Kapitola 1

Problematika algoritmizace

Řada problémů, které nás obklopují a které čas od času jsme nuceni řešit, je s výhodou řešitelná s využitím prostředků výpočetní techniky. Jednoduché i složité vědecko-technické výpočty, problematiku hromadného zpracování dat, projekční a konstrukční práce a řadu dalších úkolů již dlouho s námi řeší počítače. S příchodem pro širokou veřejnost dostupné, rozměry malé, avšak výkonné výpočetní techniky se využívání *osobních počítačů* stává součástí gramotnosti každého z nás. Mnozí lidé nacházejí v osobním počítači pomocníka ve svém oboru a využívají jeho schopností nejen při své profesionální činnosti, stejně běžně, jako využívají pračku, automobil apod.

Ambice absolventa vysoké školy by měly být pochopitelně vyšší. Jeho cílem by nemělo být počítač pouze využívat, stát se jeho pasívním uživatelem, ale i tvůrčím způsobem počítač ovládnout tak, aby se mu stal pomocníkem i při řešení *specifických úkolů inženýrské praxe*. Pod pojmem *specifický úkol* si nepředstavujme žádný zvláštní problém. Může se jím stát např. problematika řešení kvadratické rovnice v okamžiku, kdy je třeba vyřešit větší počet na prvý pohled jednoduchých kvadratických rovnic. Nemáme-li, popř. nevíme-li o žádném programu na řešení této problematiky (nebo systému, v rámci kterého je tento problém řešen), nezbývá nám, než program vlastní hlavou vyrobit.

Zásadní záležitostí při tvorbě programu je *algoritmizace problému*, tj. vytváření postupu řešení daného problému na počítači. Algoritmizace je krásná tvůrčí činnost, při které využíváme intelekt, zkušenosti, intuici a postupně vytváříme spoustu postupů řešení, z nichž ty počáteční zpravidla k cíli nevedou vůbec, další vedou k cíli pouze občas a havárie při řešení nastává již jen v určitých zvláštních (mezních) situacích, o kterých jsme na počátku našeho snažení vůbec neuvažovali (např. řešení kvadratické rovnice v oboru komplexních čísel). Postupně však vytváříme stále *lepší* algoritmus (rychlejší, relativně kratší) o kterém se snad již bude za nějaký čas hovořit jako o algoritmu optimálním[1].

Vytvořený postup řešení problému na počítači je dílem *normálního člověka* a neskrývá v sobě žádné nepřirozené, ba ani nadpřirozené akce, jak se mnoho lidí z neznalosti v úctě k počítačům domnívá. Počítač je sice složitý stroj, avšak vše, co s ním souvisí, vše co umí, vytvořil člověk „k obrazu svému“. Tedy i algoritmy mají v sobě pouze prvky způsobů komunikace člověka s člověkem, přičemž počítač se jeví jako „člověk ducha mdlého až nevýrazného“. Zdaleka tedy nerozumí všemu; rozumí *pouze* tomu, co ho lidé naučili. Pochopí-li však počítač, co mu přikazujeme, tj. domluvíme-li se s ním, pak naše příkazy vykoná daleko rychleji, přesněji a důsledněji, než by to učinil sebedokonalejší člověk.

1.1 Základní pojmy

Algoritmus je postup řešení problému. Každý se již setkal se spoustou algoritmů, ba i mnohé algoritmy sám vyřešil. Problém např. dostat se v Brně na Zemědělskou ulici č. 5 k přijímacímu řízení na PEF MZLU v Brně různí jedinci mohli řešit různě: zakoupením mapy města Brna, dotazem náhodného kolemjdoucího, bloumáním po městě a mnoha dalšími způsoby. Sbírkou řešených algoritmů je jistě i kuchařská kniha, stejně jako množství příruček řady „U“dělej si sám.

Společné pro téměř všechny nám dosud známé algoritmy je skutečnost, že jak jejich řešitelem, tak i jejich uživatelem (aplikátorem, interpretem) je člověk – bytost schopná vlastního uvažování. Řešitel algoritmu si může v této situaci dovolit používat i takové *příkazy*, jakými jsou celkem nejednoznačné, časově blíže nespecifikované příkazy typu

- Přišroubuj spodní část krytu!

[1] Optimální algoritmus je něco jako nadpřirozená bytost. Také se o ní hovoří, mnozí věří v její existenci a jiní zase tvrdí, že neexistuje. Důkazy o její existenci však nejsou věrohodné.

- Pomocí Newtonova vzorce vypočtěte odmocninu ze 6!
- Těsto osolíme popř. osladíme.

Přitom s ohledem na inteligenci člověka nepovažujeme za potřebné mu přikazovat, že

- Šroubování provádíme šroubovákem tak, že jej uchopíme do ruky (praváci do pravé a leváci do levé), nasadíme do drážky, kterou hledáme na hlavičce šroubu a ...
- Hodnotu $\sqrt{6}$ máme počítat čtyři a půl minuty, maximálně však na šest desetinných míst ...
- Hodláme-li pečivo podávat ráno k bílé kávě, pak těsto osladíme, bude-li sloužit jako příloha večer k vínu, pak těsto osolíme.

Předmětem našeho dalšího snažení je ovšem tvorba algoritmů, jejichž vykonavatelem bude stroj - *počítač*. Algoritmy pro počítač (dále jen algoritmy) tedy budou muset splňovat několik základních podmínek. Budou muset být

- deterministické, tj. v každém okamžiku jednoznačné,
- rezultativní, tj. konečné, vedoucí k cíli (k výsledkům, třeba i chybným),
- obecné (hromadné), tj. řešením úlohy pro libovolná vstupní data (zohledňující možné alternativy v postupu řešení daného problému),
- opakovatelné, tj. na základě stejných vstupních hodnot poskytovat stejné výsledky
- srozumitelné, přehledné a tudíž modifikovatelné, což zaručí jejich další rozšiřování, upravování, vylepšování.

Nejen s poslední požadovanou vlastností algoritmu souvisí otázka *způsobu zápisu algoritmu.* Dostupné počítače neumí zatím naslouchat našim mluveným příkazům, ba ani jim nerozumí, jsou-li uvedeny v nám známé písemné formě. *Zpracování algoritmu počítačem vyžaduje zápis algoritmu v programovacím jazyku.* Programovací jazyk je jazyk umělý a je zpravidla velmi zredukovanou podmnožinou přirozeného anglického jazyka. Dnes existují desítky běžně používaných programovacích jazyků jako např. Ada, Algol, Basic, C–jazyk, Cobol, Fortran, Modula, Pascal, PL–1, Prolog, RPG, Simula67 a řada dalších, více či méně speciálních jazyků. My budeme dále používat k zápisu algoritmů prostředků, kterými disponuje *programovací jazyk Pascal*, budeme tedy programovat v Pascalu.
Program, jak již čtenář asi tuší, je v podstatě zápis algoritmu v programovacím jazyku. Definici *programu* plně vystihuje název knihy [1] prof. Wirtha[2]

$$\text{Program} \;=\; \text{Datové struktury} \;+\; \text{Algoritmus} \tag{1.1}$$

Program, jenž hodláme spustit, opět nepostačuje mít na papíře, ba ani nepostačuje jej mít na disketě. Ze základů operačního systému již víme, že příkaz spuštění spustitelného souboru zajistí jeho zavedení (natažení) do operační (vnitřní) paměti počítače a poté se začne počítač řídit jednotlivými příkazy, jež jsou součástí programu; začne příkazy vykonávat (interpretovat). Algoritmus, jenž můžeme považovat právě za příkazovou část programu, zpravidla pracuje s nějakými hodnotami (např. číselnými, nebo řetězcovými aj.), které je třeba mít opět v době, kdy se s nimi pracuje v operační paměti. Tam k jejich umístění slouží *datové struktury* (synonyma ke slovu datové struktury mohou být: datové objekty, buňky, chlívečky, domky, *proměnné*), které jsou pojmenovány a mají různou velikost dle toho, jaká hodnota má být v nich umístěna (celé číslo potřebuje obecně méně místa než číslo desetinné, proměnná pro umístění příjmení může být asi menší než proměnná pro umístění celé adresy apod.). Výstavba datových struktur z jednotlivých paměťových elementů – *bytů* (čti bajtů) a jejich pojmenování (identifikace, přiřazení adresy datové struktuře), se provádí na začátku programu v části definicí a deklarací. Výše uvedený vztah (1.1) je tedy možno uvádět i ve tvaru:

$$\text{Program} \;=\; \text{Část definicí a deklarací} + \text{Část příkazová} \tag{1.2}$$

1.2 Slovníček

Než se ponoříme do problematiky algoritmizace na bázi strukturovaného jazyka Pascalu, je vhodné se naučit několik anglických slovíček.

