Ing. Tomáš Hruška, CSc.

PASCAL PRO ZAČÁTEČNÍKY

PRAHA 1989
SNTL — NAKLADATELSTVÍ
TECHNICKÉ LITERATURY

Kniha na příkladech (od nejjednodušších po složitější) vysvětluje základní pojmy jazyka Pascal i metodiku tvorby programů. Výklad je zaměřen tak, aby čtenář detailně pochopil konstrukce jazyka použité v daném programu a mohl jej na svém počítači odladit a spustit. Kapitoly jsou zakončeny definicemi zavedených pojmů a tématy pro samostatná cvičení.

Je určena nejširší veřejnosti, tj. všem, kteří mají zájem se naučit programovat v jazyku Pascal.

Lektorovali: Ing. Josef Vogel, CSc. a RNDr. Rudolf Kryl, CSc.

Redakce teoretické literatury — hlavní redaktorka
RNDr. Blanka Kutinová, CSc.

Odpovědná redaktorka RNDr. Irena Bergerová

© Ing. Tomáš Hruška, CSc., 1989

ISBN 80-03-00345-8

OBSAH

PŘEDMLUVA		7
1	POČÍTAČ A PASCAL	9
1.1	Základní pojmy	9
1.2	Historie Pascalu	10
1.3	Zpracování programu počítačem	11
2	VÝPOČTY S ČÍSLY	15
2.1	Celá čísla	15
2.2	Reálná čísla	30
2.3	Cvičení	39
3	LOGICKÉ HODNOTY	41
3.1	Cvičení	44
4	ŘÍZENÍ	45
4.1	Větvení	45
4.1.1	Podmíněný příkaz	45
4.1.2	Formát zápisu programů	54
4.1.3	Příkaz case	57
4.2	Cykly	65
4.2.1	Cyklus for	65
4.2.2	Cyklus while	68
4.2.3	Cyklus repeat	71
4.3	Cvičení	75
5	ZNAKY, VÝČTOVÉ TYPY, INTERVALY	77
5.1	Znaky a kódy	77
5.2	Jiné nenumerické hodnoty	83
5.2.1	Výčtové typy	85
5.3	Intervaly	88
5.4	Cvičení	89
6	PROCEDURY A FUNKCE	91
6.1	Procedury	91
6.2	Bloková struktura programu	106
6.3	Parametry	111
6.4	Metoda návrhu shora dolů	119
6.5	Funkce	122
6.6	Rekurzívní procedury a funkce	133
6.7	Vedlejší efekty	140
6.8	Funkcionální a procedurální parametry	141
6.9	Direktivy	148
6.10	Cvičení	149

7	POLE	151
7.1	Tabulka měření	151
7.2	Řetězce znaků	160
7.3	Jiné typy indexu	163
7.4	Vícerozměrná pole	170
7.5	Cvičení	182
8	ZÁZNAMY	187
8.1	Popis reálných objektů	187
8.2	Zpracování výsledků	191
8.3	Cvičení	220
9	VSTUPY A VÝSTUPY	221
9.1	Obecné soubory	222
9.2	Textové soubory	245
9.3	Soubory input a output	250
9.4	Konverzační způsob práce s programem	251
9.5	Cvičení	268
10	MNOŽINY	271
10.1	Cvičení	275
11	UKAZATELE	277
11.1	Datové typy a jejich vztahy	277
11.2	Dynamické proměnné	280
11.3	Abstraktní datové typy	306
11.4	Editor	308
11.5	Cvičení	331
12	ZÁVĚR	333
12.1	Příkaz skoku	333
12.2	Cvičení	336
A	SYNTAKTICKÁ DEFINICE PASCALU	337
B	PŘÍRUČKA	349
LITERATURA		361
REJSTŘÍK		363

PŘEDMLUVA

Jedním z nejrozšířenějších programovacích jazyků se stal jazyk **Pascal**. Má mnohé výhodné vlastnosti, které ho předurčují pro využití jak ve výuce, tak v programátorské praxi.

Tato kniha si klade za cíl seznámit čtenáře s vytvářením programů v Pascalu. Při výběru programů byl kladen důraz na interaktivní způsob práce s počítačem.

Text kapitol je orientován převážně na **popis jazyka** v rámci uvedeného programu, nikoliv na metody vytváření programů. Programy je proto třeba považovat za jedno z možných řešení problému, na němž jsou ukázány popisované rysy jazyka.

Příloha A (nazvaná *SYNTAKTICKÁ DEFINICE PASCALU*) obsahuje všechny přípustné pascalské konstrukce, které byly v knize popsány. K definici syntaktické stavby jsou použity syntaktické diagramy. Lze je využít pro kontrolu správnosti nově zapsaného programu.

Příloha B (nazvaná *PŘÍRUČKA*) je určena pro rychlou orientaci v konstrukcích Pascalu (např. při zápisu a ladění programů). Obsahuje všechny popsané konstrukce jazyka v jednoduchých názorných příkladech. Mimo to příloha obsahuje tabulku znakového kódu ASCII.

V knize je užito dvou typů sazby. V sazbě matematických vzorců jsou symboly (funkce a proměnné) vysázeny tradičně malými písmeny *kurzívou*. Jiný způsob sazby je obvyklý u programů. Zde jsou kurzívou obvykle vyznačena jména standardních objektů. Ostatní jména v programech jsou vysázena velkými písmeny. Kombinací obou zavedených způsobů sazby dochází někdy k tomu, že objekty označené v matematickém odvození malými písmeny jsou v odpovídajícím programu označeny písmeny velkými.

Kniha je určena čtenářům bez programátorských zkušeností. Předpokládají se matematické znalosti na úrovni absolventa střední školy. Má sloužit pro první seznámení s jazykem a ne pro jeho úplné osvojení. Proto nejsou některé rysy jazyka v této knize uvažovány (např. datový typ záznam s variantami a konformní schéma pole). Pro další seznámení s jazykem může čtenář využít např. publikace [14, 19]. Formalizovaný popis jazyka ve tvaru normy lze nalézt v příloze publikace [14].

V programech se předpokládá, že obvyklým vstupním zařízením počítače je klávesnice a výstupním zařízením obrazovka displeje. Programy jsou vytvářeny tak, aby s nimi pracovaly konverzačním způsobem. Všechny programy jsou zapsány v jazyce Turbo Pascal a mohou být ihned použity. Překladač Turbo Pascalu byl zvolen proto, že je dostupný jak na 8bitových, tak na 16bitových osobních počítačích.

Pokud to nebylo nutné, nebyly použity speciální vlastnosti Turbo Pascalu a byla využita pouze část shodující se s normou jazyka. Tam, kde jsou speciální vlastnosti Turbo Pascalu využity, je to výslovně uvedeno.

Závěrem bych chtěl poděkovat RNDr. Jiřímu Frankovi za ilustrace, jimiž oživil některé části výkladu.

Autor

1 POČÍTAČ A PASCAL

1.1 Základní pojmy

Algoritmus je postup pro řešení problému z některého oboru činnosti. Skládá se obvykle z určitého počtu kroků, přičemž každý krok popisuje dílčí úsek popisované činnosti. Algoritmus dále určuje, jak za sebou jednotlivé kroky následují.

Algoritmus pracuje s jistým počtem údajů, na jejichž základě je schopen *jednoznačně* rozhodnout, který další krok se bude provádět. Mimo to je důležité, aby algoritmus *vždy skončil* a aby dával *výsledek* své činnosti.

Má-li algoritmus provádět automaticky pracující stroj – *procesor* – předkládáme mu algoritmus zakódovaný ve tvaru *instrukcí*.

Algoritmus je nutné *zakódovat* tak, aby procesor algoritmu *rozuměl* a byl schopen jej vykonávat. Zakódovaný algoritmus se potom nazývá *program* a zápis algoritmu do formy programu *programováním*.

Procesor, jehož programováním se budeme zabývat, se nazývá *počítač*. Je to stroj na zpracování údajů. Údaje získává od okolí na svém vstupu. S těmito údaji provádí předepsané akce a zpracované údaje předává do okolí na výstupu. Zpracovává tedy údaje podle zadaného algoritmu.

Údaje vstupují do počítače a vystupují z něj v různých formách (např. znakově jako text nebo graficky). Vnitřní tvar údajů, se kterým umí procesor pracovat, je však pouze jeden a je dán vlastnostmi elektronických součástek, z nichž je počítač postaven. Údaje mohou nabývat pouze dvou hodnot (0 a 1); proto jim říkáme *dvojkové* nebo *binární*. Všechny ostatní typy údajů (včetně např. programů) se musí před zpracováním procesorem převést do dvojkového tvaru.

Jednotce dvojkové informace říkáme *bit* (z anglického **bi**nary digit – dvojková číslice). Důležitým pojmem je také název skupiny bitů, jež je schopna uchovat údaj o jednom znaku (např. písmenu, číslici). Takové skupině bitů říkáme *slabika* (někdy též nesprávně *bajt* z anglického *byte*) a obsahuje obvykle 8 bitů. Dvojkový tvar je pro počítač výhodný, ale pro člověka zvyklého počítat v desítkové soustavě a psát písmena je naopak nepřirozený.

Tvar dvojkového zobrazení přirozených čísel je obdobný obvyklému desítkovému vyjádření hodnoty. Celé číslo 100 můžeme v desítkové soustavě zapsat

$$1.10^2 + 0.10^1 + 0.10^0$$

a celé číslo 01100100 znamená ve dvojkovém tvaru

$$0.2^7 + 1.2^6 + 1.2^5 + 0.2^4 + 0.2^3 + 1.2^2 + 0.2^1 + 0.2^0,$$
$$64 + 32 + 4 = 100.$$

Rovněž znaky jsou převedeny do dvojkového tvaru. Každému znaku odpovídá jistá kombinace jedniček a nul pevné délky (nejčastěji délky jedné slabiky, tj. 8 bitů). Taková kombinace vyjadřuje ovšem také jistou celočíselnou hodnotu. Této hodnotě říkáme *ordinální číslo* znaku. Na různých počítačích se ordinální čísla přiřazená znakům liší. Říkáme, že existují různé *znakové kódy*. Nejčastěji užívaný kód ASCII (American Standard Code for Information Interchange, tj. *Americký normovaný kód pro výměnu informací*) je možno nalézt v Příloze B. V kódu ASCII odpovídá např. ordinálnímu číslu 100 znak *d*.

Pro zápis algoritmů ve formě blízké člověku i řešeným problémům byla vytvořena celá řada prostředků. Říkáme jim *programovací jazyky*. Pokud využívá programovací jazyk zápisu instrukcí konkrétního počítače, jde o programovací jazyk nižší úrovně; nejčastěji je využíván odborníky k programování speciálních algoritmů. Pokud programovací jazyk užívá pojmů blízkých člověku i řešenému problému bez závislosti na konkrétním počítači, říkáme mu *vyšší programovací jazyk*. Akcím formulovaným ve vyšším programovacím jazyce neříkáme potom instrukce, ale *příkazy*.

Pro každý programovací jazyk musí existovat algoritmus určující, jak program napsaný v tomto jazyce převést do tvaru instrukcí užitého počítače. Protože jde také o transformaci údajů, kde vstupní údaje tvoří program v programovacím jazyce a výstupní údaje proveditelné instrukce, může tuto činnost provádět automaticky počítač podle speciálního programu zvaného *překladač*. Člověk pak může formulovat algoritmy ve vyšším programovacím jazyce, počítač program přeloží a provede.

1.2 Historie Pascalu

Vyšší programovací jazyk *Pascal* byl navržen začátkem 70. let. Jeho autorem je profesor *Niklaus Wirth* z Vysoké školy technické v Curychu. Název jazyka byl zvolen na počest francouzského filozofa, matematika a fyzika Blaise Pascala (1623 — 1662). Autor sledoval návrhem jazyka dva cíle :

- vytvořit jazyk vhodný pro *výuku* programování, který by byl založen na omezeném počtu srozumitelných konstrukcí,

- navrhnout strukturu jazyka tak, aby bylo snadné implementovat Pascal na většině současných počítačů.

Zkušenosti s výukou programování totiž ukázaly, že programátoři používají nejčastěji jazyk, který se naučili nejdříve. Je-li tento jazyk komerčně úspěšný, avšak pedagogicky nevhodný, mohou vznikat těžko napravitelné škody.

První verze Pascalu byla publikována r. 1971 [29]. Využívání Pascalu odhalilo některé nedostatky, proto byla r. 1974 uveřejněna opravená definice jazyka (překlad [13]). Ta se téměř stala normou jazyka v počátcích jeho existence. V dalším období narůstala nutnost vytvořit skutečnou normu jazyka, která by umožňovala jednoduchý *přenos* programů zapsaných v Pascalu mezi počítači. Bylo vydáno několik návrhů normy ISO 7185 (International Standard Organization). Český překlad konečného tvaru normy ISO 7185 je v [14].

