A sa bude čítať po riadkoch. Nastavíme sa na prvý riadok a načítame prvky všetkých stĺpcov (a_{11}, a_{12}, ..., a_{1n}), potom na druhý riadok a znova načítame prvky všetkých stĺpcov atď., až kým sa nenačítajú všetky

riadky. Z uvedeného vyplýva, že index stĺpcov sa mení rýchlejšie ako index riadkov, to znamená, že najskôr použijeme cyklus, ktorý zabezpečí prechod po jednotlivých riadkoch a v ňom bude vnorený cyklus na prechod po jednotlivých stĺpcoch. Algoritmus uvedieme v kompletnom tvare, pričom upozorňujeme čitateľa, aby si všimol deklaráciu premennej A ako dvojrozmerného poľa (prvý rozmer udáva počet riadkov, druhý rozmer počet stĺpcov).

```
Alg Vstup_matice;
prem i, j, N,M     : celočíselné;
     A             : pole [1..50,1..60] reálnych;
začiatok
výstup("Zadaj počet riadkov a stĺpcov matice: ");
vstup(M, N);
pre i : =1 po M                    {nastaví sa postupne prvý až M–ty riadok }
   rob                             {čítaj celý riadok }
   pre j : =1 po N
   rob  výstup("Zadaj prvok A[", i, " , " j, "]: ");
   vstup(A[ i, j ]);               {načíta sa prvok riadku}
   *pre                            {koniec čítania riadku}
*pre
koniec .
```

Príklad 6. Zostavte algoritmus pre vstup štvorcovej matice A[N,N] po riadkoch.

Riešenie. Vzhľadom na rozmer matice, treba v prechádzajúcom algoritme zmeniť rozmer premennej A, načítať jeden rozmer matice a upraviť cyklus prechodu po riadkoch. Poznamenávame však, že predchádzajúci algoritmus sa dá využiť, ak pri vstupe rozmerov matice sa zadajú oba rozmery rovnaké. Treba si zároveň všimnúť, že sa v algoritme nevykonáva kontrola správnosti zadania maximálnych rozmerov matice, čo môže spôsobiť zrútenie algoritmu, ak sa presiahnu rozmery deklarovanej matice. Na kontrolu rozmerov treba použiť úplný cyklus s alebo cyklus podmienkou na konci.

```
opakuj   {načítajú sa rozmery matice}
   výstup("Zadaj rozmer štvorcovej matice:" );
   vstup(N)
   pokiaľ (N<1 ∪ N>50)
          rob     výstup("Zadávaný rozmer štvorcovej matice musí byť z intervalu <1, 50>" )
*opakuj ;
```

Neriešené príklady

1. Zostavte algoritmus pre vstup štvorcovej (obdĺžnikovej) matice po stĺpcoch.
2. Zostavte algoritmus pre vstup dvoch matíc X[M,N] a Y[M,N].

Príklad 7.
Navrhnite algoritmus na zostavenie jednotkovej (štvorcovej) matice A[N,N].

Riešenie. Jednotková matica má na hlavnej diagonále samé jednotky a na všetkých ostatných pozíciách nuly. Prvok matice $a_{i,j}$ sa nachádza na hlavnej diagonále vtedy, ak sa index jeho riadku rovná indexu stĺpca, t.j. ak i = j. Keďže vieme dopredu povedať, aké hodnoty sa budú v matici nachádzať, použijeme na zostavenie matice namiesto príkazu vstupu priraďovací príkaz.

```
Alg Generovanie_diagonálnej_matice;
prem i, j, N : celočíselné;
     A       : pole [1..N,1..N] celočíselných; {N treba nahradiť maximálnym prípustným rozmerom}
začiatok
výstup("Zadaj rozmer matice: ");
vstup(N);
pre i : =1 po N
   rob pre j : =1 po N
   rob    ak i = j tak A[i , j]:=1
                   inak A[i , j]:=0
        *ak
      *pre
*pre
koniec .
```

Neriešené príklady

1. Navrhnite algoritmus na zostavenie matice A[N,N], ktorá má na vedľajšej diagonále samé jednotky a všade inde nuly.
2. Navrhnite algoritmus na zostavenie hornej (jednotkovej) trojuholníkovej matice.
3. Zostavte algoritmus na výpočet súčtu dvoch matíc C[M,N]=A[M,N]+B[M,N], (t.j. $c_{ij}=a_{ij}+b_{ij}$)
4. Zostavte algoritmus na výpočet rozdielu dvoch matíc C(M,N]=A[M,N]-B[M,N], (t.j. $c_{ij}=a_{ij}\ b_{ij}$)
5. Zostavte algoritmus na tzv. ekvivalentné riadkové úpravy matice:
   a) násobenie k–teho riadku číslom x, rôznym od nuly,
   b) pripočítanie lineárnej kombinácie daných riadkov k inému určenému riadku
   c) výmena dvoch riadkov matice
6. Zostavte algoritmus na vytvorenie transponovanej matice k matici A[M,N].

Príklad 8.

Zostavte algoritmus, ktorý transformuje maticu A[M,N] na vektor V[M*N] po riadkoch.

Riešenie.

Najskôr sa načíta matica A. Vektor V bude vyzerať nasledovne: V=(a_{11}, a_{12}, ..., a_{1n}, a_{21},a_{22}, ..., a_{mn}). Aká bude pozícia prvku a_{ij} vo vektore V ? Pred tým, než sa zapíše prvok a_{ij}, treba, zapísať všetky prvky v prechádzajúcich riadkoch matice, ktorých je (i–1), z čoho každý má N prvkov. V i–tom riadku sa prvok a_{ij} nachádza v j–tom stĺpci. Teda pozícia prvku a_{ij} vo vektore V bude (i–1)*N + j. Transformácia sa vykoná postupným prechodom po všetkých prvkoch matice A postupným priradením ich hodnôt príslušným prvkom vektora.

```
Alg Transformácia_matice_po_riadkoch_1;
prem i, j, M, N    : celočíselné;
     A             : pole [1..M, 1..N] reálnych;
     V             : pole [1..M*N] reálnych;
začiatok
výstup("Zadaj počet riadkov a stĺpcov matice: ");
vstup(M, N);                              {načítanie matice A}
pre i : =1 po M
          rob pre j:=1 po N
               rob   výstup("Zadaj prvok A[", i, " , ", j, "]: ");
```

```
                    vstup(A[ i, j ] );
              *pre
*pre;
pre i : =1 po M                                  {zostavenie vektora V}
          rob pre j:=1 po N
              rob  V[ (i –1)*N +j]:=A[i , j]
              *pre
*pre
koniec .
```