[2] Prof. Niklaus Wirth z technické vysoké školy v Zurichu je mj. i autorem programovacího jazyka Pascal [2]

Klíčová[3] a některá rezervovaná slova jazyka Pascalu

anglicky	zkratka	česky	poznámka
and		a	logický součin
array		pole	
begin		začátek	
Boolean		Boleovský	též jako logický
case		případ	jedna z možností
character	char	znak	
constant	**const**	konstanta	
divide	**div**	dělit	operátor celočíselného dělení
do		dělej	
downto		dolů k	sestupně až po
else		jinak	v jiném případě
end		konec	
file		soubor	
for		pro	
function		funkce	
go to		jdi k (návěští)	skoč
if		jestliže	
in		v	předložka s 6.pádem
integer		celé číslo	
label		návěští	
modulo	**mod**	modulo	zbytek po celočíselném dělení
nil		nikam	prázdná hodnota ukazatele
not		ne	logický operátor negace
of		z	předložka s 2. pádem
or		nebo	logický součet
packed		sbalený	zhuštěné (úsporné) uložení strukturované proměnné
procedure		procedura	
program		program	
read		čti	
read line	readln	čti a přejdi na nový řádek	
real		reálné číslo	
record		záznam	
repeat		opakuj	
set		množina	
string		řetězec	
then		pak	je – li splněna podmínka
to		až po	vzestupně
type		typ	
until		dokud není	dokud neplatí
uses		užij	
variable	**var**	proměnná	
while		dokud	dokud platí
with		s	předložka s 7. pádem
write		piš	
write line	writeln	piš a přejdi na nový řádek	

[3] Klíčová slova se obvykle uvádějí v publikacích tučně, případně jsou podtržena pro zvýšení přehlednosti (čitelnosti) a snazší orientaci v programu.

1.3 Prostředky pro zápis algoritmu

Očekává – li čtenář nyní podstatnou část učebního textu z hlediska kvantity, bude zklamán. Prostředků (příkazů) pro zápis algoritmu není mnoho. Jedná se celkem o čtyři *jednoduché příkazy*, z nichž jeden nebudeme ani používat, a osm složitějších, tzv. *strukturovaných příkazů*, které naopak budeme využívat všechny, i když by nám postačovaly pouze tři z nich.

Pro zápis algoritmu je tedy k dispozici omezená množina vyjadřovacích prostředků - příkazů, které tvoří základ umělého jazyka Pascalu. Navíc v Pascalu na rozdíl od přirozeného jazyka (např. češtiny) platí, že co není výslovně povoleno, je zakázáno. Každý z příkazů Pascalu má svoji jednoznačnou syntaktickou definici (skladbu) a jednoznačnou sémantickou definici (význam). Je tedy třeba se naučit pouze 12 (slovy dvanáct) příkazů, jak z hlediska jejich významu (sémantiky), tak z hlediska způsobu jejich zápisu (syntaxe).[4]

Příkazem v Pascalu je

- příkaz jednoduchý	- příkaz přiřazovací	
	- příkaz procedury	
	- příkaz prázdný	
	- příkaz skoku **goto**	
- příkaz strukturovaný	- příkaz složený	
	- příkaz podmíněný	- úplný **if - then - else**
		- neúplný **if - then**
	- příkaz selektivní	
	- příkaz cyklu	**- for - to - do** resp. **- for - downto - do**
		- repeat - until
		- while - do
	- příkaz **with**	

1.3.1 Příkaz jednoduchý

Jak již název napovídá, jednoduchý příkaz bude zřejmě použitelný k vykonání nějakého jednoduchého úkonu. Za jednoduchý úkon můžeme v této fázi považovat např. *definici obsahu datové struktury*, tj. zaslání nějaké konkrétní hodnoty do proměnné. K definici obsahu proměnné můžeme použít jeden ze dvou příkazů :

- příkaz přiřazovací
- příkaz procedury read resp. readln

Chceme-li např. do *proměné označované identifikátorem* [5] *KAREL* (běžně používáme krátce „do proměnné KAREL“) poslat hodnotu $\frac{3-8}{2}$, můžeme tak učinit pomocí přiřazovacího příkazu

$$KAREL := (3-8)/2 \tag{P}$$

jenž způsobí, že proměnná KAREL bude definována hodnotou -2.5 (všimněme si, že píšeme desetinnou tečku), tj. v proměnné KAREL bude -2.5

KAREL | –2.5 |

Téhož efektu jsme mohli docílit použitím příkazu

read (KAREL) resp. readln (KAREL)

Při vykonávání tohoto příkazu počítač čeká, až pomalý[6] uživatel napíše na dohodnutém vstupním zařízení (zpravidla klávesnici) vkládanou hodnotu, tj. v našem případě až zmáčkne klávesy [–] [2] [.] [5] a tuto hodnotu reprezentovanou čtyřmi znaky *odešle počítači.*

Poznamenejme, že přiřazovací příkaz (P) můžeme psát v obecném tvaru

$$KAREL := (A+B)/C \tag{P'}$$

a objasněme si nyní způsob zpracování přiřazovacího příkazu počítačem.

[4] Na tomto místě opět upozorňujeme, že vzhledem k zaměření a rozsahu tohoto učebního textu je třeba k dokonalému osvojení si především syntaxe 12 pascalovských příkazů čerpat z učebního textu [3] (ke koupi na FAST), popř. [5], [6] (v prodejnách VUT), [4] (v prodejně VŠZ) či z celostátních učebnic [7] nebo [11].

[5] Identifikátor je posloupnost tvořená z písmen a číslic začínající písmenem (zjednodušená definice).

[6] Zatímco uživatel vykoná operaci řádově v sekundách, procesor pracuje s rychlostí cca 10^6 operací/s

Přiřazovací příkaz sestává ze tří částí : levé, střední a pravé. Pravou stranu přiřazovacího příkazu tvoří tzv. *výraz*. Ten se může skládat z *konstant, proměnných, zápisu funkcí* (např. sin(X), odd(B), upcase(Z) aj.), jež jsou vzájemně pospojovány *operátory*. K vyjádření priority dílčích částí výrazu používáme známých kulatých závorek. Výrazy mohou mít tvar např.:

$$(-B + sqrt(B * B - 4 * A * C))/(2 * A) \quad \text{(V1)}$$

$$(A > 0)\mathbf{or}(B > 0) \quad \text{(V2)}$$

$$Z1 +' H' + UpCase(Z2) \quad \text{(V3)}$$

a podle použitých komponent ve výrazu hovoříme o aritmetickém (číselném) výrazu (V1), o logickém výrazu (V2) či o řetězcovém výrazu (V3).

Střední část přiřazovacího příkazu tvoří operátor přiřazení **:=**. Jedná se o dva znaky tvořící jeden celek, které čteme jako „přiřaď".

Levou stranu přiřazovacího příkazu tvoří identifikátor proměnné (nebo její složky), do které bude hodnota získaná vyčíslením výrazu uložena.

Je zřejmé, že při zpracování přiřazovacího příkazu počítačem musí být jednoznačně definovány obsahy všech proměnných, které se podílejí na formulaci výrazu. Tak zřejmě náš příkaz (P') musí nutně předcházet příkazy definující obsahy proměnných A, B a C.

Zřejmě se nebude jednat o příkazy A:=3; B:=-8; C:=2 (povšimněte si: mezi dvěma příkazy píšeme středník; jednotlivé příkazy vzájemně oddělujeme středníkem), ale o obecný příkaz

read(A,B,C)

na který já odpovím svojí trojicí hodnot [3][ ][-][8][ ][2] (povšimněte si: mezi jednotlivými číselnými hodnotami děláme mezeru (alespoň jednu); oddělovačem číselných hodnot na vstupu je mezera), jiný uživatel odpoví svojí trojicí hodnot a další uživatel zase odpoví jinou trojicí hodnot.

Výzva k zadání svých koeficientů, jejich načtení, výpočet výsledku a jeho následné vytištění řeší algoritmus ve tvaru :

```
begin  write('Zadej tři koeficienty : ');
       readln(A,B,C);
       KAREL:=(A+B)/C;                                  (A)
       writeln('Výsledek je ', KAREL)
end.
```

Ve výše uvedeném algoritmu (A) jsme použili čtyři jednoduché příkazy: tři příkazy procedury a jeden příkaz přiřazovací.

Příkaz procedury je příkaz, kterým voláme podprogram - proceduru. Proceduru si můžeme udělat (deklarovat) sami, nebo můžeme využívat procedury *standardní*, které někdo udělal (*deklaroval*) a nám je poskytl v rámci dané implementace jazyka. V našem stádiu poznání zatím postačí, když vezmeme na vědomí, že jedněmi z mnoha standardních procedur Pascalu jsou procedury zajišťující vstup - *čtení* - hodnot z periferních vstupních zařízení počítače do jeho operační paměti a procedury zajišťující výstup - *psaní* - hodnot z operační paměti počítače na jeho výstupní periferní zařízení.