1.3 Zpracování programu počítačem

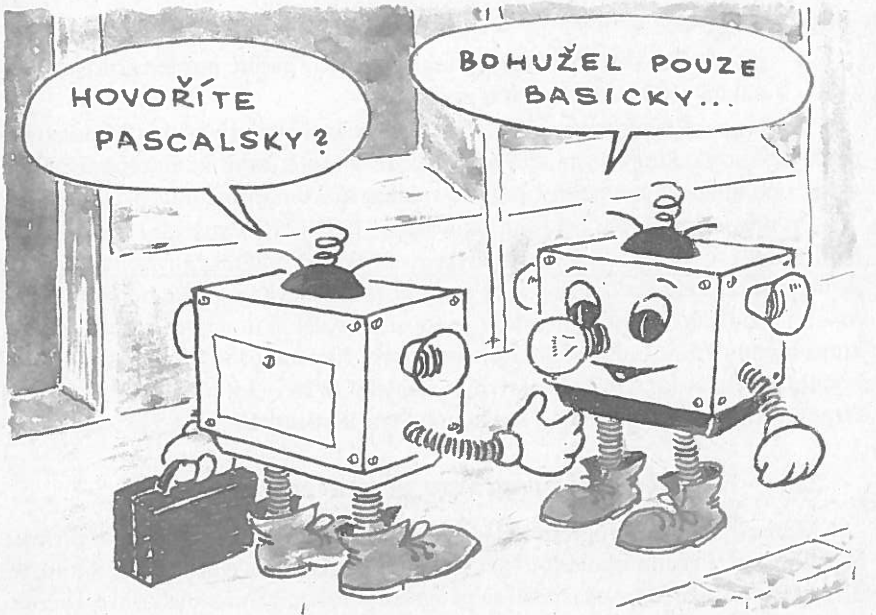
Máme-li napsán program v Pascalu, je nutné před jeho spuštěním provést několik akcí. Prvním úkolem je převedení programu do tvaru, který je možný do počítače vložit. V současnosti se program předem připravuje konverzačním způsobem tzv. *editováním*. K tomu slouží program nazvaný *editor* (kap. 11, program 11.2). Editoru zadáváme klávesnicí text programu i případné opravy.

Druhým krokem je spuštění překladače Pascalu. V Turbo Pascalu jsou editor i překladač spojeny do jednoho celku, čímž umožňují uživateli pohodlnou práci.

Definice programovacího jazyka určuje tvary programů, které je možné překladačem přeložit a spustit. Programy předkládané překladači často definici jazyka neodpovídají (obsahují chyby) a je na překladači, aby chyby odhalil. Skupině chyb, které překladač odhalí v průběhu překladu, říkáme *chyby při překladu*. Obsahuje-li program chyby, je třeba se vrátit k editování, opravit zdrojový program a znovu přeložit.

Pokud proběhne překlad bez ohlášení chyb, je možné program spustit. Při spuštění programu mohou nastat další chyby, tzv. *chyby při spuštění*. Tyto chyby jsou obvykle způsobeny nesprávnými hodnotami vypočtenými při běhu programu, proto je nelze odhalit při překladu.

Je-li program přeložen bez chyb při překladu a nejsou-li hlášeny chyby při spuštění u jednoduchých vstupních údajů, je třeba se přesvědčit, zda pro různé vstupní údaje dává program očekávané výsledky. Obvyklým postupem je tzv. *ladění*. Jde o vytvoření vzorku vstupních údajů, pro něž předem vypočteme výstupní údaje. Pak zkontrolujeme, zda program dává tytéž výsledky. Záleží na volbě testovacích údajů, jak dalece bude program vyzkoušen. Jsou-li výsled-



ky programu chybné, popř. vznikne-li při spuštění chyba, je třeba opravit editorem zdrojový text, přeložit atd.

Laděním nelze jednoznačně zjistit, zda je program správný. Lze se pouze na základě vzorku zkušebních údajů přesvědčit o správném výsledku pro tento vzorek. Proto i často užívané programy mohou obsahovat chyby, které při ladění nebyly odhaleny.

Ukažme si nyní, jak vlastně program v Pascalu vypadá (Program 1.1). Zvolili jsme program pro výpočet spotřeby benzínu z peněz zaplacených za benzín, ceny benzínu a ujetých kilometrů. Tyto údaje se snadno získají tak, že zcela naplníme nádrž a po ujetí určité vzdálenosti akci zopakujeme. Přitom si zapíšeme zaplacenou částku a cenu za 1 l známe. Vztah pro výpočet je jednoduchý:

spotřeba = ((zaplacená částka / cena za 1 l) / počet ujetých km) . 100.

Program je (kromě komentářů ve složených závorkách) napsán bez háčků a čárek. To proto, že překladače Pascalu, s nimiž pracujeme, nejsou schopny

mimo komentáře přijímat znaky české abecedy. Pokud by náš překladač takové vybavení měl, není důvod, proč bychom nemohli (kromě anglických slov charakteristických pro Pascal) psát programy česky.

```
program SPOTREBA (input, output);                                    {1}
{ Program SPOTREBA provádí výpočet spotřeby      }                  {2}
{ motorového vozidla ze zadané ceny za 1l,       }                  {3}
{ vydaných peněz a ujetých kilometrů.            }                  {4}
{ Výsledkem výpočtu je výpis na obrazovku:       }                  {5}
{ Spotreba Tveho vozidla je xx.xx l na 100 km    }                  {6}
var                                                                  {7}
  CENA, PENIZE, VZDALENOST, VYSLSPOTREBA: real;                      {8}
begin                                                                {9}
  writeln('Zadej, prosim, cenu za 1 l,',                             {10}
      ' vydane penize a vzdalenost');                                {11}
  readln(CENA, PENIZE, VZDALENOST);                                  {12}
  VYSLSPOTREBA:=((PENIZE/CENA)/VZDALENOST) * 100;                    {13}
  writeln;                                                           {14}
  writeln('Spotreba Tveho vozidla je ',                              {15}
      VYSLSPOTREBA:5:2,' l na 100 km');                              {16}
end.                                                                 {17}
```

Program 1.1

Řádek {1} programu je jeho záhlavím. Určuje, že program se jmenuje SPOTREBA, a využívá jak obvyklého vstupu (*input*), tak obvyklého výstupu (*output*) údajů. Řádky {2} až {6} tvoří komentáře, které nemají vliv na popisovaný algoritmus. *Komentář* (někdy nazývaný též *poznámka*) začíná znakem { a končí znakem }. Umožňuje nám vpisovat do programu vysvětlující a dokumentující texty. Čísla ve složených závorkách na koncích řádků jsou také komentáře a nejsou nutnou součástí programu. Zavedli jsme je jako pomůcku pro popis prvního programu.

Řádky {7} a {8} obsahují deklaraci *proměnných* (variable = proměnná). Deklarace oznamují, že máme v úmyslu využívat pro označení hodnot, které se v programu používají, slova CENA, PENIZE, VZDALENOST a VYSLSPOTREBA. Oznamujeme dále, že těmito hodnotami budou reálná čísla (*real* = reálné). Proměnná CENA bude obsahovat cenu za 1 l paliva, proměnná PENIZE částku zaplacenou za naplnění nádrže, proměnná VZDALENOST ujetou vzdálenost a konečně proměnná VYSLSPOTREBA vypočtenou výslednou spotřebu vozidla.

Popis algoritmu začíná na řádku {9} (**begin**) a končí na řádku {17} (**end** s tečkou). Jednotlivé příkazy jsou v Pascalu odděleny středníkem.

Prvním příkazem je na řádcích {10} a {11} příkaz *writeln* (**write line** = psát řádek) pro vypsání textu

Zadej, prosim, cenu za 1 l, vydane penize a vzdalenost,

přičemž za tímto textem se provede přechod na nový řádek. Dalším příkazem *readln* (**read line** = číst řádek) na řádku {12} program požadované hodnoty přečte a uloží je postupně do proměnných CENA, PENIZE a VZDALENOST. Čísla se vypisují i zadávají s desetinnou tečkou místo obvyklé desetinné čárky.

Řádek {13} je jádrem programu a představuje předpis pro výpočet spotřeby podle vztahu

VYSLSPOTREBA = ((PENIZE / CENA) / VZDALENOST) . 100.

Po vypočtení spotřeby způsobí řádky {14} až {16} výpis textu

Spotreba Tveho vozidla je xx.xx l na 100 km,

kde xx.xx je reálné číslo vyjadřující spotřebu (číslice :5:2 uvedené za jménem proměnné VYSLSPOTREBA v příkazu *writeln* udávají tvar vystupujícího čísla, tj. 5 znaků celkem, z toho 2 za desetinnou tečkou; desetinná tečka se do celkového počtu znaků započítává). Před výpisem textu se provede přechod na nový řádek (samostatný *writeln* na řádku {14}).

Je-li nám znám význam anglických slov begin, end, read, write, line a variable, představuje uvedený pascalský program srozumitelný zápis algoritmu výpočtu spotřeby.

2 VÝPOČTY S ČÍSLY

2.1 Celá čísla

Celočíselné hodnoty a výpočty s nimi jsou součástí každodenní činnosti. Chceme si např. koupit nábytek a je nám poskytnuta *bezúročná* půjčka 9000 Kčs s tím, že ji od příštího měsíce začneme splácet po 400 Kčs měsíčně (půjčená částka je násobkem měsíční splátky). Zajímá nás, za kolik měsíců půjčku splatíme, ve kterém roce a měsíci budeme platit poslední splátku. Takto uvažujeme např. 2. 6. 1989 a splácet začneme od července.

Počet měsíců, po které budeme platit *plnou* částku 400 Kčs, je dán celou částí podílu 9000 / 400, tj.

počet měsíců = *celá část podílu* (9000 / 400).

Známe-li počet měsíců, po který budeme dlužnou částku splácet, můžeme snadno vypočítat, kolik to bude let a měsíců

počet let = *celá část podílu* (počet měsíců / 12),

zbývající počet měsíců = *zbytek po dělení* (počet měsíců / 12).

Pokud má být výsledkem přímo letopočet a pořadové číslo měsíce v roce, musíme k počtu měsíců splátky přičíst počet měsíců, po který jsme v prvním roce nespláceli, snížený o 1 (tj. 5) a ke zbytku přičíst 1. Dále musíme k počtu let přičíst letopočet prvního roku splátky, tj. 1989. Konečný tvar vztahů pro výpočet potřebných údajů je pak:

počet měsíců = *celá část podílu* (9000 / 400),

letopočet ukončení splátek = *celá část podílu* ((počet měsíců + 5) / 12) + + 1989,

měsíc ukončení splátek = *zbytek po dělení* ((počet měsíců + 5) / 12) + 1.

Algoritmus výpočtu je uvedenými vztahy určen a můžeme jej zapsat v Pascalu. K tomu musíme nejprve zavést některé jazykové prostředky.

Vzhledem k tomu, že náš algoritmus počítá s celými čísly, musíme do programu zapisovat celočíselné hodnoty. Pascal používá pro zápis celých čísel stejný tvar, na který jsme zvyklí, tj. zápis čísel *v desítkové soustavě*. Proto čísla

9000 400 12 5 1989

ale také

−5 −456 −12 +6 +186

jsou správnými zápisy celých čísel v programu.

Algoritmus předepisuje provedení operací *celá část podílu*, *zbytek po dělení* a *sčítání*. Předpis pro vyhodnocení operací s celočíselnými hodnotami nazýváme *celočíselným výrazem*. Výraz je složen z *operandů* a *operátorů*. Operandem výrazu může být např. číslo, výraz nebo (jak později uvidíme) jiný objekt programu vyjadřující hodnotu. Operátor určuje druh *operace*, která se má s operandy provádět (+ určuje sčítání, **div** celočíselné dělení ap.).

Pokud má operace dva operandy, pak obvyklý zápis operace ve výrazu má následující tvar:

operand$_1$ *operátor operand*$_2$

Tento tvar zápisu, kde mezi dvěma operandy je operátor určující operaci, která se má provést, se nazývá *infixový tvar zápisu* operace. Operátoru se dvěma operandy říkáme také *binární operátor*. Pro zápis operací z našeho algoritmu jsou v Pascalu tyto operátory :

- pro sčítání +,
- pro celou část podílu **div** (**div**ide = dělit),
- pro zbytek po celočíselném dělení **mod** (**mod**ulo).

Sčítání celých čísel není třeba blíže vysvětlovat. Operace **div** a **mod**, které mají dva celočíselné operandy a celočíselný výsledek, popíšeme podrobněji.

Celá část podílu, někdy také nazývaná *celočíselné dělení*, je dělením, které dává vždy celočíselný výsledek, i když čísla nejsou beze zbytku dělitelná; výsledkem je pouze celá část podílu.

Přesněji můžeme operaci **div** definovat takto:

Označme abs(A) absolutní hodnotu celého čísla A. Pak pro hodnotu

$$P = A \textbf{ div } B$$

platí (* vyjadřuje operátor násobení)

$$\text{abs}(A) - \text{abs}(B) < \text{abs}(P * B) \leq \text{abs}(A).$$

Tak např.

25 **div** 4 = 6,
−25 **div** 4 = −6,
32 **div** 8 = 4,
900 **div** 400 = 22.