Uvedieme ešte jedno riešenie daného problému. V tomto prípade použijeme ďalšiu premennú (index), ktorá bude postupne "ukazovať" na jednotlivé prvky vektora V, do ktorých sa zapíšu prvky matice A.

```
Alg Transformácia_matice_po_riadkoch_2;
prem i, j, k, M, N  : celočíselné;
     A              : pole [1..M,1..N] reálnych;
     V              : pole [1..M*N] reálnych;
začiatok
výstup(“Zadaj počet riadkov a stĺpcov matice: “);
vstup(M, N);                                 {načítanie matice A}
pre i : =1 po M
          rob pre j:=1 po N
              rob  výstup(“Zadaj prvok A[“, i, “ , “ , j, “]: “);
                   vstup(A[i , j ] );
              *pre
*pre
k :=1;                                       {nastavenie indexu vektora V }
pre i : =1 po M                              {prechod po riadkoch}
          rob                                {matice A}
          pre j :=1 po N                     {prechod po stĺpcoch A}
                   rob V[ k ] :=A[i , j];    {zostavenie prvku vektora}
                       k :=k +1              {prechod na nasledujúci prvok vektora V }
        *pre
*pre
koniec.
```

Neriešené príklady

1. V prvom riešení príkladu 8. sa pri zostavovaní vektora V vo vnútornom cykle v každom prechode vypočítava hodnota výrazu (i–1)*N, ktorá nezávisí od riadiacej premennej cyklu (j). Zostavte efektívnejší algoritmus, ktorý tento nedostatok odstráni.
2. Zostavte algoritmus, ktorý transformuje maticu A[M,N] na vektor V[M*N] po stĺpcoch.

Príklad 9.
Zostavte algoritmus na výpočet súčtu prvkov vektora V[M].

Riešenie.

Načíta sa vektor V. Použijeme premennú SÚČET, do ktorej sa na začiatku vloží hodnota 0. K hodnote premennej SÚČET sa pripočíta hodnota prvého prvku vektora. K výsledku sa pripočíta . hodnota ďalšieho prvku. Postup sa cyklicky opakuje, až kým sa nepripočítajú všetky prvky vektora V. Nakoniec sa hodnota premennej SÚČET vypíše.

```
ALG Suma;
prem i, M        : celočíselné;
     SÚČET       : reálne;
     V           : pole [1..M] reálnych;
začiatok
výstup(“Zadaj počet prvkov vektora: “);
vstup(M);                                  {načítanie vektora V}
pre i :=1 po M
     rob   výstup(“Zadaj prvok V[“, i, “] :“);
           vstup(V[I])
*pre;
SÚČET :=0;                                 {inicializácia premennej SÚČET }
pre i :=1 po M                             {Výpočet súčtu}
     rob SÚČET := SÚČET + V[ i ]
*pre ;
výstup(“Súčet prvkov vektora V = “, SÚČET)   {výpis výsledku}
koniec .
```

Neriešené príklady

1. Zostavte algoritmus na výpočet súčtu prvkov matice B[M, N].
2. Zostavte algoritmus na výpočet aritmetického priemeru prvkov množiny X={x_1, ..., x_n}.

$$X_p = \frac{\sum_{i=1}^{n} X_i}{n}$$

3. Zostavte algoritmus na výpočet geometrického priemeru prvkov množiny X={x_1, ..., x_n}.

$$X_g = \sqrt[n]{\prod_{i=1}^{n} X_i}$$

4. Zostavte algoritmus na výpočet sústavy dvoch lineárnych rovníc.
5. Zostavte algoritmus, ktorý realizuje súčin dvoch obdĺžnikových matíc C[M, P]=A[M, N]*B[N, P].

$$C_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj}$$

6. Zostavte algoritmus na výpočet smerodajnej odchýlky prvkov množiny X={x_1, ..., x_n}.

$$S_x = \sqrt{\frac{\sum_{i=1}^{n}(X_i - X_p)^2}{n-1}}, \text{ kde } \quad X_p = \sum_{i=1}^{n}(X_i)$$

Príklad 10.

Zostavte algoritmus, ktorý z množiny $A=(a_1, ..., a_n\}$ vytvorí množinu $B=(b_1, ..., b_n\}$ s prvkami $b_i > 0$.

Riešenie.

Načíta sa množina A. Použijeme pomocnú indexovú premennú napr. k, ktorá bude ukazovať na doteraz posledne zapísaný prvok množiny B. Keďže množina B môže, byť aj prázdna, na začiatku sa vloží do premennej k hodnota 0. Potom budeme postupne prezerať všetky prvky množiny A; ak bude prvok kladný, priradí sa jeho hodnota nasledujúcemu voľnému prvku množiny B. Nakoniec sa vypíšu všetky prvky množiny B.

```
ALG Kladný_výber;
prem i, k, N: celočíselné;
     A, B   : pole[1 .. N] reálnych;
začiatok
výstup("Zadaj počet prvkov množiny A: " ) ;
vstup(N);
pre i:=1 po N
        rob  výstup("Zadaj prvok A[", i, " ] :");
             vstup(A[ I ])
*pre;
k :=0; {nastavenie indexovej}
   {premennej v množine B}
pre i := 1 po N
        rob   ak A[ i ] > 0 tak k := k + 1;             {zápis kladného prvku do}
                 B[ k ] := A[ i ]                       {množiny B}
              *ak
*pre;
výstup(" Množina B pozostáva z prvkov : ");             {výpis prvkov}
pre i := 1 po k                                         {množiny B}
         rob výstup(B[i], " , ")
*pre
koniec .
```

Neriešené príklady

1. Zostavte algoritmus, ktorý z množiny $A=\{a_1, ..., a_n\}$ vytvorí množinu $B=\{b_1, ..., b_k\}$ s prvkami $b_i <> 0$.
2. Zostavte algoritmus, ktorý zistí, pre koľko prvkov množiny $A=\{a_1, ..., a_n\}$ platí, že $a_i > 0$.
3. Zostavte algoritmus, ktorý z množiny A(N) vytvorí množinu B(K), pre ktorej prvky platí $b_i > 0$ a množinu C(L), pre ktorej prvky platí $c_j \leq 0$.