Dalším jednoduchým příkazem, který je spojen s úkonem potřebným k přerušení přirozeného, tj. postupného (sekvenčního) vykonávání jednotlivých příkazů algoritmu na nějaký jiný než následující příkaz, je

příkaz skoku **goto**. Nutno zde podotknout, že postačuje několik málo příkazů skoku a celý algoritmus se stává nepřehledným. Proto se použití tohoto příkazu vždy vyvarujeme (snad je omluvitelný v některých specifických příkladech odskoku na konec programu či podprogramu); bude-li se přesto zdát použití skoku nezbytným, nechť student laskavě konzultuje problém se svým cvičícím, přednášejícím či autorem předloženého učebního textu.

Posledním, čtvrtým jednoduchým příkazem je

příkaz prázdný. Jeho význam pochopíme plně až z následující podkapitoly. Zvídavým však napovíme již nyní :

Ve výše uvedeném algoritmu (A) je možno za čtvrtým příkazem, tj. příkazem procedury writeln napsat středník, tj. řádek bude mít tvar

writeln('Výsledek je ', KAREL);

V tom případě jsme nic nepokazili, ovšem algoritmus se nyní skládá z pěti příkazů: 3 příkazů procedury, jednoho přiřazovacího příkazu a jednoho prázdného příkazu. Proč tomu tak je, si čtenář jistě uvědomí prostudováním syntaktické definice *příkazové části programu*, která je uvedena v literatuře [3], resp. [4], [5], [6] nebo [7].

1.3.2 Příkaz strukturovaný

Název *strukturovaný příkaz* vystihuje skutečnost, že se jedná o strukturu (složitější konstrukci), v jejíž konečných částech (listech) nacházíme jednoduché příkazy. Základní struktura (strom) může být dále strukturována (větvena); na konci je však vždy jednoduchý příkaz (list).

Podmíněný příkaz je struktura ve tvaru[7]

if	podmínka	**then**	příkaz
		else	příkaz

Používá se v okamžiku, kdy nastává v algoritmu potřeba větvení dalšího postupu, přičemž **splnění** resp. **nesplnění** *podmínky* vede k provádění příkazu ve větvi **then** resp. **else**.

Požadujeme-li např. tisk informace o tom, zda v proměnné CISLO je hodnota *kladná, záporná* nebo *nulová*, použijeme podmíněný příkaz ve tvaru

if CISLO > 0 **then** write('JE KLADNÉ')
else if CISLO < 0 **then** write('JE ZÁPORNÉ')
else write('JE TO NULA')

Podmíněný příkaz je možno použít v jeho *úplné* popř. *neúplné* podobě.

Neúplný podmíněný příkaz nemá větev **else** a jeho struktura má tvar

if	podmínka	**then**	příkaz

Příklad: Máme tři proměnné CISLO, NEJMENSI a DRUHENEJ, v nichž jsou nějaké číselné hodnoty. Úkolem je zatřídit hodnotu CISLO, tj.:

if CISLO < NEJMENSI **then begin** DRUHENEJ:=NEJMENSI;
NEJMENSI:=CISLO
end
else **if** CISLO < DRUHENEJ **then** DRUHENEJ:=CISLO

Řešením tedy je, jak vidíme výše, jeden úplný podmíněný příkaz. Úplný proto, že má jak větev **then**, tak také větev **else**. Ve větvi **then** je příkaz. Jmenuje se *složený příkaz* a ve větvi **else** je také příkaz, který se nazývá *neúplný podmíněný příkaz*. Tento příkaz má pouze větev **then**, ve které je jednoduchý příkaz. Rovněž součástí složeného příkazu jsou již jednoduché příkazy; oba jsou příkazy přiřazovacími.

Složený příkaz, jak si již pozorný čtenář povšiml, používáme v okamžiku, kdy ve struktuře hodláme v místě, kde je syntaktickou definicí povolen příkaz (tj. jeden příkaz) zapsat příkazů několik. Skupina příkazů ohraničená levou příkazovou závorkou **begin** a pravou příkazovou závorkou **end** pak tvoří jediný příkaz – složený příkaz.

Selektivní příkaz **case** je struktura tvaru

case	proměnná[8] **of**		
		možnost1 :	příkaz;
		možnost2 :	příkaz;
		⋮	
		možnostN :	příkaz
end			

[7] Již je třeba znát slovíčka uvedená v odst 1.2.
[8] Podotkněme nyní, že zde nemusí být osamocená proměnná, avšak může zde být uveden výraz ve smyslu dříve uvedeném, avšak v této fázi nechceme situaci zbytečně komplikovat.

Použití selektivního příkazu demonstrujeme na příkladu :

V proměnné Z máme nějakou číslici z intervalu 1..7, kterou reprezentujeme den v týdnu (1 (jedničkou) pondělí, 2 (dvojkou) úterý až 7 (sedmičkou) neděli). Chceme slovně vytisknout, o jaký den se jedná. Řešením, vcelku nasnadě, je použít několik do sebe vnořených úplných podmíněných příkazů, tj.

```
if Z=1 then write ('pondělí')
      else if Z=2 then write('úterý')
            else if Z=3 then write('středa')
                  else if Z=4 then write('čtvrtek')
                        else if Z=5 then write('pátek')
                              else if Z=6 then write('sobota')
                                    else if Z=7 then write('neděle')
                                          else write('něco jiného')
```

Efektivnější, kratší a rychlejší je ovšem použít selektivní příkaz ve tvaru

```
case Z of
   1    : write('pondělí');
   2    : write('úterý');
   3    : write('středa');
   4    : write('čtvrtek');
   5    : write('pátek');
   6    : write('sobota');
   7    : write('neděle');
   else   write('něco jiného');
end
```

což vede také k přehlednějšímu zápisu algoritmu.

Poslední skupinu příkazů v Pascalu, příkazy v pořadí 9, 10 a 11 tvoří tzv. *příkazy cyklu*. Příkaz cyklu používáme tehdy, hodláme-li některou sekvenci příkazů (nebo i jen jeden příkaz) opakovaně provádět.

Příkaz **while - do** má strukturu

while podmínka **do** příkaz

Příkaz uvedený za klíčovým slovem **do** (většinou je tímto příkazem složený příkaz) se provádí opakovaně tak dlouho, dokud *podmínka* uvedená za klíčovým slovem **while** *je splněna*. Nemá-li však dojít k nekonečnému opakování tohoto příkazu, je třeba zajistit, aby opakující se příkaz modifikoval některou z komponent, z nichž je podmínka tvořena. Tak např. v algoritmu

```
S := 0;
I := 1;
while I <= 10 do S := S + I
```

se bude příkaz cyklu provádět do nekonečna, neboť obsah proměnné I je stále stejný (obsah proměnné S se neustále zvyšuje o 1). Požadujeme-li však sečíst pouze $1+2+\ldots+10$ a výsledek tohoto součtu uložit do proměnné S, pak algoritmus musí být ve tvaru:

```
S := 0;
I := 1;
while I <= 10 do begin   S := S + I;
                         I := I + 1
                 end
```

Příkaz **repeat - until** má strukturu

```
repeat
          příkaz;
          ⋮
          příkaz
until     podmínka
```

Příkazy uvedené mezi klíčovými slovy **repeat** a **until** se opakovaně provádějí tak dlouho, dokud *podmínka* formulovaná za klíčovým slovem **until** *není splněna*. Zatímco opakující se příkaz (říká se mu také *tělo cyklu* se u příkazu **while - do** nemusí provést ani jednou, neboť test na podmínku se provádí před případným vykonáním těla cyklu, tak u příkazu **repeat - until** se tělo cyklu alespoň jednou provede. K zabránění vzniku nekonečného cyklu platí to, co jsme v této souvislosti uvedli u příkazu **while - do**. Problém výše řešený s využitím příkazu **while - do** je pochopitelně řešitelný i pomocí příkazu **repeat - until**:

```
S := 0;                                       S := 0;
I := 1 ;                                      I := 0 ;
repeat  S := S + I;  nebo také ve tvaru:      repeat  I := I + 1;
        I := I + 1;                                   S := S + I;
until   I > 10                                until   I >= 10
```

Význam má však *obecné řešení* $SUMA = \sum_{i=A}^{B} I$ pro přírůstek ΔI = C (výše uvedené řešení odpovídalo tedy pro A=1, B=10 a C=1), které s využitím příkazu **repeat - until** můžeme zapsat ve tvaru:

```
S := 0;                                       S := 0;
I := A ;                                      I := A - C ;
repeat  S := S + I;  nebo také ve tvaru:      repeat     I := I + C;
        I := I + C;                                      S := S + I;
until   I > B                                 until      I >= B
```

Sestavit obecný algoritmus tohoto problému s použitím příkazu **while - do** by již měl čtenář zvládnout sám.