Výpočet zbytku po celočíselném dělení je často prováděn operací **mod**. Můžeme ji definovat takto:

Pro hodnotu

$$Q = A \textbf{ mod } B,$$

kde $B > 0$, platí

$$0 \leq Q < B,$$

tj. výsledek Q je nezáporný a menší než druhý operand a

$$Q = A - (K * B),$$

kde K je celé číslo. Pro nezáporné A se

$$K = A \textbf{ div } B,$$

pro záporné A je

$$K = (A \textbf{ div } B) - 1.$$

Tedy pro nezáporné A a kladné B platí

$$A \textbf{ mod } B = A - (A \textbf{ div } B) * B$$

a pro záporné A a kladné B platí

$$A \textbf{ mod } B = A - ((A \textbf{ div } B) - 1) * B.$$

Například

$$25 \textbf{ mod } 4 = 25 - (25 \textbf{ div } 4) * 4 = 25 - 6 * 4 = 25 - 24 = 1,$$

$$-25 \textbf{ mod } 4 = -25 - ((-25 \textbf{ div } 4) - 1) * 4 = -25 - (-6 - 1) * 4 =$$

$$= -25 - (-28) = 3,$$

$$9000 \textbf{ mod } 400 = 9000 - (9000 \textbf{ div } 400) * 400 = 9000 - 22 * 400 = 200.$$

S využitím uvedených operátorů bychom mohli zapsat celočíselné výrazy pro výpočet měsíců, letopočtu a měsíce ukončení splátek

$$\text{počet měsíců} = 9000 \textbf{ div } 400,$$

$$\text{letopočet ukončení splátek} = ((\text{počet měsíců} + 5) \textbf{ div } 12) + 1989,$$

$$\text{měsíc ukončení splátek} = (\text{počet měsíců} + 5) \textbf{ mod } 12 + 1.$$

Kromě uvedených operací poskytuje Pascal ještě další dvě operace se dvěma celočíselnými operandy a celočíselným výsledkem užívané v celočíselném výrazu. Jsou to

- odčítání s operátorem $-$,
- násobení s operátorem $*$.

S uvedenými operátory je možné zapisovat např. následující výrazy

26 **mod** 5 s hodnotou 1,
35 * 15 s hodnotou 525,
525 + 1 s hodnotou 526,
526 – 2 s hodnotou 524.

Operandem libovolného operátoru může být opět výraz, takže lze zapisovat i složitější výrazy skládající se z více operátorů a operandů.

Následující zápis je správným zápisem celočíselného výrazu

$$35 * 15 + 26 \textbf{ mod } 5 - 2 \qquad (2.1)$$

a skládá se z podvýrazů

$$35 * 15 \text{ a } 26 \textbf{ mod } 5$$

a celého čísla 2. Podvýrazy 35 * 15 a 26 **mod** 5 jsou operandy operace +, která spolu s nimi tvoří výraz

$$35 * 15 + 26 \textbf{ mod } 5.$$

Tento výraz je pak prvním operandem a číslo 2 je druhým operandem operace –, která tvoří konečný tvar výrazu (2.1).

Strukturu uvedeného výrazu (tj. příslušnost operandů k operátorům) můžeme snadno vyjádřit pomocí závorek

$$(35 * 15) + (26 \textbf{ mod } 5) - 2.$$

Závorky vyjadřují skutečnost, že 35 a 15 jsou operandy operace *, 25 a 5 operandy operace **mod**. Výsledky operací * a **mod** jsou operandy operace +, výsledek sčítání a číslo 2 jsou operandy operace –. Z výrazu (2.1) však jeho struktura přímo neplyne. Teoreticky by bylo možné vytvořit i jiné struktury, např.

$$(35 * (15 + 26)) \textbf{ mod } (5 - 2)$$

nebo

$$((35 * (15 + 26)) \textbf{ mod } 5) - 2.$$

Zápis výrazu však nemůže připouštět více možností pro vyhodnocování (více struktur pro jediný výraz). Proto struktura každého výrazu je jednoznačně určena tzv. *pravidly priority* operátorů. Ta určují, které operátory mají při vyhodnocování přednost. Priorita operátorů je převzata z matematiky:

Operátory *, **mod** *a* **div** *mají vyšší prioritu než operátory* + *a* –.

Podle těchto pravidel bude tedy výraz (2.1) chápán jako výraz

$$(35 * 15) + (26 \textbf{ mod } 5) - 2.$$

Uvedený výraz (2.1) je podle pravidel priority vyhodnocován takto:

$$35 * 15 + 26 \textbf{ mod } 5 - 2$$
$$525 + 1 - 2$$
$$524$$

a má tedy výslednou hodnotu 524.

Operátorů + a – lze v celočíselném výrazu užít i v jiném významu než pro vyjádření operace sčítání a odečítání, a to jako operátoru pouze s jedním operandem. Operátoru s jedním operandem říkáme také *unární* operátor. S využitím unárních operátorů můžeme psát výrazy jako

$$-(5-268)-5*6.$$

Není dovoleno zapisovat více unárních operátorů + a – za sebou. Unární operátor nemůže také bezprostředně následovat za binárními operátory. Proto zápis

$$+ -5 * -3$$

je chybný.

Přirozená priorita operací často nevyhovuje. V takovém případě lze pro vyjádření jiné požadované struktury výrazu využít okrouhlých závorek (tak, jak jsme je užili pro vyjádření struktury v předchozím textu). Zápisem části výrazu do závorek způsobíme, že operátory uvedené v závorkách budou mít vyšší prioritu.

Závorky mohou být vnořeny do sebe, a to do libovolné úrovně. V takovém případě mají nejvyšší prioritu operátory v nejvíce vnořených závorkách.

Vlastní vyhodnocení výrazu v programu je aktivováno zápisem výrazu v některých příkazech jazyka. Jednou z možností je uvést výraz v příkazu pro výpis hodnoty (*write*, resp. *writeln*); např. zápis

```
writeln ( 94 )
```

způsobí výpis hodnoty výrazu v závorkách, tj.

```
94
```

na obrazovku. Jedním příkazem *writeln* můžeme na jeden řádek obrazovky vypsat i více hodnot výrazů. V příkazu jsou pak výrazy odděleny čárkami, např.

```
writeln ( 1967 - 1692, -2150, -5+13, (35 * (15+26))mod(5 - 2) )
```

způsobí výpis čísel

```
275    -2150    8    1
```

na obrazovku. Je vidět, že v příkazu *writeln* můžeme uvádět složitější výrazy i pouhá čísla (číslo je elementární výraz). Provedením příkazu *writeln* se psaní

na řádek obrazovky ukončí a další výstup se bude provádět na řádek následující. Chceme-li, aby několik příkazů provedlo výstup na jediný řádek displeje, můžeme užít příkazu *write*, který přechod na nový řádek neprovede. Potom zápis

```
write ( 1967 – 1692 );
write ( – 2150 );
write ( – 5 + 13 );
writeln (( 35 * ( 15 + 26 )) mod ( 5 – 2 ))
```

skládající se ze čtyř příkazů (tři příkazy *write* a jeden příkaz *writeln*) popisuje týž algoritmus jako předchozí ukázka a výstup na obrazovku bude týž. Příkaz *writeln* může být uveden i bez parametru

```
writeln
```

pak způsobí pouze přechod na nový řádek výstupu. Poznamenejme znovu, že dva sousední příkazy se v Pascalu oddělují středníkem.

Budeme-li chtít zobrazit na obrazovku výsledky našeho programu o půjčce (počet měsíců, letopočet ukončení splátek, měsíc ukončení splátek) tak, aby každé číslo bylo na zvláštním řádku, zapíšeme to takto:

```
writeln ( 9000 div 400 );
writeln (( 9000 div 400 + 5 ) div 12 + 1989);                    (2.2)
writeln (( 9000 div 400 + 5 ) mod 12 + 1 )
```

Tento zápis je správným popisem algoritmu výpočtu splátek v Pascalu i s výstupem výsledků, není však ještě programem.

Popišme nyní podrobněji základní stavební kameny, z nichž je program tvořen.

Celá čísla jsme již poznali. Zápis čísel *přímo vyjadřuje jejich hodnotu*. Podobných prvků je v Pascalu více a říkáme jim *literály*. Hodnota celého čísla, kterou můžeme do programu zapsat (ale také vypočítat), je shora i zdola omezena a je dána vlastnostmi počítače. Každému číslu je totiž v operační paměti vyhrazen určitý prostor, např. 16 bitů. Pak jsou k dispozici hodnoty od – 32767 do 32767.

Dalšími důležitými prvky programu jsou jména objektů programu. Říkáme jim *identifikátory* a v našich programech jsou pro snadné odlišení psány velkými písmeny. Pascal definuje identifikátor jako posloupnost písmen a číslic začínající písmenem. Přitom se nedělá rozdíl mezi písmeny malými a velkými. Správnými identifikátory jsou např.

ALFA2 PUJCKA *output* XY23B aLfA2 alfa2 *writeln*

přičemž první, pátý a šestý identifikátor jsou shodné. Délka identifikátoru není omezena, v úvahu se však obvykle bere jen určitý počet znaků na jeho počátku. Avšak posloupnosti znaků

ALFA#2 2PUJCKA OUT_PUT 23BXZ

nejsou v Pascalu identifikátory.

ALFA#2 obsahuje nedovolený znak #, 2PUJCKA začíná číslicí, OUT_PUT obsahuje nedovolený znak _ a 23BXZ začíná opět číslicí. Poznamenejme, že u mnohých překladačů Pascalu je znak _ (podtržení) v identifikátorech povolen, např.

SOUCET_FUNKCNICH_HODNOT TISK_VYSLEDKU

To umožňuje zapisovat popisné identifikátory skládající se i z několika slov.

Pomocí identifikátorů jsou tedy v programu pojmenovány objekty (např. program 1.1 je pojmenován identifikátorem SPOTREBA a proměnné v programu identifikátory CENA, PENIZE, VZDALENOST ap). Autor programu pojmenovává vždy jen ty objekty, které sám vytváří. Objekty, které mu dává k dispozici jazyk (např. jména příkazů pro vstup a výstup), mají identifikátory standardní (např. *writeln* nebo *write*). Standardních identifikátorů je sice možné použít pro označení vlastních objektů (původní standardní objekt pak přestává být k dispozici), ale program tím ztrácí na srozumitelnosti. Proto je budeme užívat *pouze* v jejich původním významu.

Standardní identifikátory budeme v našich programech zapisovat malými písmeny a zdůrazňovat *kurzívou*. Uživatelem vytvořené identifikátory budeme zapisovat velkými písmeny.

Obdobný tvar jako identifikátory (posloupnost písmen) mají speciální symboly nazvané *vyhrazená* (též rezervovaná nebo klíčová) *slova*. Jejich význam tvůrce programu *nemůže* změnit, a musí je proto vždy použít v jejich původním významu. V našich programech je budeme zapisovat malými písmeny a zvýrazňovat **půltučným tiskem**. V programu 1.1 můžeme nalézt vyhrazená slova **program**, **var**, **begin** a **end**.

Mimo identifikátory, vyhrazená slova a literály se program skládá ze znakových a speciálních symbolů, např.

; { } ( ) := < > < = ,

které mají stejně jako vyhrazená slova pevný význam daný definicí jazyka.

O tom, že mezi závorkami { } je uveden komentář, jsme se již zmínili. K nutnosti systematického dokumentování programu pomocí komentářů se ještě dostaneme.

Vraťme se k naší úloze amortizace splátek. Výpočet (2.2) řeší sice naši

úlohu, ale není nejvýhodnější, protože výpočty některých podvýrazů se opakují, např.

9000 **div** 400

9000 **div** 400 + 5.

Bylo by výhodné mít k dispozici prostředek pro dočasné uložení hodnoty, kam bychom mohli hodnotu uložit a opětně ji získat. Tímto prostředkem je v Pascalu *proměnná* spolu s *přiřazovacím příkazem*.

Proměnná reprezentuje v programu objekt uchovávající hodnotu, která se v čase může měnit. Objektů reálného světa vyjadřujících hodnotu, která se mění v čase, je velmi mnoho. Náš věk se pravidelně zvyšuje o 1 každý rok, obsah peněženky kolísá od jedné výplaty k druhé, dveře jsou zavřeny nebo otevřeny ap. Každý takový objekt je pojmenován (věk, peněženka, dveře) a nabývá hodnoty z jisté množiny hodnot (věk z intervalu 0 až 200, peněženka z intervalu 0 až 10000, dveře hodnot z množiny {zavřeno, otevřeno}). Podobně i v Pascalu musíme před použitím každou proměnnou pojmenovat identifikátorem a přiřadit jí množinu hodnot, jichž může nabývat. Množině hodnot spolu s operacemi, které nad ní můžeme provádět, říkáme *datový typ* (často zkráceně pouze *typ*). Pojmenování proměnné spolu s určením jejího datového typu nazýváme v Pascalu *deklarací proměnné*. Každá proměnná má tedy jméno a je jistého datového typu. Jiných hodnot než těch, které jsou určeny datovým typem, nesmí proměnná nabýt.

Prozatím jsme se seznámili s jediným datovým typem, a to celočíselným (v Pascalu nazývaným *integer*). Deklarace celočíselné proměnné POCETMESICU by měla tvar

POCETMESICU : *integer*;

Více proměnných téhož typu můžeme deklarovat odděleně

POCETMESICU : *integer*;
ODZACATKU : *integer*;

nebo můžeme provést deklaraci proměnných společně

POCETMESICU, ODZACATKU : *integer*;

Deklarací *není* proměnné přiřazena hodnota.