Príklad 11.

Zostavte algoritmus, ktorý zistí, či sa v množine A nachádza prvok X. (Na vstupe sú hodnoty: N, $a_1,a_2, ..., a_n$, X).

Riešenie.

Bude sa hľadať prvý výskyt prvku X množine A. Načíta sa množina A a prvok X v tom poradí, ako sú na vstupe. Potom sa skúma prvý prvok množiny A. Ak sa jeho hodnota nerovná hodnote prvku X, nastaví sa na ďalší prvok. Postup sa môže opakovať, až v množine A sa nenájde hľadaný prvok. Teda môže sa použiť:

```
i := 1;
opakuj
      pokiaľ A[ i ]<>X
      rob      i := i +1
*opakuj ;
```

Čo sa však stane v prípade, keď sa prvok X v množine A nenachádza? Zrejme sa cyklus bude opakovať do nekonečna, pretože jeho podmienka bude stále platná. Aby sme sa vyhli tomuto tzv. "zacykleniu", treba podmienku cyklu rozšíriť o kontrolu rozmeru množiny A, t.j. (A[i]]<>X) a súčasne (i ≤ n). Cyklus teda skončí, ak bude podmienka cyklu nepravdivá. To nastane vtedy, keď bude aspoň jedna formula v podmienke nepravdivá. Ak nebude pravdivá prvá formula (t.j. nastane rovnosť A[i]=X pre niektoré i), vtedy sa prvok X v množine A nachádza na pozícii i. Ak nebude pravdivá druhá formula (t.j. i>n), vtedy sme už prezreli celú množinu A, pričom prvok X sa v nej nenašiel.

```
ALG Nájdi_prvok_v_množine;
prem i, N    : celočíselné;
       X     : reálne;
       A     : pole[1..N] reálnych;
začiatok
výstup("Zadaj počet prvkov množiny A: ");
vstup( N );                                       {načítanie množiny A}
pre i := 1 po N
          rob   výstup("Zadaj prvok A[", i, "]: ");
                vstup(A[ i ])
*pre;
výstup("Zadaj prvok X: ");
vstup( X );                                       {načítanie prvku X}
i := 1;                                           {nastavenie indexu na začiatok množiny A }
opakuj   {prezeranie prvkov A}
   pokiaľ (A[i]<>X) a súčasne (i ≤ N)
   rob i := i +1
*opakuj ;
ak i >N   (výpis výsledku}
   tak    výstup(X," sa v množine nenachádza")
   inak   výstup(X," sa nachádza na pozícii: ", i)
*ak
koniec .
```

Príklad 12.

Zostavte algoritmus na zjednotenie dvoch množín C(K)=A(N)+B(M), ktorých prvky sa neopakujú.

Riešenie.

Pri riešení tohto problému budeme vychádzať z predchádzajúcich algoritmov. Použijeme pomocnú premennú k, ktorá bude ukazovať na posledný zapísaný prvok množiny C. Je zrejmé, že množina C bude obsahovať celú množinu A. To znamená, že všetky jej prvky možno zapísať do množiny C. Z množiny B je však možné do C

zapísať len tie prvky, ktoré sa v množine A nenachádzajú. Teda budeme prezerať jednotlivé prvky množiny B a zisťovať, či sa nachádzajú v množine A (namiesto X budeme hľadať B[j], pre j =1, ..., M). Ak zistíme, že sa prvok B[j] v A nenachádza, zapíše sa do nasledujúceho voľného prvku množiny C. Nakoniec sa vypíšu všetky prvky množiny C.

ALG Zjednotenie_množín;
prem i, j, k, N, M : celočíselné;

A : pole[1..N] reálnych;

B : pole[1..M] reálnych;

C : pole[1..(M+N)] reálnych;

začiatok
výstup("Zadaj počet prvkov množiny A: ");
vstup(N);
pre i := 1 **po** N {načítanie množiny A}

rob výstup("Zadaj prvok A[", i, "]: ");

vstup(A[i])

***pre** ;

Príklad 12.

Zostavte algoritmus na výpočet zjednotenia dvoch množín C{K}=A{N}+B{M}, ktorých prvky sa neopakujú.
Riešenie.

Pri riešení tohto problému budeme vychádzať z predchádzajúcich algoritmov. Použije sa pomocná premenná k, ktorá bude ukazovať na posledný zapísaný prvok množiny C. Je zrejmé, že množina C bude obsahovať celú množinu A. To znamená, že všetky jej prvky možno zapísať do množiny C. Z množiny B je však možné do C zapísať len tie prvky, ktoré sa v množine A nenachádzajú. Teda budeme prezerať jednotlivé prvky množiny B a zisťovať, či sa nachádzajú v množine A (miesto X budeme hľadať B[j], pre j = 1, ... , M). Ak zistíme, že sa prvok B[j] v A nenachádza, zapíše sa do nasledujúceho voľného prvku množiny C. Nakoniec sa vypíšu všetky prvky množiny C.