Příkaz **for - to - do** má strukturu

for P **to** V **do** příkaz

kde P je příkaz definující počáteční hodnotu proměnné cyklu
V je výraz poskytující konečnou hodnotu, jíž bude proměnná cyklu nabývat
a platí, že počáteční hodnota proměnné cyklu <= konečné hodnotě proměnné cyklu. Pakliže je tomu naopak, tj. počáteční hodnota proměnné cyklu >= konečné hodnotě proměnné cyklu, používáme příkaz ve tvaru

for P **downto** V **do** příkaz

Problém součtu řady hodnot počínaje hodnotou A a konče hodnotou B po kroku (přírůstu) C je řešitelný s využitím příkazů **for - to - do** resp. **for - downto - do** pouze za předpokladu, že hodnoty A, B jsou celočíselné hodnoty (*obecně hodnoty ordinálního typu*) a C=1 resp. C=-1. Pak řešení může mít tvar:

```
S := 0;                           respektive   S := 0;
for I:=A to B do S :=S + I;                    for I:=B downto A do S :=S + I;
```

Příkaz **with** je posledním, dvanáctým, existujícím příkazem v Pascalu. Jedná se o příkaz ve tvaru

with proměnná typu záznam **do** příkaz

jenž se využívá toliko při práci s proměnnými typu záznam a čtenář o jeho existenci nemusí zatím ani vědět.

1.4 Příklad na úvod

Vytvořte algoritmus pro řešení kvadratické rovnice tak, aby poskytoval kořeny kvadratické rovnice, která je např. ve tvaru

$$x^2 - 5x + 6 = 0.$$

Řešení: Jednou ze základních vlastností kladených na algoritmus je jeho masovost. Jistě není žádoucí, aby programátor konstruoval program, který by uměl vyřešit pouze výše uvedenou konkrétní kvadratickou rovnici. Problém bude řešit obecně, bude předpokládat obecný tvar kvadratické rovnice

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

Způsob řešení kořenů kvadratické rovnice je znám z matematiky :

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

a pro naše koeficienty $a = 1$, $b = -5$ a $c = 6$ dostáváme kořeny $x_1 = 3$ a $x_2 = 2$. Způsob určení jednotlivých kořenů bude tvořit i jádro našeho algoritmu. Rovnice

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{1.3}$$

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{1.4}$$

vytváří podklad pro základní příkazy algoritmu. Pak tedy příkazy

$$X1 := (-B + sqrt\,(B * B - 4 * A * C))\,/(2 * A) \tag{P1}$$

$$X2 := (-B - sqrt\,(B * B - 4 * A * C))\,/(2 * A) \tag{P2}$$

budeme považovat za příkazy počítače k výpočtu hodnot x_1 a x_2.
Poznamenejme, že k výpočtu druhé mocniny (např. B^2) vedle použitého zápisu $B * B$ bylo možno použít i standardní funkce *sqr*, tedy zapsat výpočet druhé mocniny B ve tvaru sqr(B).

Příkazy (P1) a (P2) představují pouze vhodný přepis rovnic (1.3) a (1.4), přičemž

*	označujeme operátor násobení
/	označuje operátor dělení
sqrt	je symbolické značení funkce druhá odmocnina (z anglického SQuare RooT)

Je zřejmé, že tak jako my nedokážeme při pohledu na příkazy (P1) a (P2) určit, jaké hodnoty nabudou proměnné (buňky) X1 a X2, nemůže ani počítač poskytnout výsledky, neboť nejsou známy (určeny) konkrétní hodnoty koeficientů A, B a C, *nejsou definovány obsahy proměnných A, B a C.* K výsledku by vedla náhrada koeficientů konkrétními konstantami, např. 1, -5 a 6. V tom případě by se ovšem nemohlo hovořit o algoritmu ve smyslu definice (a účelu) algoritmu, neboť by nebyla splněna podmínka hromadnosti.

Řešení spočívá v tom, že počítači přichystáme konkrétní trojici hodnot (prostřednictvím vstupního zařízení - např. klávesnice) a příkazem *read* počítač donutíme, aby si trojici hodnot přečetl.

Příkaz

$$read(A, B, C); \tag{P0}$$

reprezentuje instrukce, na základě nichž se pomocí vstupního zařízení přečtou ze vstupního média postupně tři číselné údaje, přičemž prvá hodnota (aritmetická celočíselná konstanta 1) se uloží do datové struktury, do proměnné označené identifikátorem (adresou) A, druhá (-5) se uloží do paměťového místa (proměnné) zpřístupňovaného pomocí identifikátoru B a třetí hodnota (6) do proměnné označované identifikátorem C.

Na základě konkrétních hodnot, které byly uloženy do operační paměti počítače příkazem (P0) (do proměnných A, B a C) počítač vykonáním dalších příkazů (P1) a (P2) získá výsledné hodnoty, představující kořeny řešené kvadratické rovnice. Výsledné hodnoty jsou uloženy v proměnných označovaných identifikátory X1 a X2. Shrneme-li výše uvedené, je pak zřejmé, že *definovat obsah datové struktury (tj. uložit do proměnné určitou hodnotu)* lze dvojím způsobem: buď *načtením*, nebo *přiřazením.*

Vytištění obsahu proměnných X1 a X2 způsobíme zadáním příkazu k tisku :

$$write(X1, X2); \tag{P3}$$

Příkaz write zabezpečí vytištění obsahu paměťových míst označovaných identifikátory X1 a X2 na standardně dohodnuté výstupní zařízení, např. obrazovku terminálu.

Pak celý algoritmus má tvar

```
begin  read(A,B,C);
       X1:=(–B + sqrt(B*B – 4*A*C))/(2*A);
       X2:=(–B – sqrt(B*B – 4*A*C))/(2*A);
       write(X1,X2)
end.
```

V algoritmu je použito celkem pět proměnných : A, B, C, X1 a X2. Je zapotřebí před začátkem vlastního algoritmu určit – deklarovat – jejich typ. Deklarací proměnné vymezujeme v operační paměti

počítače prostor pro uložení hodnoty z určité množiny přípustných hodnot, např. celých čísel (a to buď velkých celých čísel, nebo malých celých čísel, popř. malých nebo velkých celých nezáporných čísel), desetinných čísel, znaků aj. Tento vymezený prostor pojmenováváme, adresujeme pomocí tzv. *identifikátoru.* Zatím postačí uvěříme-li, že proměnná (paměťový prostor) typu celé číslo (k uložení celočíselných hodnot) bude co do velikosti asi jiná, než proměnná typu znak (k uložení znaků) nebo proměnná typu *pole* celočíselných hodnot (k uložení více celočíselných hodnot). Dohodneme-li se, že koeficienty A, B a C budou zadávány vždy ve tvaru celých čísel, pak je zřejmé, že X1 a X2 mohou obsahovat také celá čísla, ale také mnohem častěji desetinná čísla. Jak známo množina desetinných čísel obsahuje i množinu celých čísel a tedy potřebné deklarace budou mít tvar :

```
var   A, B, C : integer;        {integer = celé číslo (viz slovníček)}
      X1, X2 : real;            {real = desetinné číslo}
```

Poté bude následovat algoritmus uvedený výše. A, B a C jsou tedy identifikátory označující proměnné (datové struktury) pro uložení hodnoty *typu integer*, tj. celého čísla z intervalu např. $< -32768, 32767 >$ a X1, X2 jsou identifikátory označující proměnné pro uložení hodnot *typu real*, tj. desetinného čísla z konečného počtu (intervalu) určitých reálných hodnot.