Proměnné můžeme přiřadit hodnotu různě. Nejčastěji pomocí *přiřazovacího příkazu*, který má tvar

identifikátor proměnné := *výraz*

Našim proměnným POCETMESICU a ODZACATKU můžeme přiřadit hodnoty např. takto:

```
POCETMESICU := 9000 div 400;
ODZACATKU   := 9000 div 400 + 5
```

Proměnné se mohou vyskytovat nejen před přiřazovacím symbolem := (je to dvouznakový symbol a je nutné jej zapisovat bez mezery), ale i jako prvky výrazu. Je tedy možné napsat např.

```
ODZACATKU := POCETMESICU + 5
```

Přiřazovací příkaz je nejčastěji využívaným příkazem v pascalských programech. Především je nutné si uvědomit, že *nejde o zápis rovnice*, ale o předpis k provedení jisté akce. Vždy se *nejprve* vyhodnotí výraz *za* symbolem := a *potom* se jeho výsledná hodnota přiřadí proměnné *před* symbolem :=. Hodnoty napravo od symbolu := jsou použity jako operandy pro vyhodnocení výrazu. Mění se pouze hodnota proměnné nalevo od symbolu :=. V naší dvojici příkazů

```
POCETMESICU := 9000 div 400;
ODZACATKU   := POCETMESICU + 5
```

se nejprve vyčíslí výraz

```
9000 div 400
```

a proměnné POCETMESICU se přiřadí hodnota 22. Potom se vyčíslí výraz

```
POCETMESICU + 5
```

a proměnné ODZACATKU se přiřadí hodnota 27.

Použijeme-li proměnnou ve výrazu, je *nutné*, aby měla předem přiřazenu hodnotu (samotná deklarace, jak víme, proměnné hodnotu nepřiřazuje). V opačném případě dojde k chybě při spuštění programu.

Vyskytuje-li se v přiřazovacím příkazu táž proměnná jak ve *výrazu* za symbolem :=, tak i před tímto symbolem, vyčíslí se výraz s původní hodnotou proměnné a proměnné se přiřadí výsledná hodnota výrazu. Pouhým použitím ve *výrazu* se hodnota proměnné *nemění*. Jedinou proměnnou, jejíž hodnota je po provedení přiřazovacího příkazu změněna, je proměnná na levé straně přiřazovacího symbolu :=.

Následující posloupnost příkazů ukazuje změny hodnot dvou proměnných I a J v průběhu výpočtu. Změny hodnot jsou popsány přiřazovacími příkazy:

```
                { I, J nemají definovány hodnoty }
I := 28;        { I = 28, J nemá definovánu hodnotu }
```

```
J := I - 10;                { I = 28, J = 18 }
I := ( J * I ) mod 5;       { I = 4 , J = 18 }
J := J + 7                  { I = 4, J = 25 }
```

S využitím proměnných a přiřazovacích příkazů zapišme nyní celý program pro výpočet doby splátek půjčky (program 2.1):

```
program PUJCKA (output);
{ Program vypočte dobu splátek v měsících,poslední rok a měsíc splátek
  u bezúročné půjčky 9000 Kčs. Měsíční splátky jsou 400 Kčs a začíná
  se splácet v 7. měsíci roku 1989.

  Vstupy programu    : nejsou
  Výstupy programu   : na třech řádcích pod sebou

                       počet měsíců
                       letopočet ukončení splátek
                       měsíc ukončení splátek
}
var
  POCETMESICU,              { počet měsíců }
  ODZACATKU: integer;       { počet měsíců od začátku roku }

begin { PUJCKA }
  POCETMESICU:=9000 div 400;
  ODZACATKU:=POCETMESICU + 5;
  writeln ( POCETMESICU );
  writeln (( ODZACATKU div 12 + 1989);
  writeln ( ODZACATKU mod 12 + 1)
end. { PUJCKA }
```

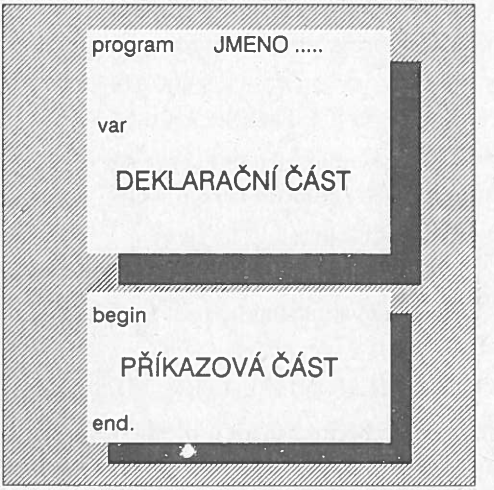
Program 2.1

Program 2.1 není příliš rozsáhlý. Lze na něm však již ukázat základní části pascalského programu.

Každý program napsaný v jazyce Pascal má dvě hlavní části (obr.2.1), a to část *deklarační* a část *příkazovou*.

Části deklarační začíná jménem programu (v našem příkladu PUJCKA), které může mít význam např. pro další zpracování programu počítačem. V deklarační části jsou dále uvedeny prostředky pro styk programu s okolím (např.

output = výstup znamená, že program bude provádět výstup na obvyklé zařízení, např. displej). Hlavním úsekem deklarační části jsou deklarace objektů jazyka, v našem případě pouze proměnných (POCETMESICU a ODZACATKU datového typu *integer*). Deklarace proměnných jsou uvedeny vyhrazeným slovem **var**.



Obr.2.1 Hrubá struktura programu

Deklarace je informací pro překladač jazyka o objektech vytvořených uživatelem, které se budou v programu vyskytovat, a jejich vlastnostech. V Pascalu jsou deklarace povinné a každý objekt, který je v programu použit a není *standardní* (tj. poskytovaný definicí jazyka), musí být *před použitím* deklarován.

Na základě informací získaných v deklaracích může překladač provádět kontrolu správného užití objektů v programu i odhalit mnohé chyby. To zrychluje a zefektivňuje ladění. Práce *navíc* se zápisem deklarací se vždy vyplatí.

Algoritmus řešení úlohy je popsán ve příkazové části programu, která se skládá z příkazů. Začíná klíčovým slovem **begin** a končí **end** s tečkou.

Variantu programu 2.1 nelze považovat za nejlepší. Provádí sice algoritmus efektivně, ale nepřijímá vstupní údaje ze svého okolí, neboť celková částka, výše splátek, datum začátku splátek ap., jsou zadány přímo ve formě čísel v textu programu a program je proto jednoúčelový. Pro jednoúčelové výpočty však nemá smysl vytvářet programy. Zvládneme je snadněji na kalkulátoru. To,

co z počítače dělá silný nástroj, je *univerzálnost* (nebo též hromadnost) programů schopných dávat výsledky pro různé vstupní údaje.

Představme si nyní, že počítač ve spořitelně používá referent pro poskytování bezúročných půjček, takže na základě výše půjčky, měsíčních splátek a zadaného data vám ihned vypočte dobu a měsíc ukončení splátek. Vztahy pro výpočet jsou stejné, až na to, že se nebude počítat s literály, ale s proměnnými VYSEPUJCKY, MESICNISPLATKA, MESIC a ROK.

Přiřazovací příkazy z programu 2.1 budou mít nyní tvar

```
POCETMESICU := VYSEPUJCKY div MESICNISPLATKA;
ODZACATKU   := POCETMESICU + MESIC - 1
```

První a poslední příkaz *writeln* zůstanou beze změny, jen příkaz pro výpočet letopočtu bude modifikován zadaným rokem ROK

```
writeln ( ODZACATKU div 12 + ROK )
```

Počet měsíců v roce je zapsán číslem 12, protože je *konstantní*. Do programu jsme přidali čtyři proměnné, a musíme je proto deklarovat

```
var
  VYSEPUJCKY, MESICNISPLATKA, MESIC, ROK: integer;
```

Jak víme, deklarací se hodnota těmto proměnným nepřiřadí. Přiřazovací příkaz není možné použít, neboť chceme získat údaje z okolí programu prostřednictvím vstupního zařízení. Pro čtení hodnot a jejich přiřazení proměnným poskytuje Pascal obdobné prostředky jako pro výstup hodnot. Základním příkazem pro čtení hodnot je příkaz *readln*, např.

```
readln (VYSEPUJCKY)
```

který přečte celočíselnou hodnotu z jednoho řádku napsaného z klávesnice. Hodnotu přiřadí proměnné VYSEPUJCKY. *Po* provedení vstupu přejde na čtení dalšího řádku z klávesnice. Tvar čísel zadávaných na klávesnici je týž, jako při zápisu čísel do textu programu. Sousední čísla jsou oddělena mezerami, nebo mohou být na různých řádcích.

Všechny mezery a konce řádků napsané na vstupním zařízení před čtenou hodnotou jsou *před* čtením vlastní hodnoty přeskočeny.

Podobně jako u výpisu příkazem *writeln* je možné přečíst z jednoho řádku i více hodnot zápisem

```
readln (VYSEPUJCKY, MESICNISPLATKA, MESIC, ROK)
```

nebo použít příkazu *read* (Ten neprovádí po čtení přechod na nový řádek vstupního textu). Zápis se stejným významem je pak např.:

```
read ( VYSEPUJCKY );
read ( MESICNISPLATKA );
read ( MESIC );
readln ( ROK )
```

skládající se ze čtyř příkazů.

Program 2.2 je již univerzálním programem pro výpočet požadovaných údajů na základě vstupních hodnot. Poznamenejme, že provádění vstupu ze standardního vstupního zařízení (klávesnice) je podmíněno uvedením identifikátoru *input* v záhlaví programu.

```
program PUJCKA (input, output);
{ Program vypočte dobu splátek v měsících, poslední rok a měsíc splátek
u bezúročné půjčky.
Vstupy programu    :  na jednom řádku oddělené mezerou
     výše půjčky   měsíční splátka   aktuální měsíc   aktuální rok

Výstupy programu   : na třech řádcích pod sebou
     počet měsíců
     letopočet ukončení splátek
     měsíc ukončení splátek
}
var
  POCETMESICU,          { počet měsíců splátky }
  ODZACATKU,            { počet měsíců od začátku 1.roku }
  VYSEPUJCKY,           { výše půjčky v Kčs }
  MESICNISPLATKA,       { měsíční splátka v Kčs }
  MESIC,                { číslo měsíce, kdy se začalo splácet }
  ROK : integer;        { rok, kdy se začalo splácet }

begin { PUJCKA }
  readln( VYSEPUJCKY, MESICNISPLATKA, MESIC, ROK);
  POCETMESICU:= VYSEPUJCKY div MESICNISPLATKA;
  ODZACATKU := POCETMESICU + MESIC -1;
  writeln  ( POCETMESICU );
  writeln  (( ODZACATKU div 12 ) + ROK);
  writeln  ( ODZACATKU mod 12 + 1)
end. { PUJCKA }
```

Program 2.2

Přeložíme-li a spustíme program 2.2, rozběhne se a bez jakéhokoliv výpisu na obrazovku se zastaví a čeká na zadání čtyř čísel, které se budou číst příkazem *readln*. Po zadání čísel, např.

9000 400 6 1989

se vypíší tři řádky

22
1991
4

opět bez textu nebo vysvětlení. Co je co se dozvíme z komentáře v programu, pokud jsme byli pečliví a program komentovali. Zdrojový tvar programu s komentáři však téměř nikdy nemáme k dispozici, nejsme-li tvůrci programu. Proto by program pracující konverzačním způsobem měl vždy svou činnost komentovat vysvětlujícími *texty*, z nichž by i laický uživatel poznal, jak program užívat.

Výpis *textu* na obrazovku se v Pascalu provádí obdobně jako výpis hodnoty výrazu, a to příkazy *write* nebo *writeln*. Stejně jako existují literály číselné, existují i literály textové, a ty je možné uvedenými příkazy vypsat.

Textový literál (říkáme mu *řetězec znaků*) je libovolný text uvedený mezi dvěma apostrofy (znaky '). Jako příklady uveďme řetězce

'VYPOCET DOBY SPLATNOSTI'
'Pocet mesicu splatky je'
'Ktery mesic je nyni ?'