ALG Zjednotenie_množín;
Prem i, j, k, N, M: celočíselné;

A: pole [1 .. N] reálnych;

B: pole [1 .. M] reálnych;

C: pole [1 .. (M+N)] reálnych;

začiatok
výstup("Zadaj počet prvkov množiny A: ");
vstup (N); {načítanie množiny A}
pre i := 1 **po** N

rob výstup("Zadaj prvok A[", i, "]: ");

vstup(A[i])

```
*pre;
výstup (“Zadaj počet prvkov množiny B: “);
vstup ( M );                                  {načítanie množiny B}
pre j := 1 po M
   rob  výstup(“Zadaj prvok B[“, j, “]: “);
          vstup(B[ j ])
*pre;
pre i := 1 po N                               {zápis prvkov množiny}
   rob C[ i ] := A[ i ];                      {prvok z A do C}
*pre;
k := N;                                       {nastavenie indexu na posledný prvok v C }
pre  j := 1 po M                              {prechod po prvkoch množiny B }
   rob
         i := 1;                              {hľadanie prvku B[ j ] v A}
      opakuj
         pokiaľ (A[i]<>B[j]) a súčasne (i ≤ n)
         rob i := i + 1
      *opakuj ;
      ak i > N                                {ak B[ j ] nepatrí do A}
         tak  k := k +1;                      {posun sa na voľný prvok}
              C[k] := B[ j ]                  {a zápis do množiny C}
     *ak
*pre;
výstup (“Množina C pozostáva z prvkov : “);   {výpis množiny C}
pre i := 1 po k
   rob výstup(C[ i ], “ , “)
*pre
koniec .
```

Cvičenia.

1. Zostavte algoritmus na výpočet rozdielu dvoch množín C{K}=A{N}–B{M}, ktorých prvky sa neopakujú.
   Návod: Prvok množiny A sa priradí do množiny C len vtedy, ak sa nenachádza v množine B.
2. Zostavte algoritmus na výpočet prieniku dvoch množín C{K}=A{N}∩B{M}, ktorých prvky sa neopakujú.
   Návod: Prvok množiny A sa priradí do množiny C len vtedy, ak sa nachádza aj v množine B.

Príklad 13.

Zostavte algoritmus na zámenu dvoch prvkov A, B postupnosti X = {x_1, x_2, ..., x_n}.

Riešenie:

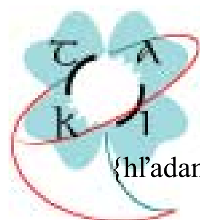
Po načítaní postupnosti X, treba zistiť pozíciu oboch prvkov v postupnosti (pozA, pozB) a potom vymeniť hodnoty na týchto pozíciách. Vzhľadom na to, že obe hodnoty budú vlastne uložené dva krát (raz ako jednoduchá premenná a druhý raz ako prvok poľa X), netreba na výmenu použiť ďalšiu premennú. (Pozri A.6 vyššie). Predpokladáme, že oba prvky sa určite v množine X nachádzajú, t.j. nebudeme v podmienke cyklu kontrolovať dĺžku postupnosti X.

```
ALG Výmena_dvoch _prvkov_postupnosti;
Prem i, N, pozA, pozB     : celočíselné;
     A , B                : reálne;
     X                    : pole[1 .. N] reálnych;
začiatok
výstup(“Zadaj počet prvkov postupnosti X: “);
vstup( N );                                    {načítanie postupnosti X}
pre i := 1 po N
  rob  výstup(“Zadaj prvok X[“, i, “]: “);

       vstup( X[ i ] )
*pre;
výstup(“Zadaj prvok A : “ );
vstup( A );                                    {načítanie prvku A}

výstup(“Zadaj prvok B : “);

vstup( B );                {načítanie prvku B}
i := 1;                                        {hľadanie prvku A v postupnosti X }
opakuj
  pokiaľ (X[ i ]< >A)
  rob i := i +l
*opakuj;
pozA := i;
i := 1;                                        {hľadanie prvku B v postupnosti X }
opakuj
  pokiaľ (X[ i ]< >B)
  rob i := i +l
*opakuj;
pozB := i;
X[ pozA ] := B;                                {výmena prvkov na}
X[ pozB ] := A;                                {nájdených pozíciách}
koniec .
```



Príklad 14.

Zostavte algoritmus na cyklický posun prvkov postupnosti A = $\{a_1, ..., a_n\}$o 1 prvok vľavo.

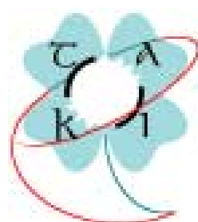
Riešenie.

Cyklický posun postupnosti A o jeden prvok vľavo vytvorí z postupnosti A postupnosť $\{a_2, a_3, ...,a_n, a_i\}$. To znamená, že na prvú pozíciu treba presunúť prvok a_2 (dostaneme $\{a_2, a_2, a_3, ..., a_n\}$), na druhú a_3 (dostaneme $\{a_2, a_3, a_3, a_4, ..., a_n\}$), na i–tu prvok a_{i+1}, kde i =1, ... , n–1 (dostaneme $\{a_2, a_3, a_4, ..., a_n, a_n\}$). Nakoniec treba na n–tú pozíciu zapísať prvok a_1. Aby sa jeho pôvodná hodnota "posúvaním" n–1 nasledujúcich prvkov neprepísala, treba ju na začiatku uložiť do pomocnej premennej napr. POMOC a po posune nasledujúcich prvkov priradiť na n–tú pozíciu postupnosti. Pre kontrolu necháme vypísať pôvodnú aj posunutú postupnosť.

```
ALG Cyklický_posun_vľavo;
Prem i, N       : celočíselné;
     POMOC : reálne;
     A          : po1e[1 .. N] reálnych;
```

```
začiatok
výstup ("Zadaj počet prvkov postupnosti A : ");
vstup( N );                                      {načítanie postupnosti A}
pre i:=1 po N
   rob   výstup("Zadaj prvok A[", i, "]: ");

         vstup(A[ i ])
*pre;
výstup(" Pôvodná postupnosť A : ")               {výpis pôvodnej postupnosti}
pre i := i po N
   rob výstup(A[i] ,' , ' );
*pre;
POMOC := A[1];
pre i := 1 po N-1                                {posun nasledujúcich prvkov }
   rob A[ i ] := A[i +l]
*pre;
A[N] := POMOC;                                   {presun 1. prvku na koniec}
výstup("Posunutá postupnosť A : ");              {výpis posunutej postupnosti}
pre i := 1 po N
   rob   výstup(A[ i ], ",");
*pre
koniec .
```



Príklad 15.