Zatím předvedený algoritmus řešení kvadratické rovnice vychází z předpokladu, že řešíme vždy pouze jedinou kvadratickou rovnici, která je navíc řešitelná v oboru reálných čísel. Ze zkušenosti víme, že hodnota diskriminantu $D = b^2 - 4ac$ může být i záporná. Kořeny pak jsou komplexními čísly a rovnici řešíme v oboru komplexních hodnot. Algoritmus rozlišující řešení v oboru reálných a komplexních hodnot a umožňující řešit jakýkoliv tvar kvadratické rovnice bude v následující podobě :

```
var     A, B, C, D : integer;
        X1, X2, RE, IM1, IM2 : real;
begin   read(A,B,C);
        D:=B*B – 4*A*C;
         if D < 0    then    begin   RE :=–B/(2*A);
                                     IM1:= sqrt(ABS(D))/(2*A);
                                     IM2:=–sqrt(ABS(D))/(2*A);
                                     write(RE,IM1,RE,IM2);
                             end
                     else    begin
                                     X1:=(–B + sqrt(D))/(2*A);
                                     X2:=(–B – sqrt(D))/(2*A);
                                     write(X1,X2)
                             end;
end.
```

Dále není jistě žádoucí, aby algoritmus řešil pouze jednu, byť libovolnou, kvadratickou rovnici. Dáme-li výše uvedený algoritmus do cyklu, jehož počet opakování bude determinován existencí trojice koeficientů A, B, C ve vstupním souboru, pak dostáváme následující tvar algoritmu :

```
var     A, B, C, D : integer;
        X1, X2, RE, IM1, IM2 : real;
begin   readln(A,B,C);
        while A <> 0 do
           begin  D:=B*B–4*A*C;
                  if D < 0       then    begin   RE := –B/(2*A);
                                                 IM1:= sqrt(ABS(D))/(2*A);
                                                 IM2:= –sqrt(ABS(D))/(2*A);
                                                 write(RE,IM1,RE,IM2);
                                         end
                                 else    begin
                                                 X1:=(–B + sqrt(D))/(2*A);
                                                 X2:=(–B – sqrt(D))/(2*A);
                                                 write(X1,X2)
                                         end;
                  readln(A,B,C);
           end;
end.
```

Na způsobu ukončení výpočtu kořenů kvadratických rovnic se musíme dohodnout. To, že nemáme již žádnou rovnici k řešení, sdělíme počítači např. tak, že mu zadáme trojici nějakých nesmyslných hodnot. Takovými nesmyslnými hodnotami mohou být koeficienty 0 0 0 (musí být tři, neboť program přikazuje

počítači číst trojici). Požadujeme-li řešit např. čtyři kvadratické rovnice

$$\begin{aligned} 2x^2 - 7x + 11 &= 0 \\ x^2 + 4x + 4 &= 0 \\ x^2 - 5x + 6 &= 0 \\ 3x^2 - 11x + 22 &= 0 \end{aligned}$$

pak vstupní údajový soubor (vstupní data) bude mít následující tvar :

2	−7	11
1	4	4
1	−5	6
3	11	22
0	0	0

1.5 Datové struktury

Jak již bylo řečeno, každý algoritmus pracuje s nějakými hodnotami (některé mohou být vstupní, jiné výstupní, další mohou být hodnoty pomocné, pracovní), operuje nad datovými strukturami - proměnnými, ve kterých jsou hodnoty uloženy. Proměnné jsou organizovány (umístěny, vystavěny) v operační paměti, ve které zabírají určité místo, určitý počet paměťových adresovatelných jednotek - bytů. Aby počítač věděl, že bude s určitou proměnnou pracováno, je zapotřebí tuto proměnnou *deklarovat*, tj. v operační paměti ji vytvořit. Každá stavba někde stojí - má určitou adresu - a každá stavba k něčemu slouží - někdo ji např. obývá - a od toho je odvozena její velikost (malý či velký rodiný domek, vysoký panelák apod.). Nejinak je tomu s proměnnou.

Deklarací proměnné provádíme

a) pojmenování (identifikaci) určitého paměťového místa, tj. tomuto paměťovému místu přiřazujeme jeho adresu v podobě *identifikátoru*

b) přiřazujeme proměnné takový prostor, který je schopen pojmout hodnotu z určité množiny přípustných hodnot, hodnotu určitého *typu dat*.

Všechny deklarace proměnných použitých v programu provádíme v úseku deklarací proměnných, který začíná klíčovým slovem **var**.

Typ dat, zkráceně jen *typ*, charakterizuje určitou množinu, jejíž prvky se nazývají hodnotami typu, a které určité operace mohou být s nimi prováděny.

Definicí typu provádíme

a) pojmenování (identifikaci) typu identifikátorem, pomocí něhož se budeme na definovaný typ dále odvolávat (hovoříme o *identifikátoru typu*)

b) vymezení množiny určitých hodnot a množiny operací, které nad těmito hodnotami lze provádět.

Všechny definice typů, na které se odvoláváme v úseku deklarací proměnných, provádíme v úseku definicí typů, který začíná klíčovým slovem **type**. Nemusíme však provádět definice základních typů dat, které jsou součástí programovacího jazyka. Tyto jazykem definované typy nazýváme *standardními typy* a označujeme je rezervovanými identifikátory typů. Mezi standardní typy jazyka patří jednoduché typy

PASCAL	(TURBO-PASCAL (má větší počet standardních typů))	
a) integer	(shortint,longint,byte,word)	- celočíselné hodnoty
b) real	(single, double, extended)	- reálné hodnoty
c) Boolean		- logické hodnoty
d) char		- znakové hodnoty
a strukturovaný typ		
e)	(text)	- textový soubor

Definice typů píšeme před deklarace proměnných a teprve poté zapisujeme algoritmus - příkazovou část.

Deklarace proměnné má tvar

identifikátor proměnné : identifikátor typu[9]

např. A : T, což čteme jako „proměnná A je typu T“.
Definice typu má tvar

identifikátor typu = popis typu

např. T = **array** [1..10] **of** integer,
což čteme jako „typ T je typ desetice celočíselných hodnot“.

Typy dat se dělí na *jednoduché* a *strukturované*.

1.5.1 Jednoduché typy

Jednoduché typy charakterizují množiny hodnot, které je možno nazvat hodnotami elementárními. Proměnná jednoduchého typu je paměťový prostor např. pro *jedno* celé číslo, nebo prostor pro *jeden* znak apod. Jednoduchými datovými typy všech implementací Pascalu jsou čtyři výše uvedené jednoduché standardní typy, *typ interval* a *typ definovaný výčtem*.

Deklarujeme-li

```
var  A,B    :  real;
     Z      :  char;
     I,J,K  :  integer;
     L1,L2  :  Boolean;
```

pak v operační paměti počítače vzniknou proměnné (buňky)
A [ ] B [ ]
k uložení desetinných čísel z intervalu $\langle \pm 2.9 * 10^{-39} \cdots \pm 1.7 * 10^{38} \rangle$ a každá z proměnných A,B zabírá 6B,
I [ ] J [ ] K [ ]
k uložení celočíselných hodnot z intervalu $\langle -32768, +32767 \rangle$ a každá z proměnných I,J,K zabírá 2B,
Z [ ]
k uložení jednoho znaku z množiny maximálně 256 přípustných znaků, přičemž proměnná zabírá 1B
a konečně L1 [ ] L2 [ ]
k uložení logických hodnot (jsou dvě: **false** a **true**) a každá z proměnných L1 a L2 zabírá 1B.

Definujeme-li intervaly a výčet

```
type  LETOPOCET = 0..2000;
      CELAZAP = -maxint..-1;
      DEN = (pondeli,utery,streda,ctvrtek,patek,sobota,nedele);
      PRACDEN = pondeli..patek;
```

pak deklaracemi

```
var   LP : LETOPOCET;
      C1, C2 : CELAZAP;
      KDY : DEN;
      RX, RY : PRACDEN;
```

vzniknou proměnné
LP [ ] C1 [ ] C2 [ ]
každá o velikosti 2B (neboť jejich možným obsahem je jedna z více než 256 přípustných celočíselných hodnot)
a proměnné
KDY [ ] RX [ ] RY [ ]
každá o velikosti 1B (neboť jejich možným obsahem je jedna z méně než 256 přípustných hodnot, konkrétně u proměnné KDY je to jedna ze sedmi možných hodnot, u proměnných RX a RY je to jedna z pěti možných hodnot - identifikátorů).