Znakové řetězce mohou obsahovat libovolné znaky (nejen písmena a číslice). Můžeme zde psát cokoliv bez ohledu na deklarované identifikátory i existenci jiných objektů v programu. Apostrofy nejsou součástí řetězce, pouze jej ohraničují. Jediný problém je s vyjádřením samotného znaku apostrof (máli se vyskytovat uvnitř řetězce), protože jeho uvedením se řetězec ukončuje. Proto, chceme-li v textu řetězce uvést apostrof jako platný znak, musíme jej *zdvojit* (tj. zapsat dvakrát za sebou). Tím do textu řetězce zařadíme *jediný* apostrof a řetězec se neukončí. Například

'ISN''T' vyjadřuje text ISN'T

Následující příkazy ukazují použití řetězců v příkazech *writeln* a *write*:

```
writeln ('VYPOCET DOBY SPLATNOSTI');
write   ('Doba splatnosti je ');
```

Hodnoty číselných výrazů vypisované příkazem *write* nebo *writeln* obsahují vždy jistý počet platných číslic. Způsob výstupu, který jsme doposud využívali v této kapitole, způsobí výpis hodnoty výrazu vždy na stejný počet znaků. Tento

počet je dán vlastnostmi překladače Pascalu a způsobu výstupu říkáme *neformátovaný*, např. příkaz

writeln ('Doba splatnosti je ',POCETMESICU);

by při standardní šířce neformátovaného výstupu 12 znaků vypsal

Doba splatnosti je 22

Zpráva je sice správná, ale chtěli bychom, aby výpis hodnoty byl zobrazen třemi znaky. To lze v programech vyjádřit následovně

writeln ('Doba splatnosti je ',POCETMESICU:3)

tj. uvést za dvojtečkou výraz, který určuje, kolik znaků se má využít pro výstup hodnoty výrazu před dvojtečkou. Tomuto způsobu výstupu říkáme *formátovaný* a užili jsme jej již v programu 1.1. Pokud se hodnota na udaný počet znaků nevejde, vypíše se na tolik znaků, kolik je potřeba. Předchozí příkaz tedy způsobí pro stejné vstupní údaje výpis

Doba splatnosti je 22

Než uvedeme konečný tvar programu PUJCKA, poznamenejme, že proměnná ODZACATKU je v programu zbytečná, neboť místo ní je možné s výhodou použít proměnné POCETMESICU. Tato proměnná nejprve nabude hodnoty počtu měsíců splátky. Po jejím výpisu se hodnota změní na počet měsíců od začátku prvního roku tak, jak je to v další části programu potřebné. Konečný tvar programu je univerzálním konverzačním programem pro výpočet údajů o půjčce.

```
program PUJCKA (input, output);
{ Program vypočte dobu splátek v měsících, poslední rok a měsíc splátek
u bezúročné půjčky.

Vstupy programu     :  Po výzvě
      Zadej pujcku .....
   na jednom řádku oddělené mezerami
      výše půjčky   měsíční splátka    aktuální měsíc   aktuální rok
Výstupy programu    :  na dvou řádcích pod sebou
      Doba splatnosti je xxx mesicu
      Ukonceni platby xx. mesic roku xxxx
}
var
  POCETMESICU,                { počet měsíců splátky }
```

```
    VYSEPUJCKY,              { výše půjčky v Kčs }
    MESICNISPLATKA,          { měsíční splátka v Kčs }
    MESIC,                   { číslo měsíce, kdy se začalo splácet }
    ROK : integer;           { rok, kdy se začalo splácet }

begin { PUJCKA }
  writeln('VYPOCET DOBY SPLATNOSTI BEZUROCNE PUJCKY');
  writeln;
  writeln('Program vypocte dobu splatek bezurocne pujcky');
  writeln('v mesicich, posledni rok a mesic splatek.');
  writeln;
  writeln('Zadej pujcku v Kcs,mesicni splatku,mesic a rok');
  writeln('pocatku splaceni.');
  writeln('Hodnoty zadej na jednom radku oddelene mezerou.');
  readln( VYSEPUJCKY, MESICNISPLATKA, MESIC, ROK);
  POCETMESICU: = VYSEPUJCKY div MESICNISPLATKA;
  writeln('Doba splatnosti je ',POCETMESICU:3,' mesicu');
  POCETMESICU: = POCETMESICU + MESIC – 1;
  writeln('Ukonceni platby ',POCETMESICU mod 12 + 1 :2,
        '. mesic roku ',POCETMESICU div 12 + ROK:4)
end. { PUJCKA }
```

Program 2.3

2.2 Reálná čísla

Délka, hmotnost a čas, popř. jiné měřitelné fyzikální veličiny jsou běžné pojmy našeho života.

Výsledkem měření však většinou nejsou celočíselné hodnoty, nýbrž hodnoty reálné (teplota těla 36,5° C, vzdálenost 3,25 m, hmotnost brambor 3,5 kg). Chceme-li vytvářet v Pascalu programy pracující s reálnými hodnotami, musíme mít pro reálná čísla k dispozici obdobné vyjadřovací prostředky jako pro práci s celými čísly.

Datový typ reálných čísel se v Pascalu nazývá *real*, a lze tudíž deklarovat proměnné datového typu *real*.

```
var
  V, ALFA: real;
```

Reálná čísla zapisujeme v Pascalu dvojím způsobem. První z nich odpovídá obvyklému desetinnému zápisu reálných čísel a skládá se z celé a desetinné

části. Jak jsme již uvedli, je v Pascalu celá a desetinná část čísla oddělena desetinnou tečkou (nikoli čárkou). Následující příklady jsou správnými zápisy reálných čísel

9.81 –0.256 28753.1 –156.981 10001.001 3.14159

Druhým způsobem zápisu reálného literálu je *semilogaritmický* tvar. Umožňuje snadný zápis hodnot, u nichž by desetinný zápis vyžadoval příliš mnoho nul před nebo za desetinnou tečkou. Semilogaritmický tvar dovoluje bezprostředně za každé celé číslo nebo reálné číslo v desetinném tvaru zapsat *exponent*

Exx E+xx nebo E–xx

vyjadřující násobení původní hodnoty mocninou čísla 10 (*xx* jsou číslice), s významem

10^{xx} (první dva zápisy) nebo 10^{-xx} (třetí zápis).

Tak lze např. nepřehledná a dlouhá reálná čísla v desetinném tvaru

250000000000.0, resp. 0.00000000386

snadno zapsat v semilogaritmickém tvaru jako

25.0E10 2.5E11 0.25E+12 25E+10 250E9,

resp.

38.6E–10 3.86E–9 386.0E–11 386E–11.

Ukažme si praktický příklad použití reálných čísel:

Jak daleko vrhne atlet kouli, provede-li vrh pod úhlem α konstantní počáteční rychlostí v ? Odpor vzduchu zanedbejme.

Je to klasická úloha z fyziky, proto teoretické odvození vztahu pro vodorovnou vzdálenost x šikmého vrhu vzhůru

$$x = (v^2 * \sin(2 * \alpha))/g \qquad (2.3)$$

můžeme snadno nalézt v učebnici mechaniky [11] ($g = 9{,}81$ m s^{-2} je tíhové zrychlení a α úhel mezi směrem vrhu a vodorovnou rovinou).

Hodnotu tíhového zrychlení 9,81 můžeme zapsat např. takto:

9.81 0.981E1 98.1E–1

Pro popis výpočtu hodnoty podle vztahu (2.3) použijeme reálného výrazu. Některé operátory pro operace reálného výrazu jsou vyjádřeny stejnými symboly jako u celočíselného výrazu a mají i stejnou prioritu:

+	identita, resp. sčítání,
–	změna znaménka, resp. odečítání,
*	násobení.

Navíc je zde operátor (s prioritou stejnou jako má **div**)

/ dělení.

Pascal *nemá* operátor pro umocňování x^y.

Často užívanou součástí reálných výrazů jsou goniometrické, popř. jiné funkce. V Pascalu je funkce předpis pro výpočet hodnoty. Funkce je pojmenována a může mít jeden nebo více parametrů (argumentů). Například funkce abs(*X*) má jeden parametr *X* a je to předpis pro výpočet absolutní hodnoty čísla.

Je-li *X* nezáporné, pak je hodnotou funkce abs. Je-li záporné, je hodnotou funkce $-X$.

Pascal poskytuje řadu standardních funkcí:

sqrt(X)	druhá odmocnina z *X*
sqr(X)	druhá mocnina *X*
sin(X)	sin *X*
cos(X)	cos *X*
arctan(X)	arctg *X*
ln(X)	přirozený logaritmus *X*
exp(X)	exp *X*
abs(X)	absolutní hodnota *X*

Zápis funkcí v Pascalu je pro nás obvyklý, tj. jméno funkce následované argumentem (argumenty) v závorkách (X je reálný výraz). Zdůrazněme, že argumenty funkcí je v Pascalu nutné vždy (na rozdíl od matematického zápisu) uzavírat do závorek. Zápis funkce ve výrazu nazýváme také *voláním funkce*.

Výraz pro výpočet vodorovné vzdálenosti šikmého vrhu vzhůru bude mít pak tento tvar:

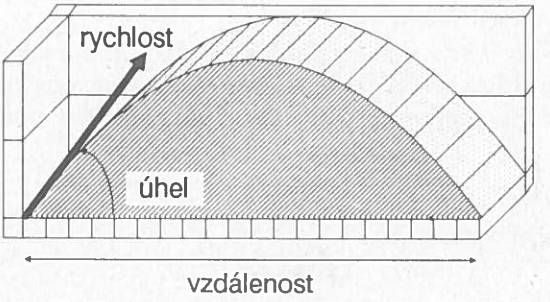
sqr(V) * *sin* (2 * ALFA) / 9.81.

Úhly jsou zadávány i počítány v radiánech (nikoliv ve stupních). Vzájemný převod je snadný: Nechť *s* je hodnota úhlu ve stupních (úhel se zadává ve stupních jako desetinné číslo, nelze zadávat stupně, minuty a vteřiny) a *r* v radiánech. Pak

$$s/180.0 = r/3.141592654.$$

Rozsah hodnot datového typu *real* je omezen vlastnostmi počítače a my budeme předpokládat, že je pro naše potřeby vždy dostatečně velký. Vzhledem k omezenému prostoru pro zobrazení reálné hodnoty v paměti počítače dochází často k zaokrouhlování reálných hodnot na posledních platných místech.

Vstup reálných hodnot se provádí stejným příkazem *read* , resp. *readln* jako u hodnot celočíselných, pouze jako parametry příkazu jsou uvedeny proměnné typu *real*. Hodnoty jsou na vstupu zapisovány ve stejném tvaru jako reálné nebo celočíselné literály v programu. Do reálných proměnných lze číst i celočíselné hodnoty.



Obr.2.2 Šikmý vrh vzhůru

Neformátovaný výstup reálných hodnot se zadává obdobným způsobem jako u celých čísel, tj. příkazem *write* nebo *writeln*. Je-li parametrem těchto příkazů reálný výraz, je na výstupním zařízení zobrazeno reálné číslo v semilogaritmickém tvaru. Počet znaků je dán vlastnostmi překladače. Pro standardní délku neformátovaného tvaru (např. 16 znaků) by měl výstup hodnoty tíhového zrychlení z našeho příkladu tvar

9.810000000E+00

Podobně jako u celých čísel, můžeme i zde zadávat formát výstupních hodnot, a to jak pro desetinný, tak i pro semilogaritmický tvar.

Formát semilogaritmického tvaru je týž, jako pro celá čísla, tj.

writeln(*výraz* : *počet znaků*)

Dvojtečkou je od reálného výrazu oddělen výraz celočíselný, udávající *počet znaků*, na něž se semilogaritmický tvar na výstupním zařízení vypíše, včetně místa pro znaménko, desetinné tečky, znaku E, znaménka exponentu a exponentu samotného. Exponent se obvykle zobrazuje dvěma platnými

číslicemi. Je-li hodnota výrazu kladná, je prvním zobrazeným znakem mezera (nikoliv +), je-li záporná, je prvním zobrazeným znakem –.

Formát desetinného tvaru je složitější

writeln(*výraz* : *počet znaků* : *délka desetinné části*)

První celočíselný výraz udává opět celkovou délku (*počet znaků*) výpisu hodnoty. Za další dvojtečkou je uvedena *délka desetinné části* čísla (rovněž celočíselný výraz). Do celkové šířky se započítává i místo pro znaménko (podobně jako u semilogaritmického tvaru) a desetinnou tečku. Je-li požadovaná šířka větší než počet platných číslic vystupující hodnoty, je vystupující hodnota doplněna na zadanou délku mezerami zleva.

Výstup hodnoty 9.81 v semilogaritmickém tvaru na 9 znaků (místo pro znaménko, jedna číslice před desetinnou tečkou, desetinná tečka, dvě číslice za desetinnou tečkou, písmeno E, znaménko exponentu a dvě číslice exponentu) zapíšeme

writeln (9.81:9)

a jemu odpovídající výstup bude

9.81E+00

Výstup téže hodnoty v desetinné notaci na 5 znaků (místo pro znaménko, jedna číslice před desetinnou tečkou, desetinná tečka a dvě číslice za desetinnou tečkou) zapíšeme

writeln(9.81:5:2)

a výstup bude

9.81

Pokud je formát zadán tak, že v něm vypisovanou hodnotu nelze zobrazit, provede se výstup v žádaném tvaru na minimální nutný počet znaků.

Nyní již můžeme napsat program pro výpočet vodorovné vzdálenosti dopadu při vrhu koulí (šikmého vrhu vzhůru):

```
program DELKA (input, output);
{ Program provádí výpočet vodorovné vzdálenosti dopadu x šikmého
vrhu vzhůru podle 2.3
  Vstup  :  Po výzvě

        Zadej rychlost a uhel ...
```

```
      na jeden řádek rychlost v metrech za sekundu a úhel ve stupních
      (oddělit mezerou). Úhel se zadává jako desetinné číslo (nelze
      zadat stupně, minuty a vteřiny)

   Výstup:
      Delka vrhu je xxx.xx m.
}
var
   V,                  { rychlost }
   ALFA: real;         { úhel }
begin { DELKA }
   writeln('VYPOCET VODOROVNE VZDALENOSTI DOPADU');
   writeln('PRI SIKMEM VRHU VZHURU');
   writeln;
   writeln('Vstupnimi hodnotami je rychlost vrhu');
   writeln('v metrech za sekundu a uhel vrhu');
   writeln('ve stupnich (jako desetinne cislo).');
   writeln;
   writeln('Zadej rychlost a uhel na jeden radek');
   writeln('oddelene mezerou:');
   readln(V,ALFA); { čtení vstupních hodnot }
   ALFA := (ALFA * 3.141596536) / 180.0;
           { převod stupňů na radiány }
   writeln('Delka vrhu je ',sqr(V) * sin(2.0 * ALFA)/ 9.81:7:3,' m.');
end. { DELKA }
```

Program 2.4

Víme již, že se v programu vyskytují objekty, jejichž hodnota se v průběhu výpočtu může měnit (proměnné), i objekty, jejichž hodnota je v čase konstantní (literály). Literály jsou vhodné v případě, kdy je konstantní hodnota využita jednorázově, není nutné ji pojmenovat a je jisté, že se nebude v průběhu ladění programu měnit (např. počet měsíců v roce v programu PUJCKA).