Zostavte algoritmus na cyklický posun prvkov postupnosti A= $\{a_1, ..., a_n\}$ o 1 prvok vpravo.

Riešenie.

Cyklický posun postupnosti A o 1 prvok vpravo vytvorí z postupnosti A postupnosť $\{a_n, a_1, a_2, ..., a_{n-1}\}$. Ak by sme posunuli prvok a_1, na druhú pozíciu, dostali by sme postupnosť $\{a_1, a_1, a_3, ..., a_n\}$, tým by sa však "stratila" hodnota a_2. Treba teda postupovať z opačného konca. Najskôr sa uloží hodnota a_n do pomocnej premennej POMOC, potom sa na n–tú pozíciu presunie prvok a_{n-1} (vynikne $\{a_1, a_2, ..., a_{n-1}, a_{n-1}\}$), potom na pozíciu (n–1) prvok a_{n-2}, atď., až na druhú pozíciu prvok a_1. Nakoniec sa na prvú pozíciu uloží pôvodná hodnota prvku a_n. Vzhľadom na to, že postupnosť presunu prvkov je klesajúca (n, n–1, ..., 2), použijeme na presun cyklus pre s klesaním.

```
ALG Cyklický_posun_vpravo;
Prem     i, N       : celočíselné;
         POMOC      : reálne;
         A          : pole[1 .. N] reálnych;
začiatok
výstup(" Zadaj počet prvkov postupnosti A : ");
vstup( N );                                        {načítanie postupnosti A}
pre i := 1 po N

   rob   výstup("Zadaj prvok A[", i, "]: ");

         vstup( A[ i ] )
*pre ;
```

```
výstup("Pôvodná postupnosť A : ");          {výpis pôvodnej postupnosti}
pre  i := 1 po N
   rob výstup(A[ i ], ",");
*pre;
POMOC := A[N];                              {uloženie posledného prvku}
pre i := N klesaním po 2                    {posun predchádzajúcich}
    rob    A[ i ]:=A[ i –1]                 {prvkov}
*pre;
A[ 1 ] := POMOC;                            {presun posledného prvku na začiatok }
výstup("Posunutá postupnosť A : ");         {výpis posunutej postupnosti}
pre  i := 1 po N
   rob    výstup(A[ i ], ", ");
*pre
koniec  .
```

Cvičenia.

1. Zostavte algoritmus na cyklický posun o k prvkov vľavo.
2. Zostavte algoritmus na cyklický posun o k prvkov vpravo.
3. Zostavte algoritmus, ktorý vloží prvok X do vzostupne usporiadanej postupnosti A={a_1, ..., a_n} tak, že postupnosť bude po vložení prvku rovnako usporiadaná.
4. Zostavte algoritmus ktorý, vloží prvok X do zostupne usporiadanej postupnosti A= {a_1, ..., a_n} tak, že postupnosť bude po vložení prvku rovnako usporiadaná.
5. Zostavte algoritmus na zlúčenie dvoch rovnako usporiadaných postupností A = {a_1, a_2, ..., a_N} a B = {b_1, b_2, ..., b_M} do postupnosti C={c_1, c_2, ..., c_{M+N}}, ktorá bude tak isto usporiadaná.
6. Zostavte algoritmus na zlúčenie dvoch usporiadaných postupností A, B do jednej postupnosti A tak, že A bude tiež usporiadaná (vkladaním prvkov b_i do postupnosti A na príslušné pozície).

Príklad 16.

Zostavte algoritmus, ktorý nájde minimálny prvok postupnosti A= {a_1, ..., a_n}.

Riešenie.

Pri riešení problému budeme vychádzať z predpokladu (hypotéza), že minimálny je prvý prvok postupnosti A. Použijeme premennú MIN, do ktorej na začiatku vložíme hodnotu a_1. Potom budeme postupne prezerať všetky nasledujúce prvky. Ak hodnota niektorého z nich bude menšia ako doterajšia hodnota premennej MIN, potom sa do premennej MIN uloží hodnota tohto prvku (korekcia hypotézy). Nakoniec sa nájdená minimálna hodnota vypíše.

```
ALG Hľadanie_minima;
Prem      i, N      : celočiselné:
          MIN       : reálne;
          A         : pole[1 .. N] reálnych;
začiatok
výstup("Zadaj počet prvkov postupnosti A : ");
vstup( N );                                 {načítanie postupnosti A}
pre  i := 1 po N
    rob    výstup("Zadaj prvok A[", i, "]: ");

           vstup(A[ i ])
```

 ISBN 80–228–0428–2

```
*pre;
MIN := A[1];                                {predpoklad, že min = a1}
pre   i :=2  po N
   rob ak A[ i ] < MIN                      {porovnanie s prvkami postupnosti}
         tak MIN := A[ i ]
        *ak
*pre;
výstup(“Minimum postupnosti  = “, MIN)      {výpis nájdenej hodnoty}
koniec .
```

Cvičenia.

1. Zostavte algoritmus, ktorý nájde minimálny a maximálny prvok matice A[M, N].
2. Zostavte algoritmus, ktorý nájde prvok postupnosti A, ktorý sa najmenej líši od prvku X, vypíše jeho hodnotu a polohu v postupnosti.
3. Zostavte algoritmus, ktorý presunie najväčší prvok postupnosti A= $\{a_1, ..., a_n\}$ na koniec.
4. Zostavte algoritmus, ktorý presunie najväčší prvok postupnosti A= $\{a_1, ..., a_n\}$ na začiatok.
5. Zostavte algoritmus na usporiadanie postupnosti A= $\{a_1, a_2, ..., a_n\}$ (presúvaním menšieho prvku na začiatok).

Príklad 17.

Zostavte algoritmus, ktorý nájde 2 najmenšie prvky postupnosti A = $\{a_1, ..., a_n\}$ (napr. A= {3,2,7,2,9,8}).

Riešenie.