[9]Místo identifikátoru typu zde může být uveden přímo popis typu. Jsme však názoru, že především začátečníci by měli deklarovat proměnné pomocí identifikátoru typu buď standardního, nebo výše definovaného. Tedy místo, jak později poznáme, ne vždy vhodného a k cíli vedoucího zápisu

```
var  MAT:array [1..10] of integer
```

píšeme raději

```
type  MATICE= array[1..10] of integer;
var   MAT:MATICE.
```

Zastavme se ještě před strukturovanými datovými typy u typu definovaného výčtem. Použití proměnných definovaných výčtem vede k tvorbě čitelných algoritmů (programů). Je vhodné, aby i čtenáři znalí jiných způsobů algoritmizace než algoritmizace na bázi PASCALu si osvojili používání proměnných definovaných výčtem. Vždyť oč přehlednější je zápis

```
if   POHLAVI = muz then
     if VZDELANI = gymnazium then writeln(PRIJMENI)
```

než běžně často používaný zápis

```
if   POHLAVI = 1 then
     if VZDELANI = 3 then writeln(PRIJMENI)
```

Zatímco ke druhému způsobu vystačíme s deklarací

```
var POHLAVI, VZDELANI : byte
```

je k prvnímu uvedenému způsobu zapotřebí definice a deklarace např. ve tvaru:

```
type   DRUH = (zena,muz);
       SKOLY= (zakladni,vyucen,USO,gymnazium,VS);
var    POHLAVI : DRUH;
       VZDELANI : SKOLY
```

Nevýhodou druhého způsobu zápisu je skutečnost, že uživatel musí znát *způsob kódování informace*, tj např. musí vědět, že informace *žena* je kódována nulou (0) a informace *muž* je kódována jedničkou (1). Nevýhodou prvého způsobu pak je skutečnost, že hodnoty nelze přímo načítat ze standardního vstupního textového souboru, tj. např. z klávesnice, neboť text např. „gymnazium“ je pochopitelně posloupností znaků vzniklých stiskem kláves

[g] [y] [m] [n] [a] [z] [i] [u] [m],

tedy *řetězcem znaků*. A ze znalosti syntaxe i sémantiky PASCALu je nám již zřejmé, že je rozdíl mezi řetězcem znaků a identifikátorem, rozdíl mezi *'gymnázium'* a *gymnazium*[10]. Tuto nevýhodu však odstraňuje použití *netextového souboru*. Problematice vytváření netextového souboru na základě informací (údajů) vstupujících z klávesnice či jiného textového souboru, tvorbě programů zpracovávajících údaje obsažené v netextovém souboru a zápisu požadovaných informací na obrazovku, tiskárnu či do jiného textového souboru je věnována kapitola o algoritmizaci nad soubory.

Jsou-li proměnné POHLAVI a VZDELANI celočíselného typu *byte*, pak způsob načtení vstupních hodnot z klávesnice je možný pomocí příkazů

read(POHLAVI,VZDELANI) resp. readln(POHLAVI,VZDELANI)

Tato jednoduchost má ovšem svá již vzpomínaná negativa: nepřehledný zápis algoritmu pro neznalého způsobu kódování. Jsou-li však proměnné VZDELANI a POHLAVI výčtového typu tak, jak bylo výše uvedeno, pak nelze *hodnoty do nich načítat přímo z klávesnice* [11] (či jiného textového souboru).

Pro vstup z klávesnice je třeba výčtové hodnoty (identifikátory) u všech typů zakódovat. Doporučujeme výčtové hodnoty v pořadí, jak jsou uvedeny ve výčtu, kódovat celými čísly počínaje nulou (0) výše., tj. např. identifikátor *základní* kódovat pro vstup z klávesnice nulou (0), *vyučen* jedničkou (1), *ÚSO* dvojkou (2), *gymnazium* trojkou (3) a *VŠ* čtyřkou (4) a u druhého typu obdobně kódovat identifikátor *žena* nulou (0) a *muž* jedničkou (1). U případných dalších typů bychom postupovali obdobně, což zaručuje provedení potřebné zpětné konverze opět jednoduchým způsobem provést (zpětné překódování). Načtení reprezentantů výčtů a definování obsahů výčtových proměnných pak provedeme příkazy:

```
write('Zadej vzdělání (0 – 4): '); readln(POM);
VZDELANI := SKOLY(POM);
write('Zadej údaj o pohlaví (0 – 1): '); readln(POM);
POHLAVI := DRUH(POM);
```

za předpokladu, že proměnná POM byla deklarována jako celočíselná proměnná např. POM : byte.

Další nespornou výhodou výčtových proměnných je jejich paměťová náročnost. U všech jednoduchých datových typů vyjma typu *real*, tj. u *ordinálních datových typů* jsou hodnoty v počítači reprezentovány svými ordinálními (pořadovými) čísly. U číselných typů je ordinální číslo celočíselné hodnoty rovno této

---

[10] Řetězec 'gymnazium' je něco jiného než řetězec 'GYMNAZIUM' a ten je zase něco jiného než řetězec 'Gymnazium' – viz tabulka ASCII kódu v učebním textu PASCALu. Identifikátor *gymnazium* je totéž jako identifikátor *GYMNAZIUM* popř. *Gymnazium*, tj. uvedené identifikátory identifikují stejný objekt (proměnnou, výčtovou položku aj.) – viz syntaktická definice identifikátoru

[11] Pomocí příkazů read resp. readln můžeme z textového souboru do patřičně deklarovaných proměnných načítat postupně po jednom
a) celém čísle
b) desetinném čísle
c) znaku a často také d) řetězci znaků

celočíselné hodnotě, u ostatních ordinálních typů pak pořadovému číslu hodnoty uvedené v podstatě [12] ve výčtu přípustných hodnot. Ordinální čísla u nečíselných ordinálních typů začínají nulou. Tak např. příkazy

POHLAVI := zena; VZDELANI := USO;

vedou k definování obsahů proměnných POHLAVI resp. VZDELANI takto:

POHLAVI [0] VZDELANI [2]

přičemž každá z proměnných zabírá v operační paměti 1B, pokud jejím definičním oborem je méně, maximálně však 256 přípustných hodnot (identifikátorů). Jinak zabírá 2B.

Uvedený způsob kódování hodnot ordinálního typu v počítači umožňuje jednoduchým způsobem převádět hodnotu jednoho ordinálního typu na hodnotu jiného ordinálního typu, tj. umožňuje provádět *typové konverze.*

Výše použité příkazy VZDELANI := SKOLY(POM) a POHLAVI := DRUH(POM) právě možnost zápisu typové konverze využívají. Je-li v proměnné POM hodnota např. 1, pak výše uvedené příkazy přiřadí *z našeho pohledu* do proměnné VZDELANI hodnotu *vyučen*, do proměnné POHLAVI hodnotu *muž*. V obou proměnných je však *z počítačového hlediska* (ve vnitřní reprezentaci) hodnota stejná, jako je v proměnné POM, tj. obsah proměnných je následující:

POM [1] POHLAVI [1] VZDELANI [1]

1.5.2 Strukturované typy

Strukturované typy charakterizují množiny struktur (uskupení) elementárních hodnot, přičemž se dají vytvářet struktury struktur hodnot do libovolného počtu vnoření. Proměnná strukturovaného typu je paměťový prostor např. pro *několik* celých čísel (pro tzv. pole čísel). Prostor pro uložení matice číselných hodnot je pole polí čísel (tzv. dvourozměrné pole čísel) atp. Strukturovanými datovými typy všech implementací Pascalu jsou:

- homogenní datové typy
  - typ pole (**array**)
  - typ množina (**set**)
  - typ soubor (**file**)
- typ záznam (**record**) jako nehomogenní datový typ

Typ pole umožňuje vytvářet proměnné strukturované na složky stejného typu. Tak např.
definujeme-li

```
type  PEPIK  = array[1..4]  of  integer;
      PEPA   = array[1..5]  of  PEPIK;
```

pak deklarací např.

```
var  R :  PEPIK;
     M :  PEPA
```

vznikne v operační paměti proměnná - struktura **R** k uložení čtyř celočíselných hodnot

R

zabírající celkem 8B (tj. $4 * 2$)
a proměnná **M** k uložení pěti čtveřic celočíselných hodnot

M

[12] I tabulku přípustných znaků (např. kódu ASCII) považujeme za výčet přípustných hodnot, znaků.

zabírající celkem 40B (tj. $5 * 4 * 2$).

S jednotlivými složkami proměnné typu pole pracujeme pomocí indexů, pomocí zápisu indexované proměnné. Hodláme-li obsah proměnné M vynulovat, není zápis M:=0 přípustný – na pravé straně je jen jedna nula (jedna konstanta jednoduchého typu), zatímco levou stranu tvoří mnoho (v našem případě 20) složek (proměnných) jednoduchého typu. Vynulování matice M můžeme provést po řádcích resp. po sloupcích jedním příkazem

```
for  I:=1 to 5 do                          respektive  for  J:=1 to 4 do
     for J:=1 to 4 do M[I,J] :=0;                           for I:=1 to 5 do M[I,J]:=0;
```

Přípustný je také zápis pomocí dvou příkazů s využitím další výše deklarované proměnné R (musí mít tolik složek, kolik sloupců má proměnná M):

```
for I:=1 to 4 do R[I]:=0;
for I:=1 to 5 do M[I]:=R;
```

Máme-li na vstupu (v sešitě) připraveny hodnoty matice rozměru $R \times S$ (R je počet řádků, S je počet sloupců a využijeme-li výše uvedené daklarace proměnné M, pak musí platit, že R <= 5 a S <= 4) ke vstupu do počítače po řádcích resp. po sloupcích, tj. obecně ve tvaru:

$$\begin{array}{l} R\ S \\ m_{11}m_{12}\dots m_{1S} \\ m_{21}m_{22}\dots m_{2S} \\ \vdots \\ m_{R1}m_{R2}\dots m_{RS} \end{array} \qquad \text{resp.} \qquad \begin{array}{l} R\ S \\ m_{11}m_{21}\dots m_{R1} \\ m_{12}m_{22}\dots m_{R2} \\ \vdots \\ m_{1S}m_{2S}\dots m_{RS} \end{array}$$

a s konkrétními hodnotami ve tvaru např.:

```
2   3          respektive      2   3
6  -3   0                      6   4
4   1   1                     -3   1
                               0   1
```

Tyto hodnoty můžeme načíst příkazem:

```
readln(R,S);                                    readln(R,S);
 for  I:=1 to R do                  respektive  for  J:=1 to S do
      for J:=1 to S do read(M[I, J]);                for I:=1 to R do read (M[I, J]);
```

Oba způsoby zápisu příkazů čtení složek pole M slouží *k definování obsahu proměnné M* následujícími hodnotami:

M

6	-3	0	
4	1	1	

Obsah nevyplněných složek je nedefinován.