V situaci, kdy konstantní hodnota má své tradiční jméno (např. ve vzorcích π, e, g ap.), popř. se v programu vyskytuje vícekrát, nebo se předpokládá její modifikace, je výhodné ji označit identifikátorem. Takovou hodnotu nazýváme v Pascalu *pojmenovanou konstantou*. Pojmenované konstanty a literály budeme souborně nazývat *konstantami*. Pojmenovaná konstanta nevyjadřuje přímo svým zápisem hodnotu.

Identifikátory pojmenovaných konstant či jiných objektů označených řec-

kými písmeny (nebo jinými znaky, které nejsou v textu pascalského programu povoleny) můžeme uvést ve fonetickém přepisu, např. π nahradíme identifikátorem PI.

V deklarační části před proměnnými se v případě použití konstant vyskytuje *deklarace konstant* ve tvaru

```
const
  identifikátor1 = konstanta1;
  identifikátor2 = konstanta2;
  :
  identifikátorn = konstantan;
```

Konstantou na pravé straně znaménka = může být literál, popř. identifikátor již dříve deklarované pojmenované konstanty. Například

```
const
  POCETPOLOZEK  = 5;
  PI            = 3.141592654;
  G             = 9.81;
  ZAHLAVI1 =
    'VYPOCET VODOROVNE VZDALENOSTI DOPADU';
  ZAHLAVI2 = 'PRI SIKMEM VRHU VZHURU';
```

Zde přiřadíme pojmenované konstantě jméno (konstanty lze tedy deklarovat pouze pro takové datové typy, které umožňují zápis literálů) a v dalším programu pak místo přímého zápisu hodnoty literálem používáme deklarovaného identifikátoru. Kromě přehlednosti má tento způsob zápisu výhodu v tom, že deklarace pojmenovaných konstant (a jejich skutečných hodnot) je soustředěna *na jedno místo*. Takový tvar programu usnadňuje jeho modifikaci.

Jazyk Pascal poskytuje standardní pojmenovanou konstantu *maxint* udávající maximální možnou celočíselnou hodnotu, se kterou můžeme v programu pracovat (říkali jsme, že rozsah celých čísel je omezen). Máme tedy k dispozici celočíselné hodnoty v intervalu (viz též str. 20)

$$< -maxint, maxint >.$$

Aritmetické operace +, − a * mají stejné operátory pro zápis celočíselného i reálného výrazu. Můžeme rovněž zapisovat do jediné operace jeden z operandů celočíselný a druhý reálný. Pro určení typu výsledku platí následující pravidla.

Pokud jsou všechny operandy (tj. jeden u operací s jedním operandem, oba u operací se dvěma operandy) celočíselné, je i výsledný výraz celočíselný. Pokud je alespoň jeden z operandů reálný, je i výsledný výraz reálný.

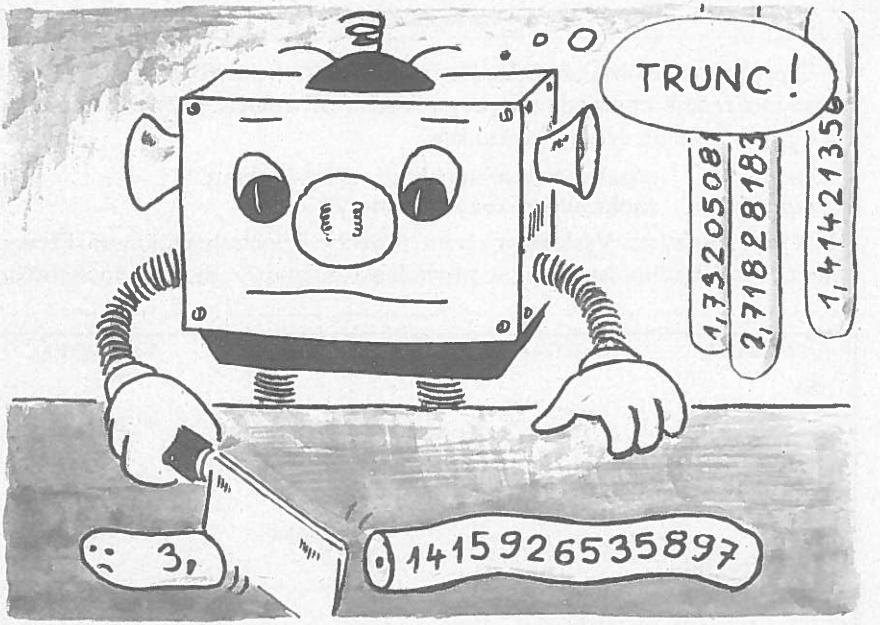
Z předchozího plyne, že nelze (bez použití dalších prostředků) zapsat výraz obsahující reálné operandy tak, aby výsledek byl celočíselný. Aby to možné bylo, jsou v Pascalu definovány funkce

trunc(X) odseknutí desetinné části reálné hodnoty *X*,
round(X) zaokrouhlení reálné hodnoty *X*,

kde *X* je reálný výraz. Výsledkem těchto funkcí je celočíselná hodnota. Převod celočíselných hodnot na reálné se provádí automaticky zápisem celočíselného

OPERÁTOR	1. OPERAND	2. OPERAND	VÝSLEDEK
div			
	integer	integer nenulové	integer
mod			
	integer	integer pouze kladné	integer
+ (identita) – (změna znaménka)			
	integer	–	integer
	real	–	real
– (odčítání) * (násobení)			
	integer	integer	integer
	real	real	real
	integer	real	real
	real	integer	real
/ (dělení)			
	integer	integer	real
	real	real	real
	integer	real	real
	real	integer	real
sqrt, sqr, sin, cos, arctan, ln, exp			
	integer	–	real
	real	–	real
trunc, round			
	real	–	integer
abs (abs.hodnota)			
	integer	–	integer
	real	–	real

Tab. 2.1 Typy operandů a výsledků



operandu do operace s dalším operandem reálným. Opačný převod je nutné provést jednou z uvedených funkcí *trunc*, resp. *round*.

Podobně u přiřazovacího příkazu je možné přiřadit reálné proměnné celočíselnou hodnotu, která bude převedena na reálnou (ne však naopak).

V Pascalu tedy v jednom výrazu mohou existovat operandy různých datových typů i operace, které mají různý typ operandů i výsledku. Proto je nutné u každé operace i funkce znát možné typy operandů a v závislosti na nich i typ výsledku. Operace a funkce, které jsme dosud poznali, jsou uvedeny v tab. 2.1. U všech dalších operací i funkcí budeme vždy typ operandů i výsledku uvádět.

Článek uzavřeme druhou variantou příkladu DELKA pro výpočet vodorovné vzdálenosti šikmého vrhu vzhůru. Jsou v něm deklarovány pojmenované konstanty PI a G pro vyjádření Ludolfova čísla π a tíhového zrychlení g. Rovněž je použit smíšený výraz obsahující celočíselné i reálné hodnoty.

```
program DELKA (input, output);
{ Úvodní poznámka je totožná s programem 2.4 }

const
  PI = 3.141596536;    { Ludolfovo číslo }
  G = 9.81;            { tíhové zrychlení }
  ZAHLAVI1 =
    'VYPOCET VODOROVNE VZDALENOSTI DOPADU';
  ZAHLAVI2 = 'PRI SIKMEM VRHU VZHURU';
var
  V,                   { rychlost }
  ALFA: real;          { úhel }
begin { DELKA }
  writeln(ZAHLAVI1);
  writeln(ZAHLAVI2);
  writeln;
  writeln('Vstupnimi hodnotami je rychlost vrhu');
  writeln('v metrech za sekundu a uhel vrhu');
  writeln('ve stupnich (jako desetinne cislo).');
  writeln;
  writeln('Zadej rychlost a uhel na jeden radek');
  writeln('oddelene mezerou:');
  readln(V,ALFA);                    { čtení vstupních hodnot }
  ALFA := (ALFA * PI) / 180;         { převod stupňů na radiány }
  writeln('Delka vrhu je ',sqr(V) * sin(2 * ALFA) / G:7:3,' m.');
end. { DELKA }
```

Program 2.5

2.3 Cvičení

U všech programů nezapomínejte na vhodné vysvětlující texty vypisované na obrazovku.

1. Vytvořte program, který po spuštění vypíše hodnoty 64, 128 a 256 pod sebe. Hodnoty budou začínat od 10. sloupce.

   Návod: Chceme-li vypisovat hodnoty od 10. sloupce obrazovky, musíme na začátek každého řádku zapsat 9 mezer následovaných významnými znaky čísla. Můžeme toho dosáhnout příkazem

   writeln(64:11)

tj. výpisem celého čísla na 11 znaků (9 úvodních mezer a 2 znaky čísla).

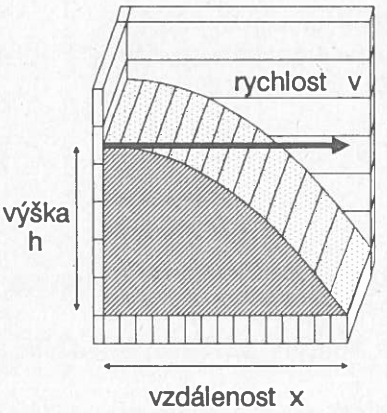
2. Vypište hodnoty z příkl. 1 na jeden řádek vedle sebe tak, aby čísla začínala ve sloupcích 10, 20 a 30.
3. Modifikujte příkl. 1 a 2 tak, aby vypisované hodnoty byly čteny ze vstupu.
4. Vytvořte program, který přečte dvě hodnoty a a b ze vstupu a vypíše jejich součet, rozdíl, součin, podíl, a **mod** b, změnu znaménka a a změnu znaménka b. Nakonec vypočte a vytiskne hodnotu výrazu

$$243 * ((1987 \textbf{ mod } a) + b).$$

5. Vytvořte program, který pro danou hodnotu v Kčs vytvoří výčetku (tj. počet 1000 Kčs, 500 Kčs, ... , 1 Kčs) s co nejmenším počtem bankovek a mincí.
6. Určete, jakého výsledného datového typu jsou následující výrazy

2 / 3 + 4,	5 – 4.6,
2 **div** 3 + 4,	–12.0 + 3,
sin (*round* (2.8) * 2.0),	2 * 8.

7. Vytvořte program pro výpočet délky přepony pravoúhlého trojúhelníka
   - podle Pythagorovy věty $a^2 = b^2 + c^2$ z délek odvěsen,
   - využitím funkcí sin a cos z délky jedné odvěsny a úhlu. Vstupní úhly zadávejte jak ve stupních, tak v radiánech.



Obr.2.3 Vodorovný vrh

8. Vytvořte program pro výpočet vzdálenosti dopadu vodorovného vrhu konstantní počáteční rychlostí v z výšky h (obr. 2.3) podle vztahu

$$x = v \sqrt{[(2h / g)]}.$$

9. Modifikujte program PUJCKA tak, aby prováděl výpočet s hodnotou půjčky v datovém typu *real* (tj. v korunách i haléřích).
10. Míč vyhozený směrem vzhůru s počáteční rychlostí v bude za čas t ve výšce

$$h(t) = -0{,}5\, gt^2 + vt$$

($g = 9{,}81\ \mathrm{m\,s^{-2}}$). Vytvořte program pro výpočet dosažené výšky v čase t při zadané počáteční rychlosti v.

3 LOGICKÉ HODNOTY

O každých dvou hodnotách datových typů *integer* a *real* je možné rozhodnout, zda

- jsou stejné,
- nejsou stejné,
- první je menší než druhá,
- první je menší, nebo jsou obě stejné,
- první je větší než druhá,
- první je větší, nebo jsou obě stejné.

Porovnávání dvou hodnot uspořádaného datového typu je častá činnost v programech. Proto pro všechny datové typy, jejichž hodnoty jsou uspořádány (prozatím známe *integer* a *real*), jsou definovány operace pro porovnání dvou hodnot. Porovnání dvou hodnot podle daného předpokladu provádíme jedním z těchto šesti zápisů:

*výraz*1	=	*výraz*2
*výraz*1	< >	*výraz*2
*výraz*1	<	*výraz*2
*výraz*1	< =	*výraz*2
*výraz*1	>	*výraz*2
*výraz*1	> =	*výraz*2

Operátory =, < >, <, < =, >, > = určují postupně porovnání na rovnost, nerovnost, menší, menší nebo rovno, větší, větší nebo rovno a říkáme jim *relační operátory*.