Na základe predchádzajúcich algoritmov by sa dal problém riešiť dvojnásobným aplikovaním algoritmu z predchádzajúceho príkladu, pričom je v druhom hľadaní nutné zabezpečiť vylúčenie nájdenia toho istého prvku. Ukážeme riešenie pomocou jedného prechodu postupnosťou. Budeme vychádzať z predpokladu (hypotéza), že najmenšie sú prvé dva prvky (priradenie ich hodnôt premenným MIN1, MIN2 sa uskutoční na základe ich porovnania tak, aby platilo MIN1 ≤ MIN2, rovnako aj neskôr). Potom budeme postupne prezerať ostatné prvky postupnosti (korekcia hypotézy), pričom pre prvok a_i, kde i =3, ...,N môžu nastať tri prípady:

1) a_i ≤ MIN1 ≤ MIN2, vtedy sa "posunie" hodnota z MIN1 do MIN2 a do MIN1 sa priradí hodnota a_i
2) MIN1 < a_i < MIN2, v tomto prípade treba do MIN2 uložiť hodnotu a_i
3) MIN1 ≤ MIN2 ≤ a_i, vtedy netreba robiť nič.

```
ALG Hľadanie_dvoch_ najmenších prvkov;
Prem   i, N          : celočíselné;
       MIN1, MIN2    : reálne;
       A             : pole[1 .. N] reálnych:
začiatok
výstup(“Zadaj počet prvkov postupnosti A : “);
vstup( N );                                 {načítanie postupnosti A}
pre i := 1 po N
   rob   výstup(“Zadaj prvok A[“, i, “]: “);
         vstup(A[ i ])
*pre;
ak A[1] ≤ A[2]                              {počiatočné priradenie}
   tak   MIN1 := A[1];                      {hodnôt premenným MINI, MIN2}
         MIN2 := A[2]
```

```
    inak  MIN2 := A[1];
          MIN1 := A[2]
*ak;
pre i := 3 po N                       {porovnanie s nasledujúcimi prvkami postupn.}
    rob ak A[ i ] ≤ MIN1              {prvý prípad}
        tak     MIN2 := MIN1;
                MIN1 := A[ i ]
        inak  ak A[ i ] < MIN2        {druhý prípad}
              tak MIN2 := A[ i ]
              *ak
        *ak
*pre ;
výstup("Dva najmenšie prvky postupnosti sú: ", MIN1, ", ", MIN2)
koniec.
```

Cvičenie.

1. Zostavte algoritmus, ktorý nájde 2 najväčšie prvky postupnosti A= {a_1, ..., a_n}.

3.1 Neriešené príklady

1. Zostavte algoritmus na prevod celého desiatkového čísla do binárnej sústavy.
2. Zostavte algoritmus na prevod celého desiatkového čísla do sústavy o základe Z, kde $2 \le Z \le 20$.
3. Zostavte algoritmus, ktorý pre dané prirodzené číslo zistí, či je deliteľné tromi.
4. Zostavte algoritmus na modifikáciu vektora X obrátením poradia svojich vlastných hodnôt.
5. Zostavte algoritmus na vylúčenie ekvivalentného alebo "subdominantného" riadku matice A. (Napr. k riadku s hodnotami -2,3,7,0,0 je subdominantý riadok 0,3,7,0,0 a1ebo -2,3,0,0,0 alebo 0,0,7,0,0 atď.).
6. Zostavte algoritmus na vylúčenie ekvivalentného alebo dominantného riadku matice.
7. Zostavte algoritmus na redukciu vektora V. Redukovaný vektor nebude obsahovať žiadny prvok s hodnotou C.
8. Zostavte algoritmus na nahradenie každého záporného prvku vektora V nulou.
9. Navrhnite algoritmus na vytvorenie všetkých kombinácií 3. triedy z čísel 1, 2, 3, 4, 5, 6, 7, 8, 9.
10. V priestore je N bodov. Zostavte algoritmus na vyhľadanie dvoch bodov s najmenšou vzdialenosťou.
11. V priestore je N bodov. Zostavte algoritmus na vyhľadanie bodu s najmenšou vzdialenosťou od počiatku súradného systému.
12. V priestore je N bodov. Zostavte algoritmus na vyhľadanie dvoch bodov s najmenším priemetom vektora, ktorý definujú, do roviny XY (XZ, YZ).
13. V priestore je N bodov. Zostavte algoritmus na vyhľadanie dvoch bodov s najväčším priemetom vektora, ktorý definujú, do roviny XY (XZ, YZ).
14. Navrhnite algoritmus na určenie výhry v súťaži, kde pre 1. cenu treba uhádnuť 6 čísel zo 45, pre 2. cenu 5 čísel, atď. Piata cena sa už neudeľuje.

Prepis algoritmu do programovacieho jazyka Pascal.

Na riešenie rôznych problémov sa v súčasnosti používa mnoho programovacích jazykov. Jedným z najrozšírenejších je programovací jazyk PASCAL. Vzhľadom na to, že jeho syntax a sémantika je podobná algoritmickému metajazyku, použijeme ho na overenie správnosti algoritmov na počítači.

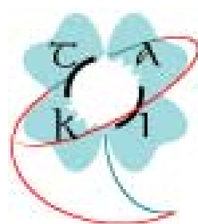
Príklad 1.

Zostavte program v programovacom jazyku Pascal, ktorý presunie najmenší prvok postupnosti celých čísel A= $\{a_1, a_2, ..., a_n\}$ na jej začiatok.

Riešenie.