Indexování složek typu pole je možné pomocí jakýchkoliv hodnot ordinálního typu. Tak např. definujeme-li

type POCITADLO = **array**[char] **of** byte;

pak deklarací

```
var  KOLIK   : POCITADLO;
     Z       : char;
```

vznikne proměnná KOLIK s 256 složkami indexovanými prostřednictvím znaku. Tak příkaz

for Z:=chr(0) **to** chr(255) **do** KOLIK[Z]:=0;

způsobí vynulování obsahů všech složek pole KOLIK. Zvýšení obsahu příslušné složky počítadla KOLIK o jedničku v závislosti na načteném znaku (počítání počtu jednotlivých znaků ze vstupu) provedeme příkazy (zřejmě by byly součástí cyklu):

write('Zadej další znak:');

readln(Z);
KOLIK[Z] := KOLIK[Z] + 1;
a vytištění např. informace o počtu jednotlivých malých písmen (a pouze těch) provedeme příkazem
for Z:='a' **to** 'z' **do** writeln('Písmeno ',Z,' je v textu celkem ',KOLIK[Z],' krát');

Důležitým polem velice často používaným je pole znaků **array of char** (v některých implementacích Pascalu **packed**[13]**array of** char. V Turbo – Pascalu je k vytváření pole znaků - *řetězců* používán další typ,

typ řetězec(**string**)Proměnné typu **string** na rozdíl od proměnných typu **array** jsou svým způsobem dynamické struktury.
Definujeme-li

```
type   RET1 = string[12];
       RET2 = string[20]
```

pak deklarací např.

```
var   AUTOR : RET1;
      NAZEV : RET2
```

vzniknou v operační paměti proměnné AUTOR a NAZEV. Do proměnné AUTOR se vejde až 12 znaků, do proměnné NAZEV až 20 znaků. Proměnná typu řetězec je pole složek typu char indexovaných od nuly po deklarovanou délku N, celkem tedy N+1 bytů. Nultá složka proměnné typu řetězec obsahuje informaci o počtu znaků umístěných v proměnné. Po přiřazení např. příkazy

AUTOR := 'Bílek'; a NAZEV := 'Ola';

je v paměti počítače

AUTOR

5	B	í	l	e	k

a NAZEV

3	O	l	a

obsazeno v proměnné AUTOR 6B a v proměnné NAZEV 4B, po přiřazení např.

AUTOR := 'Adam'; a NAZEV := 'Kamna';

je v paměti počítače

AUTOR

4	A	d	a	m

a NAZEV

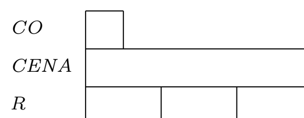
5	K	a	m	n	a

obsazeno v proměnné AUTOR 5B a v proměnné NAZEV 6B.

Typ záznam umožňuje vytvářet proměnné strukturované na složky nestejného typu. Tak např. definujeme-li

```
type   MATERIAL   =   (drevo,kov,umely,jiny);
       ROZMERY    =   array [1..3] of word;
       VYROBEK    =   record R      : ROZMERY;
                             CO     : MATERIAL;
                             CENA   : real;
                      end
```

pak deklarací např.
var NABYTEK : VYROBEK
vznikne v operační paměti proměnná NABYTEK k uložení pěti jednoduchých hodnot: jedné výčtové, tří celočíselných (strukturovaných v jedné proměnné typu pole) a jedné reálné. Proměnná NABYTEK má tvar:



[13]Některé implementace PASCALu blízké standardnímu PASCALu rozlišují u strukturovaných typů **typy zhuštěné** a **typy nezhuštěné**. Jak již název napovídá, u proměnných zhuštěného typu dochází k efektivnímu (úspornému) uložení dat. Nevýhodou pak zase je, že ke složkám takovéto proměnné je možný přístup až po rozbalení proměnné $\Longrightarrow$ časová náročnost.

Proměnná NABYTEK zabírá v paměti celkem $1 + 3 * 2 + 6 = 13$B.

S jednotlivými složkami typu záznam pracujeme pomocí tečkové notace. Hodláme-li do složky CENA proměnné NABYTEK uložit hodnotu 2580.0, pak tak učiníme např. příkazem

NABYTEK.CENA := 2580.0

Obdobně příkazy např.

```
NABYTEK.CO    :=  kov;
NABYTEK.R[1]  :=  160;      {délka}
NABYTEK.R[2]  :=  100;      {šířka}
NABYTEK.R[3]  :=  80        {výška}
```

zapříčiní, že obsah proměnné NABYTEK bude definovát tak, jak ukazuje následující obrázek:

NABYTEK

CO	*1*		
CENA	*2580.00*		
R	*160*	*100*	*80*

Pět výše uvedených přiřazovacích příkazů efektivně zapíšeme s využitím příkazu **with** takto:

with NABYTEK **do begin** CENA := 2850.0;
CO := kov;
R[1] := 160;
R[2] := 100;
R[3] := 80
end

Hodnoty do proměnné můžeme také načíst z klávesnice. Máme-li na vstupu hodnoty 2850.0 1 160 100 80
a je-li proměnná POM typu *byte*, pak příkazy
readln(NABYTEK.CENA,POM,NABYTEK.R[1],NABYTEK.R[2],NABYTEK.R[3]);
NABYTEK.CO := MATERIAL(POM)
resp. ekvivalentní příkaz
with NABYTEK **do**
begin readln(CENA,POM,R[1],R[2],R[3]);
CO := MATERIAL(POM)
end
vedou k definování obsahu proměnné NABYTEK stejně, jak tomu bylo při využití příkazů přiřazovacích.

Typ množina umožňuje vytvářet proměnné, jejichž složkami jsou prvky množiny.
Definujeme-li
type MNOZZNAKU = **set of** char;
pak deklarací
var SAMOHL, CISLICE : MNOZZNAKU;
ZNAK : char;
POCSAM, POCCIS : byte;
vzniknou proměnné SAMOHL, CISLICE, do kterých je možno ukládat množiny vytvářené z množiny všech přípustných znaků. Obě proměnné mají velikost 256 bitů, tj. 32B. Každý potenciální prvek množiny má v proměnné typu množina rezervován jeden bit, přičemž proměnná typu množina zabírá v paměti minimálně 1B a maximálně 32B a její velikost je vždy v celých bytech.

Je-li na vstupu několik alfanumerických znaků ukončených znakem '*' a my máme určit, kolik z nich je samohlásek a kolik z nich je číslic, pak sestavíme za předpokladu výše uvedených definicí a deklarací následující algoritmus:

```
begin POCSAM := 0; POCCIS := 0;
      SAMOHL :=['A','E','I','O','U','Y']
      CISLICE :=['0'..'9'];
      write('Zadej znak :'); readln(ZNAK);
```

```
      while ZNAK <> '*' do begin if ZNAK in CISLICE then POCCIS := POCCIS + 1;
                                 ZNAK:= UpCase(ZNAK);
                                 if ZNAK in SAMOHL then POCSAM := POCSAM + 1;
                                 write('Zadej další znak (konec=*): ');
                                 readln(ZNAK);
                           end;
      writeln('Počet samohlásek v~textu bylo :',POCSAM);
      writeln('Počet číslic v~textu bylo :',POCCIS);
end.
```

Pozn.: Příkaz ZNAK := UpCase(ZNAK) díky funkci UpCase převede případné malé písmeno na velké písmeno. Jedná se o nestandardní funkci, která je součástí implementace Turbo Pascalu. Ve standardu bychom její absenci museli obejít např. příkazem:
if ZNAK **in** ['a'..'z'] **then** ZNAK:=chr(ord(ZNAK)–(ord('a')-ord('A')))

Pro uvedený algoritmus nebylo ale zapotřebí vůbec proměnné typu množina deklarovat, neboť jejich obsah je v daném případě v celém algoritmu konstantní. Algoritmus za předpokladu výše uvedených deklarací pouze proměnných POCSAM, POCCIS a ZNAK by mohl být ve tvaru:

```
begin POCSAM := 0; POCCIS := 0;
      write('Zadej znak :'); readln(ZNAK);
      while ZNAK <> '*' do begin if ZNAK in ['0'..'9'] then POCCIS := POCCIS + 1;
                                 ZNAK:= UpCase(ZNAK);
                                 if ZNAK in ['A','E','I','O','U','Y']
                                    then POCSAM := POCSAM + 1;
                                 write('Zadej další znak (konec=*): ');
                                 readln(ZNAK);
                           end;
      writeln('Počet samohlásek v~textu bylo :',POCSAM);
      writeln('Počet číslic v~textu bylo :',POCCIS);
end.
```

V algoritmu jsme použili *konstanty typu množina* bez nároků na definice a deklarace obdobně, jako jsme použili konstanty 0 a 1 typu celé číslo.