Pokud I, J jsou proměnné typu *integer* a A, B proměnné typu *real*, lze vytvářet např. tyto zápisy

I < J	I > 1	I < = J + 3
A > I	A > 2.35	A < > B * 2E–8

Je tedy možné navzájem porovnávat i reálné a celočíselné hodnoty (např. zápisem A > I).

Priorita relačních operátorů je nejnižší. Proto je vždy nejprve provedeno vyhodnocení porovnávaných výrazů a nakonec vlastní porovnání.

Výsledkem porovnání je výrok, zda je tvrzení o vzájemném vztahu dvou výrazů pravdivé nebo nepravdivé. Jinak řečeno, výsledkem porovnání mohou být dvě hodnoty, a to pravda (relace je splněna), nebo nepravda (relace není splněna). V angličtině jsou tyto dvě hodnoty nazvány *true* (true = pravdivý) a *false* (false = nepravdivý).

Datový typ mající pouze dvě hodnoty *true* a *false* je často v programech využíván nejen jako výsledek porovnání. Byl proto zařazen mezi standardní datové typy a pojmenován identifikátorem *boolean* (na počest matematika George Boolea). Hodnotám a proměnným datového typu *boolean* říkáme *logické hodnoty* a *proměnné* a výrazům *logické výrazy*. Příklad deklarací logických proměnných B1 a B2 může být tento :

```
var
  B1, B2: boolean;
```

Existují dva standardní identifikátory (pojmenované konstanty) *false* a *true* reprezentující hodnoty datového typu *boolean*. Platí

false < *true*

Je možné deklarovat též další logické pojmenované konstanty typu *boolean*

```
const
  PRAVDA = true;
  LEZ    = false;
```

Můžeme tedy zapsat

```
B1  := true;
B2  := LEZ;
```

Vzhledem k tomu, že porovnání má výsledek typu *boolean*, můžeme napsat

```
B1  := A <> 2E-8;
B2  := I < J;
```

Logické hodnoty lze vypisovat (obdobně jako číselné hodnoty) příkazem *write* nebo *writeln*, jehož parametrem je logický výraz. Má-li tento výraz hodnotu *true*, je vypsáno TRUE (tj. čtyři znaky). Má-li hodnotu *false*, je vypsáno FALSE (tj. pět znaků). Výstup lze formátovat obvyklým způsobem. Potom se řetězec znaků TRUE nebo FALSE doplní zleva mezerami na požadovanou šířku. Je-li šířka malá, vystoupí jen počáteční znaky. Je nutné podotknout, že takový výstup logických hodnot není příliš výhodný a užívá se zřídka.

Pro užití v logickém výrazu jsou definovány tři operace s operátory **not** (negace), **and** (logický součin) a **or** (logický součet).

Nejvyšší prioritu má operace **not**, následuje **and** s prioritou stejnou jako násobení. Nejnižší prioritu má **or** (stejnou jako sčítání).

Vzhledem k tomu, že ve výrazu lze zapisovat operandy různých typů společně, je v tab. 3.1 uvedena priorita všech operátorů počínaje nejvyšší.

PRIORITA	OPERÁTOR
4	**not**
3	*, /, **div, mod, and**
2	+, −, **or**
1	=, < >, >, <, < =, > =

Tab.3.1 Priorita operátorů výrazu

X	Y	X **and** Y	X **or** Y	**not** X
false	*false*	*false*	*false*	*true*
false	*true*	*false*	*true*	*true*
true	*false*	*false*	*true*	*false*
true	*true*	*true*	*true*	*false*

Tab.3.2 Operace and, or a not

Definice logických výrazů pro operace **not, and** a **or** je v tab. 3.2.

Častým zápisem logického výrazu je kombinace relací a logických operací **not, and** a **or**. Například test, zda proměnná CISLO má hodnotu v intervalu < 0,9 >, se zapisuje takto:

(CISLO > = 0) **and** (CISLO < = 9)

Je-li třeba provádět porovnání *před* jinou operací výrazu, je *nutné* využít závorek. Zápis bez závorek

CISLO > = 0 **and** CISLO < = 9

je chybný, neboť zde má operace **and** celočíselné operandy, což není přípustné. Úvodní část výrazu bude překladačem interpretována jako

CISLO > = (0 **and** CISLO)

a zápis bude označen jako chyba.

Vzhledem k tomu, že existuje uspořádání logických hodnot

false < *true*

je možné relační operátory užít i pro logické hodnoty. Tím získáme další logické operace. Například použijeme–li pro logické operandy

= obdržíme *ekvivalenci*,

< > obdržíme *nonekvivalenci* a pro

< = obdržíme *implikaci*.

Pro logické proměnné

```
var
  VYROK1, VYROK2 : boolean;
```

reprezentující pravdivost dvou výroků lze zapsat výraz

VYROK1 < = VYROK2

který vyjadřuje tvrzení

VYROK1 *implikuje* VYROK2.

3.1 Cvičení

1. Vytvořte program, který přečte ze vstupu 2 reálné hodnoty a provede s nimi operace porovnání =, < >, <, >, > = a < =. Výsledek těchto 6 operací vypíše na výstupu.

4 ŘÍZENÍ

Všechny programy, kterými jsme se doposud zabývali, měly jednu společnou vlastnost. Prováděly výpočet daný jedinou možnou posloupností příkazů tak, jak byly příkazy do programu zapsány. Takových programů je málo a bývají jednoduché. Většina programů využívá možnosti větvit posloupnost prováděných příkazů na základě hodnot vypočtených v době provádění programu.

Konstrukce jazyka, které umožňují takové větvení v programu popsat, popř. i jinak řídit posloupnost provádění příkazů, nazýváme *řídicí struktury*.

4.1 Větvení

4.1.1 Podmíněný příkaz

Každý z nás měl pravděpodobně v ruce klíč k určování rostlin. Je to příručka, která na základě kladených otázek a našich odpovědí umožňuje blíže určit nalezenou rostlinu. Struktura klíče je obvykle tato:

```
vyber si z těchto možností      – rostliny výtrusné,
                                – rostliny semenné;
jde-li o rostliny výtrusné, pak:
  vyber si z těchto možností      – lodyhy článkované,
                                  – lodyhy nečlánkované;
  jde-li o lodyhy článkované, pak:
      jde o přesličky
  jinak (lodyhy nečlánkované):
      vyber si z těchto možností  – nadzemní stonky vidličnatě
          větvené, husté drobné úzké listy, výtrusnice v konečných
          klasech nebo jednotlivě v paždí listů,
                                  – z oddenku vyrůstá trs plochých
          listů, jednoduše až mnohonásobně zpeřených, výtrusnice
          na spodu listů nebo odděleně, ale uspořádány v laty;
      jde-li o nadzemní stonky atd., pak:
          jde o plavuně,
      jinak (z oddénku vyrůstá atd.)
          jde o kapradiny;
jinak (rostliny semenné):
```

vyber si z těchto možností — dřeviny,
— byliny a polokeře;
jde-li o dřeviny, pak:
.
.
atd.

Algoritmus určování je poměrně jednoduchý. Vždy jsou předloženy dvě nebo více vlastností rostliny. V našem případě jde o klíč dichotomický, proto jsou na výběr vždy pouze dvě možnosti. Je zaručeno, že rostlina nemůže mít obě vlastnosti zároveň.

Na základě pravdivosti či nepravdivosti tvrzení, že rostlina má danou vlastnost, zvolíme jednu ze dvou možností. Celý postup se opakuje tak dlouho, dokud není daná rostlina určena.

Přesto, že jde o algoritmus, nemáme dosud k dispozici prostředek, tj. jazykovou konstrukci s významem:

Platí-li podmínka, pak
proveď variantu$_1$ algoritmu,
jinak
proveď variantu$_2$ algoritmu.

Tedy na základě pravdivosti či nepravdivosti podmínky (hodnoty logického výrazu) se má provést jedna ze dvou variant algoritmu.

Jazyk Pascal poskytuje pro zápis větvení konstrukci nazvanou *podmíněný příkaz*

if *logický výraz* **then**
příkaz$_1$
else
příkaz$_2$

V této konstrukci je podmínka vyjádřena *logickým výrazem*. Výraz nabývá jedné z hodnot *true* nebo *false*. Je-li *logický výraz* pravdivý, je proveden *příkaz*$_1$. Jinak je proveden *příkaz*$_2$. Vyhrazená slova **if**, **then** a **else** vyjadřují v angličtině totéž, co v češtině slova *jestliže*, *pak* a *jinak*.

Například v podmíněném příkazu

```
if I > 0 then
   I := I - 1
else { I < = 0 }
   I := I + 1
```

je provedeno větvení programu na základě hodnoty celočíselné proměnné I.

Je-li její hodnota kladná, je proveden příkaz I: = I – 1, jinak je proveden příkaz I: = I + 1.

Rozhodování na základě složitějšího logického výrazu může mít např. tvar

```
if (CISLO > =0) and (CISLO < =9) then { 0 < = CISLO < = 9 }
   writeln('Hodnota ', CISLO,' je v intervalu <0,9>')
else { mimo interval }
   writeln('Hodnota ', CISLO,' neni v intervalu <0,9>')
```

U hodnoty celočíselné proměnné CISLO se zjišťuje, zda leží v daném intervalu. Výrok o příslušnosti hodnoty do daného intervalu je potom vypsán příkazem *writeln*.

Každou z variant podmíněného příkazu může být pouze *jediný příkaz*. Doposud jsme poznali příkaz přiřazovací, příkazy vstupu a výstupu a nyní nově příkaz podmíněný. Pokud bychom mohli použít jako variantu větvení pouze tyto příkazy, byl by podmíněný příkaz obtížně použitelný.

Proto Pascal poskytuje jazykovou konstrukci, která dovoluje zapsat do programu na místo jednoho příkazu celou posloupnost příkazů. Tato konstrukce se nazývá *složený příkaz*.

Složený příkaz má tvar

```
begin
   příkaz1;
   příkaz2;
   příkaz3;
   :
   příkazn
end
```

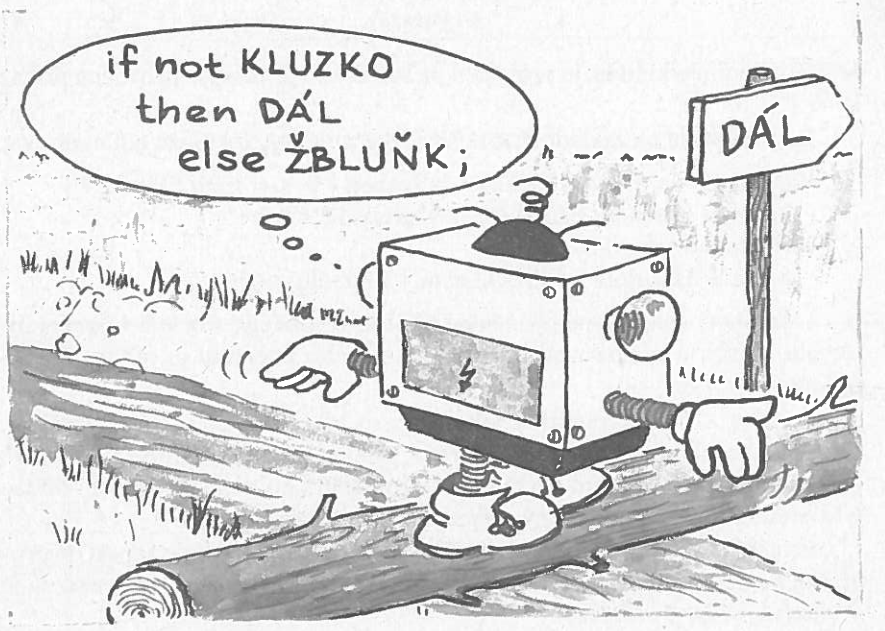
Vidíme, že je to několik příkazů za sebou oddělených středníky. Posloupnost je uzavřena do příkazových závorek **begin** a **end**.

Složený příkaz je možno zapsat v programu tam, kde je dovoleno zapsat jediný příkaz (tedy i do podmíněného příkazu). Příkazy ve složeném příkazu jsou prováděny v pořadí, v jakém jsou za sebou zapsány.

Počet příkazů v posloupnosti není omezen. Může zde být příkaz jediný, popř. nemusí být mezi **begin** a **end** žádný příkaz.

Poznamenejme též, že příkazová část programu je tvořena jedním složeným příkazem.

Zapišme nyní část programu, ve které se provádí určení ceny lístku v kině. Lístek v 1. řadě stojí 7 Kčs, ve 2. – 6. řadě 8 Kčs a v 7. – 15. řadě 10 Kčs. Využijeme podmíněného příkazu (RADA je celočíselná proměnná, jejíž hod-



nota udává řadu sedadel a CENA je celočíselná proměnná ceny lístku):

```
if RADA = 1 then
   CENA := 7  { 1. řada }
else if (RADA >= 2) and (RADA <= 6) then
   CENA := 8  { 2.–6. řada }
else if (RADA >= 7) and (RADA <= 15) then
   CENA := 10    { 7.–15. řada }
else { jiná řada }
   writeln ('Tato rada v kine neni');
```

V tomto podmíněném příkazu je *příkazem*$_1$ příkaz označený komentářem { 1. řada } a *příkazem*$_2$ opět podmíněný příkaz. Tento vnořený podmíněný příkaz má opět dvě varianty. Jednak příkaz označený komentářem { 2.–6. řada } jako první variantu a jako druhou variantu další podmíněný příkaz. Ten má jako první variantu příkaz označený komentářem { 7.–15. řada } a jako druhou variantu má příkaz výstupu oznamující, že hodnota proměnné RADA udává řadu, která v kině není.