Pri riešení problému budeme vychádzať z príkladov 15 a 16. Podľa riešenia príkladu 16 vieme nájsť minimum postupnosti. Tento algoritmus rozšírime o príkazy, pomocou ktorých zistíme pozíciu (premenná POZ) minimálneho prvku v postupnosti. Dostaneme:

```
výstup("Zadaj počet prvkov postupnosti : ");
vstup( N );                                   {načítanie postupnosti}
pre i := 1 po N
   rob   výstup("Zadaj prvok A[", i, "]: ");
         vstup(A[ i ])
*pre;
MIN := A[1];                                  {hľadanie minima}
POZ := 1;
pre i:=2 po N
   rob  ak A[ i ] < MIN
          tak   MIN := A[ i ];
                POZ: =i
          *ak
*pre;
výstup("Minimum postupnosti = ", MIN)         {výpis minima}
```



Ak chceme presunúť prvok, ktorý sa nachádza na určitej pozícii POZ, treba najskôr posunúť všetky prvky od začiatku až po prvok, ktorý sa nachádza na pozícii POZ –1 o jeden prvok vpravo a potom na začiatok zapísať prvok z pozície POZ (v našom prípade je to nájdený minimálny prvok). Teda možno aplikovať algoritmus z príkladu 15 na časť postupnosti $\{a_1, a_2, ..., a_{POZ}\}$. Upozorňujeme na skutočnosť, že presúvaná hodnota a_{POZ} je už uložená v pramennej MIN, t.j. na jej uchovanie netreba ďalšiu premennú POMOC, a preto z algoritmu možno vynechať príkaz

POMOC := A[N]; ktorý by mal byť nahradený príkazom POMOC := A[POZ].

Dostávame:

```
pre j := POZ klesaním po 2
rob A[ j ] := A[ j –1]
*pre;
A[1] := MIN;
```

Celý algoritmus potom bude vyzerať nasledovne:

```
ALG Hľadanie_minima;
Prem   i, N, POZ, MIN   : celočíselné;
       A                : pole[1 .. N] celočiselných;
začiatok
výstup("Zadaj počet prvkov postupnosti: ");
vstup( N );
```

```
pre i := 1 po N
    rob  výstup(“Zadaj prvok A[“, i, “]: “);
           vstup(A[ i ])
*pre;
MIN := A[1];
POZ := 1;
pre  i := 2 po N
    rob ak A[ i ] < MIN
          tak MIN := A[ i ];
                POZ := i
          *ak
* pre;
výstup(“Minimum postupnosti = “, MIN);
pre  j := POZ klesaním po 2
    rob A[ j ] := A[ j–1]
*pre;
A[1] := MIN;
koniec .
```

Použitím prekladovej tabuľky uvedenej na konci tejto kapitoly prepíšeme algoritmus do programovacieho jazyka PASCAL, pričom miesto deklarácie rozmeru poľa A (1 **..**N) zapíšeme predpokladaný maximálny rozmer (napr. 1**..**10).

```
Program Hľadanie_minima;
Var   i, N ,POZ ,MIN : integer;
        A:                   : array [1..10] of  integer;
begin
write( ' Zadaj počet prvkov postupnosti:  ' );              {Načítanie}
readln( N );                                                {postupnosti}
for i := 1 to N do
    begin      write( ' Zadaj prvok A[ ' , i, ' ]:  ' );
                readln(A[ i ] )
    end;
MIN := A[1];                                                {Hľadanie minima}
POZ := 1;
for  i := 2 to N do
    if  A[ i ] < MIN
        then  begin   MIN := A[ i ];
                        POZ := i
                end;
writeln ( ' Minimum postupnosti  =  ' , MIN);
for j := POZ downto 2 do                                    {Presun minima}
    A[ j ] := A[ j –1]                                      {na začiatok}
    A[1] := MIN
end .
```

3.2 Prekladová tabuľka Pascal

z algoritmického metajazyka do programovacieho jazyka PASCAL

NÁZOV objektu – zápis v metajazyku	Preklad do jazyka PASCAL
1	2
OPERÁTORY **algebraické:** + – * / **^2** – (druhá mocnina) **div** (celočíselný podiel) **mod** (zvyšok po delení)	 + – * / sqr(<prepis výrazu>) div mod
relačné: < , > , = ≤ , ≥ , ≠	 < , > , = <= , >= , < >
logické: ∩ –a súčasne, ∪ – alebo, ¬ – negácia **nebude** (v cykle za **pokiaľ**)	 and, or, not not
Základné aritmetické **FUNKCIE**: $\sqrt{}$ (druhá odmocnina) |v| (absolútna hodnota) sin, cos, tg, cotg(v) e^v ln(v)	 sqrt(<prepis výrazu>) abs(<prepis výrazu>) sin, cos, tan, 1/tan(<prepis výrazu>) exp(<prepis v>) ln(v)
Príkaz PRIRADENIA premenná **:=** výraz ; premenná ← výraz ;	 premenná := <prepis výrazu> ;
Príkaz VSTUPU Vstup(vstupný zoznam); {čítaj(vstupný_zoznam);}	write('Zadaj ... : '); read(vstupný zoznam); {readln(vstupný_zoznam);}
Príkaz VÝSTUPU výstup(výstupný_zoznam); {píš(výstupný_zoznam);}	 write (Výstupný zoznam); {writeln(Výstupný_zoznam);}
SEKVENCIA (príkaz_1; ... **;** príkaz_n;)	 begin <preklad príkaz_1>; ... ; <preklad príkaz_n>; end
Podmienený príkaz AK úplný: **ak** podmienka **tak** príkaz_1 **inak** príkaz_2 ***ak ;**	 if <prepis podmienky> then {begin} <preklad príkaz_1> {end} else {begin} <preklad príkaz_1> {end} {endif};
neúplný: **ak** podmienka **tak** príkaz ***ak;**	 if <prepis podmienky> then {begin} <preklad príkazu> {end} {endif};

1	2
Príkaz cyklu: OPAKUJ (úplný): **opakuj** príkaz_1 **pokiaľ** podmienka **rob** príkaz_2 ***opakuj;**	<preklad príkaz_1>; while <prepis podmienky> do begin <preklad príkaz_2>; <preklad príkaz_1>; end {endwhile};
s podmienkou na začiatku: **opakuj** **pokiaľ** podmienka **rob** príkaz ***opakuj;**	while <prepis podmienky> do {begin} <preklad príkazu> {end} {endwhile};
s podmienkou na konci: **opakuj** príkaz **pokiaľ** podmienka ***opakuj;**	repeat <preklad príkazu> until not <preklad podmienky> {endrepeat};
Cyklus so známym počtom opakovaní: **PRE** stúpanie (s krokom +1): **pre** op := v_1 **po** v_2 **rob** príkaz1 ***pre;**	for op:=<prepis v_1> to <prepis v_2> do {begin} <preklad príkaz1> {end} {endfor};
klesanie (s krokom –1): **pre** op := v_1 **klesaním po** v_2 **rob** príkaz ***pre;**	for op:=<prepis v_1> downto <prepis v_2> do {begin} <preklad príkazu> {end} {endfor};
Typy údajov: celočíselné reálne logické znaky pole ... reálnych	integer real boolean char array ... of real
Ďalšie vyhradené slová: **Alg** **prem** **začiatok** **koniec**	Program var begin end