Potřeba deklarovat proměnnou ZNAKY jako proměnnou typu množina znaků, tedy
var ZNAKY : MNOZZNAKU;
by nastala např. v případě, že na vstupu je *jakýchsi* pět znaků, jejichž souhrnný výskyt v dále následujícím textu nás zajímá. Řešením je algoritmus ve tvaru:

```
type MNOZZNAKU = set of char;
var  ZNAKY       : MNOZZNAKU;
     ZNAK        : char;
     I, POCZNAKU : byte;
begin POCZNAKU := 0;  ZNAKY := [];    {prázdná množina}
      for I:=1 to 5 do begin write('Zadej ',I,'. znak do množiny: ');
                             readln(ZNAK);
                             ZNAKY := ZNAKY + [ZNAK]
                       end;
      write('Zadej znak :'); readln(ZNAK);
      while ZNAK <> '*' do begin if ZNAK in ZNAKY then POCZNAKU := POCZNAKU + 1;
                                 write('Zadej další znak (konec=*): ');
                                 readln(ZNAK);
                           end;
      writeln('Počet sledovaných znaků v~textu bylo :',POCZNAKU);
end.
```

Typ soubor umožňuje vytvářet proměnné strukturované na složky stejného typu
(srovnej s typem pole). Rozdíl oproti typu pole je v tom, že proměnná typu soubor není organizována v operační paměti počítače (jako všechny ostatní deklarované proměnné), ale je organizována na vnější

paměti (pevném či pružném disku, obrazovce, tiskárně apod.).

Definujeme-li

```
type  MATERIAL  =  (drevo,kov,umely,jiny);
      ROZMERY   =  array [1..3] of word;
      VYROBEK   =  record R    : ROZMERY;
                          CO   : MATERIAL;
                          CENA : real;
                   end;
      VYROBKY = file of VYROBEK
```

pak deklarací např.

```
var  NABYTEK  : VYROBEK
     F        : VYROBKY
```

vzniká proměnná typu soubor, jejíž umístění (alokace) je specifikována v různých implementacích PASCALu různě (v Turbo Pascalu procedurou *assign*, v některých implementacích procedurou *reset* nebo *rewrite* nebo *open*, v jiných implementacích (též ve standardním PASCALu) v hlavičce programu). Ze souboru můžeme brát (číst) jeho složku pomocí procedury *read* (byl-li otevřen pro čtení), tj. např.

read(F,NABYTEK)

nebo do souboru můžeme složku přidávat (zapisovat) pomocí procedury *write* (byl-li otevřen pro zápis), tj. např.

write(F,NABYTEK)

Existuje-li soubor výrobků F, pak s využitím výše uvedených definicí a deklarací (navíc použitá proměnná A je typu real) jej můžeme v Turbo Pascalu zpracovat po jednotlivých složkách (větách) např. takto:

```
begin assign(F,'SKLAD.TYP');  reset(F);    A := 0;
      while not eof(F) do begin read(F,NABYTEK);
                                if NABYTEK.CO = kov then A := A + NABYTEK.CENA;
                          end;
      close(F);
      writeln('Hodnota všech kovových výrobků na skladě je ',A:10:2)
end.
```

Nutno podotknout, že soubor F není textový soubor, jeho vytvoření není tudíž možné editorem, ale programem. Uvedení problému na tomto místě je pouze orientační a další souvislosti uvedeme v části pojednávající o algoritmizaci nad soubory (4.2).

Důležitou kategorií souborů jsou soubory textové. Základní složkou textového souboru je znak. Znaky se v textovém souboru sdružují v řádky obecně nestejné délky. Řádky znaků tvoří textový soubor, který deklarujeme pomocí standardního identifikátoru strukturovaného typu textový soubor - identifikátoru *text*.

Deklarací

var S : text {něco jako **file of** char}

vzniká proměnná S typu textový soubor. Vedle již zmíněných procedur read a write pracují nad textovými (a pouze textovými) soubory také procedury readln a writeln.

Hodláme-li zjistit, zda v souboru DOPIS.TXT na disketě v mechanice A, který jsme pořídili nějakým editorem je více znaků 'B' (velké písmeno B), nebo více znaků '3' (číslice 3), pak musíme probrat znak za znakem ze všech řádků souboru, zjistit počet požadovaných znaků a vytisknout závěrečné sdělení. Uvedené řeší program např. ve tvaru:

```
Program  ZPRACUJTEXT;
   var   S:text;
         ZNAK:char;
         PB,P3:word;
   begin assign(S,'A:DOPIS.TXT');reset(A);
         PB:=0;    P3:=0;
         while not eof(S) do
         begin while not eoln(S) do
```

```
            begin read(S,ZNAK);
                 if ZNAK='B' then PB:=PB+1
                             else if ZNAK ='3' then P3:=P3+1;
            end;
            readln(S)                                 {opustíme "vyčtený" řádek}
        end;
        close(S);
        if PB>P3 then writeln('Více je písmen B')
              else if PB=P3 then writeln('B je stejně jako 3')
                            else writeln('čísel 3 je více')
    end.
```

Slovně lze uvedený algoritmus popsat takto:

1. Zpracovávanou informací je jeden znak.

   Znak přečteme a pakliže jím je znak 'B', zvýšíme počítadlo těchto znaků PB o jedničku, pakliže jím je znak '3', zvýšíme počítadlo trojek P3 o jedničku.

2. Bod 1. opakujeme tak dlouho, dokud (**while**) není (**not**) konec řádku (**e**nd **o**f **lin**e) v souboru S. Jinak jdeme dále.

3. Odhodíme „vyčtený" řádek (čtecí hlavu nastavíme z konce přečteného řádku na začátek dalšího řádku–příkazem readln(S).

4. Opakujeme body 1–3 dokud (**while**) není (**not**) konec souboru (**e**nd **o**f **f**ile) S. Jinak jdeme dále.

5. Vyhodnotíme obsah počítadel PB a P3 a vytiskneme příslušný text rozhodnutí.

1.6 Struktura programu

Struktura programu, sestává z hlavičky programu[14], např.

Program KVADRROVNICE;

dále z části, ve které uvádíme definice a deklarace, tj. části definicí a deklarací, např.

```
var   A, B, C, D : integer;
      X1, X2, RE, IM1, IM2 : real;
```

a konečně z vlastního algoritmu, tj. části příkazové, např.

```
begin  readln(A,B,C);
       while A <> 0 do
          begin  D:=B*B–4*A*C;
                 if D < 0      then   begin   RE := -B/(2*A);
                                              IM1:= sqrt(ABS(D))/(2*A);
                                              IM2:= –sqrt(ABS(D))/(2*A);
                                              write(RE,IM1,RE,IM2);
                                      end
                               else   begin
                                              X1:=(–B + sqrt(D))/(2*A);
                                              X2:=(–B – sqrt(D))/(2*A);
                                              write(X1,X2)
                                      end;
                 readln(A,B,C);
          end;
end.
```

[14] Hlavička programu v Pascalu-Turbo nemusí být uváděna. Ve standardním PASCALU a jemu blízkých implementacích (např. i v HP Pascalu) je v hlavičce veddle klíčového slova **program** a indentifikátoru programu ještě nutno uvádět seznam externích modulů, tj. především souborů, se kterými konkrétní program pracuje. Tak např. hlavička programu KVADRROVNICE by ve standardním PASCALU musela mít tvar:

Program KVADRROVNICE(input,output);

I v Turbo-Pascalu však z hlediska lepší orientace doporučujeme hlavičku programu ve zkrácené podobě tak, jak je zapsána např. v programu KVADRROVNICE, uvádět.