Použité symboly:

{...} – poznámka, upozornenie na nutnosť výskytu pri sekvenciách, ak tieto nahrádzajú príkaz v iných štruktúrovaných príkazoch
<...> – nutnosť naplnenia konkrétnym obsahom
podmienka – logický výraz
v – aritmetický výraz
op – premenná ordinálneho typu
v_1, v_2 – výrazy typu zhodného s typom premennej op.

 ISBN 80–228–0428–2

3.3 Prekladová tabuľka C++

z algoritmického metajazyka do programovacieho jazyka C++

<table>
<tr><td>NÁZOV objektu – zápis v metajazyku</td><td>Preklad do jazyka C++</td></tr>
<tr><td>1</td><td>2</td></tr>
<tr><td>OPERÁTORY
algebraické:
+ – * /
^2 – (druhá mocnina)
<u>div</u> (celočíselný podiel)
<u>mod</u> (zvyšok po delení)</td><td>

+ – * /
(<prepis výrazu>)*(<prepis výrazu>)
/
%</td></tr>
<tr><td>relačné:
< , > , =
≤ , ≥ , ≠</td><td>
< , > , = =
<= , >= , !=</td></tr>
<tr><td>logické:
∩ –a súčasne, ∪ – alebo, ¬ – negácia</td><td>
&& , || , !</td></tr>
<tr><td>Základné aritmetické FUNKCIE:
√ (druhá odmocnina)
|v| (absolútna hodnota)
sin, cos, tg,
cotg(v)
e^v
ln(v)</td><td>
sqrt(<prepis výrazu>)
abs(<prepis výrazu>)
sin, cos, tan,
1/tan(<prepis výrazu>)
exp(<prepis v>)
ln(v)</td></tr>
<tr><td>Príkaz PRIRADENIA
premenná := výraz ;
premenná ← výraz ;</td><td>
premenná = <prepis výrazu> ;</td></tr>
<tr><td>Príkaz VSTUPU
Vstup(vstupný zoznam);
{čítaj(vstupný_zoznam);}</td><td>
<<</td></tr>
<tr><td>Príkaz VÝSTUPU
výstup(výstupný_zoznam);
{píš(výstupný_zoznam);}</td><td>
>></td></tr>
<tr><td>SEKVENCIA
(
príkaz_1;
... ;
príkaz_n;
)</td><td>
{
<preklad príkaz_1>;
... ;
<preklad príkaz_n>;
}</td></tr>
<tr><td>Podmienený príkaz <u>AK</u>
úplný:
<u>ak</u> podmienka
<u>tak</u>
príkaz_1
<u>inak</u>
príkaz_2
<u>*ak ;</u></td><td>

if (<prepis podmienky>)

<preklad príkaz_1>;
else
<preklad príkaz_2>;
/*end if */</td></tr>
<tr><td>neúplný:
<u>ak</u> podmienka
<u>tak</u>
príkaz
<u>*ak;</u></td><td>
if (<prepis podmienky>)

<preklad príkazu>;
/*end if */</td></tr>
</table>

1	2

Príkaz cyklu: OPAKUJ
(úplný):
opakuj
príkaz_1
pokiaľ podmienka
rob
príkaz_2
*opakuj;
	<preklad príkaz_1>;
	while (<prepis podmienky>)
	{
	<preklad príkaz_2>;
	<preklad príkaz_1>
	};
	/*end while*/

s podmienkou na začiatku: **opakuj** **pokiaľ** podmienka **rob** príkaz *opakuj;	while (<prepis podmienky>) <preklad príkaz>; /*end while*/
s podmienkou na konci: **opakuj** príkaz **pokiaľ** podmienka *opakuj;	do <preklad príkazu> while (<preklad podmienky>); /*end do*/
Cyklus so známym počtom opakovaní: **PRE** stúpanie (s krokom +1): **pre** op := v_1 **po** v_2 **rob** príkaz1 *pre;	for (incializácia; koncový_výraz; spôsob_iterácie) <preklad príkazu1>; /*end for*/
klesanie (s krokom –1): **pre** op := v_1 **klesaním po** v_2 **rob** príkaz *pre;	for (incializácia; koncový_výraz; spôsob_iterácie) <preklad príkazu>; /*end for*/
Typy údajov: celočíselné k reálne x logické t znak z pole A[0..20] reálnych	 int k float x *resp*. double x bool t char z float A[21]

Ďalšie vyhradené slová:
Alg Názov;
prem
začiatok
koniec
	
	void Názov ( );
	/*var*/
	{
	}

Použité symboly:

<...>	–	nutnosť naplnenia konkrétnym obsahom
podmienka	–	logický výraz
v	–	aritmetický výraz
op	–	premenná ordinálneho typu
v_1, v_2	–	výrazy typu zhodného s typom premennej op.

4 Literatúra

1. HVORECKÝ, J.– KELEMEN, J.: Algoritmizácia, Bratislava, Alfa,1987.
2. PIVARČIOVÁ, E. – ŠIPOŠ, Ľ.: Programovacie techniky, Zvolen, vydavateľstvo TU, 2004
3. Turbo Pascal Version 6.0- User's Guide, Borland International, USA, 1990.
4. WIRTH, N.: Algoritmy a štruktúry údajov, Bratislava, Alfa, 1988.

Táto príručka, ani akákoľvek jej časť nesmie byť kopírovaná a ani inak rozširovaná bez súhlasu autorov. Informácie, návody a príklady uvedené v príručke sú chránené autorským zákonom

ISBN 80–228–0428–2