Vidíme, že variantou podmíněného příkazu může být opět podmíněný příkaz. Tomuto zápisu říkáme *vnoření podmíněných příkazů* a lze jím realizovat i poměrně složité rozhodovací algoritmy.

Vraťme se nyní k našemu příkladu s klíčem k určování rostlin. Pokusme se pomocí podmíněného příkazu realizovat část takového klíče jako program, který formou dialogu s uživatelem nahrazuje experta v oboru botaniky (program 4.1).

Rozhodovací proces našeho programu je tvořen podmíněnými příkazy a rozhodování se provádí na základě komunikace s uživatelem. Program zadává uživateli možnosti pomocí příkazů *writeln* a požaduje odpověď číslem. Z prostorových důvodů jde pouze o malou část expertních znalostí z botanického klíče. Lze ji však snadno rozšiřovat až na potřebnou velikost.

```
program EXPERT (input, output);
{ Konverzační program umožňující určování rostlin metodou botanického
klíče
Vstup a výstup:
   Na obrazovku se vypíší vždy dvě vlastnosti rostliny ve tvaru

   1 – popis vlastnosti 1,
   2 – popis vlastnosti 2.

   Uživatel odpoví znakem 1 nebo 2 (volba vlastnosti). Je-li určena
   příslušná rostlina (zde pouze třída nebo jiná skupina rostlin), je vypsán
   výsledek hledání a program se ukončí.
}
var
   ODPOVED: integer;          { odpověď na otázku }

begin { EXPERT }
   writeln('KLIC K URCOVANI ROSTLIN');
   writeln;
   writeln('Zvol vzdy jednu z variant zadanim znaku 1 nebo 2');
   writeln('1 – rostliny vytrusne');
   writeln('2 – rostliny semenne');
   readln(ODPOVED);
   if ODPOVED = 1 then
      begin { rostliny výtrusné }
```

```
        writeln('1 – lodyhy clankovane');
        writeln('2 – lodyhy neclankovane');
        readln(ODPOVED);
        if ODPOVED = 1 then { lodyhy článkované – PŘESLIČKY }
          writeln(' * PRESLICKY – Equisetopsida')
        else
          begin { lodyhy nečlánkované }
            writeln('1 – nadzemni stonky vidlicnate vetvene');
            writeln('   huste,drobne,uzke listy, vytrusnice');
            writeln('   v konecnych klasech nebo jednotlive');
            writeln('   v pazdi listu, sporofyly skupinovite');
            writeln('2 – z oddenku vyrusta trs plochych listu');
            writeln('   jednoduse az mnohonasobne zperenych');
            writeln('   vytrusnice na spodu listu nebo oddelene');
            writeln('   ale usporadany v laty');
            readln(ODPOVED);
            if ODPOVED = 1 then { PLAVUNĚ }
              writeln('PLAVUNE – Lycopodiopsida')
            else { KAPRADINY }
              writeln('KAPRADINY – Pteridopsida')
          end { lodyhy nečlánkované }
      end { rostliny výtrusné }
    else
      begin { rostliny semenné }
        writeln('ROSTLINY SEMENNE');
        { a tak dále }
      end; { rostliny semenné }
  end. { EXPERT }
```

Program 4.1

Doposud jsme uvažovali podmíněný příkaz ve tvaru, kdy při pravdivosti logického výrazu se prováděla jedna varianta a při nepravdivosti druhá varianta. Často však chceme vyjádřit v algoritmu i situaci, kdy při pravdivosti logického výrazu se provede jistý příkaz, avšak při nepravdivosti *se neprovádí žádný příkaz*. Pro vyjádření takového algoritmu existuje zápis

if *logický výraz* **then**
  příkaz

kterému říkáme *neúplný podmíněný příkaz*. Dosud užívanému podmíněnému

příkazu říkáme pak *úplný*. Jak je vidět, neúplný podmíněný příkaz (na rozdíl od úplného) neobsahuje variantu

```
else
  příkaz2
```

Potom zápis

```
if (RADA < 1) and (RADA > 15) then
  writeln('V nasem kine tato rada neexistuje')
```

vyjadřuje podmíněné provedení příkazu výstupu. Provedení je podmíněno pravdivostí logického výrazu (RADA < 1) **and** (RADA > 15). Není-li podmínka splněna, neprovádí se žádný příkaz.

Zavedením neúplného podmíněného příkazu vzniká problém. Vzhledem k tomu, že variantou podmíněného příkazu může být opět podmíněný příkaz, je možné zapsat konstrukci

if *logický výraz*$_1$ **then if** *logický výraz*$_2$ **then** *příkaz*$_1$ **else** *příkaz*$_2$

V této konstrukci není zřejmé, zda varianta

else *příkaz*$_2$

přísluší k prvnímu nebo k druhému **then** (neboli není zřejmé, který ze dvou podmíněných příkazů je úplný a který neúplný). Předchozí zápis je možno chápat jako

```
if logický výraz1 then
  begin
    if logický výraz2 then
      příkaz1
    else
      příkaz2
  end
```

popř.

```
if logický výraz1 then
  begin
    if logický výraz2 then
      příkaz1
  end
else
  příkaz2
```

V Pascalu je obdobná kombinace podmíněných příkazů vždy chápána jako první z uvedených možností, tj. varianta **else** přísluší vždy k nejbližšímu předchozímu **then**, které ještě nemá variantu **else**. Při užívání složitějších vnořených podmíněných příkazů je výhodnější užít příkazových závorek **begin** a **end** pro ohraničení neúplného podmíněného příkazu. Vyvarujeme se tím těžko odhalitelných chyb.

Odbočme nyní od hlavní náplně této kapitoly a ujasněme si problém zápisu středníků oddělujících příkazy, tj. kde se středník zapsat *musí* a kde *může*.

Uvažujme složený příkaz

begin
  příkaz$_1$;
  příkaz$_2$;
  :
  příkaz$_{n-1}$;
  příkaz$_n$
end

Mezi *příkazem*$_n$ a vyhrazeným slovem **end** definice složeného příkazu nedovoluje zapisovat středník. To je dosti nevýhodné, neboť ve většině případů zápisu programu se středník za příkazem zapisuje a mohlo by snadno dojít k chybnému zápisu. Pascal proto poskytuje pomocný prostředek, který umožňuje obě varianty zápisu (se středníkem i bez středníku za posledním *příkazem*$_n$ ve složeném příkazu).

Tímto prostředkem je *prázdný příkaz*. Prázdný příkaz neobsahuje ve svém zápisu *žádný znak*. Může tedy být všude tam, kde mohou být ostatní příkazy. Je možné napsat např.

if I > 0 **then** I: = I + 1 **else** { zde je prázdný příkaz }

s významem

if I > 0 **then** I: = I + 1

Jiná možnost zápisu s prázdným příkazem ve variantě úplného podmíněného příkazu je

if I > 0 **then** { zde je prázdný příkaz } **else** I: = I + 1

s významem

if I < = 0 **then** I: = I + 1

Zapíšeme-li prázdný příkaz do posloupnosti složeného příkazu, budou dva středníky vedle sebe

```
begin
  I:=I+1;
  { zde je prázdný příkaz };
  writeln(I)
end
```

V případě, že prázdný příkaz je v posloupnosti příkazů složeného příkazu *poslední*, objeví se středník před vyhrazeným slovem **end**.

```
begin
  I:=I+1;
  writeln(I);
  { zde je prázdný příkaz }
end
```

Ve složeném příkazu se tedy středník před vyhrazeným slovem **end** (vzhledem k možnému výskytu prázdného příkazu) zapisovat *může*, ale nemusí. V poznámkách k metodice uvedeme, proč je výhodné středník před **end** psát vždy.

Jednotlivé příkazy posloupnosti složeného příkazu *musí* být (jak plyne z předchozího) odděleny alespoň jedním středníkem.

Jiná situace je u úplného podmíněného příkazu mezi vyhrazenými slovy **then** a **else**. Zde může být zapsán *jediný* příkaz bez středníku. Nelze sem vložit navíc příkaz prázdný, neboť pak by zde byly příkazy *dva*. Proto před **else** středník stát *nesmí*.

Úplný podmíněný příkaz rozděluje algoritmus do dvou větví. Požadujeme-li větvení do více větví, zapisujeme na místo *příkazu*$_2$ další podmíněný příkaz atd. To můžeme opakovat tak dlouho, dokud nevytvoříme potřebný počet větví. Například příkaz

```
if I < 0 then { záporné }
  writeln ('Hodnota je zaporna')
else if I = 0 then { nulové }
  writeln ('Hodnota je nulova')
else { kladné }
  writeln ('Hodnota je kladna')
```

větví algoritmus do tří variant pro I záporné, nulové a kladné. Dvojice vyhrazených slov **else if** uvozuje další podmínku a samotné **else** uvozuje větev prováděnou při nesplnění žádné z podmínek.

U tohoto způsobu větvení programu se postupně vyhodnocují logické výrazy od prvního k poslednímu. Provádí se ta větev, jejíž podmínka (logický

výraz) je pravdivá. Další logické výrazy se nevyčíslují, ani větve se neprovedou, i když by i jejich logický výraz byl při vyčíslení pravdivý.

Kromě této možnosti větvení programu na více větví, při které se postupně vyhodnocují logické výrazy až do nalezení pravdivého výrazu (není-li nalezen, provádí se poslední větev), existuje i možnost zápisu větvení, u něhož se počítá jediný výraz. Následné větvení se provádí na základě vypočtené hodnoty výrazu, který nemusí být pouze logický. Půjde o větvení na základě výrazu libovolného ordinálního typu (příkaz case, odst. 4.1.3).

4.1.2 Formát zápisu programů

Formát zápisu programu je volný. Nezáleží na tom, kolik mezer, volných řádků nebo komentářů se napíše mezi jednotlivými prvky (identifikátory, čísly, vyhrazenými slovy ap.) programu.

Volného formátu lze s výhodou využít pro zdůraznění vnořených řídicích (a jak později poznáme i datových) struktur. Čitelnost programu však na jeho zápisu závisí a není jedno, zda program zapíšeme ve tvaru

```
if A < B then I:= -1 else if A = B then I:= 0 else I:= 1
```

nebo ve tvaru

```
if A < B then
  I:= -1
else if A = B then
  I:= 0
else
  I:= 1
```

Druhý způsob zápisu je jistě přehlednější a graficky vystihuje strukturu programu.

Každý uživatel jazyka by si měl osvojit metodu grafického zdůrazňování struktury programu a *důsledně* ji dodržovat, neboť chybně vyznačená struktura může při ladění a modifikaci programu vést k tomu, že i zcela zřejmá chyba se stane *neviditelnou*. Například následující zápis ukazuje možnost, jak lze chybně vyznačenou strukturou programu chybu ukrýt:

```
if I < 0 then      { I je menší než 0 }
  if I < -10 then { menší než -10 }
    writeln('Hodnota je < -10')
else                    { větší rovno 0 }
  writeln('Hodnota je >= 0')
```

Jde o část programu, jež bude provádět jinou činnost, než kterou graficky zdůrazňuje zápis programu. Přesto, že varianta **else** je graficky umístěna na úrovni příslušející variantě

```
if I < 0 then
```

platí stále pravidlo přiřazující **else** nejbližšímu předchozímu **then**. Proto se varianta **else** bude provádět v případě, že –10 < = I < 0. Příkaz by proto měl mít strukturu

```
if I < 0 then
  begin { menší než 0 }
    if I < –10 then { menší než –10 }
      writeln('Hodnota je < –10')
  end  { menší než 0 }
else { větší nebo rovno 0 }
  writeln('Hodnota je > = 0')
```

Ponecháme-li jej v původním tvaru, měl by graficky vypadat např. takto:

```
if I < 0 then { menší než 0 }
  if I < –10 then { menší než –10 }
    writeln('Hodnota je < –10')
  else { –10 < = I < 0 }
    writeln('Hodnota je v intervalu <–10,0)')
```

Nyní popíšeme pravidla užívaná zde pro grafickou úpravu programů:

- Každý příkaz začíná na novém řádku.
- Každá řídicí struktura (prozatím známe podmíněný a složený příkaz) způsobí, že vnořené příkazy jsou zapisovány o 3 znaky vpravo.

Podmíněný příkaz bude pak zapisován takto:

```
if logický výraz then
  příkaz1
else
  příkaz2
```

resp.

```
if logický výraz then
  begin
    příkazy1
    :
  end
else
```

begin
  příkazy$_2$
  :
end

Dvojice **else if** je zapisována jako jeden celek (ne však jako jedno slovo; vyhrazená slova musí být oddělena alespoň jednou mezerou). Jednotlivé varianty **else if** zapisujeme pod sebe.

if *logický výraz*$_1$ **then**
  příkaz$_1$
else if *logický výraz*$_2$ **then**
  příkaz$_2$
else if
  :
else
  příkaz$_n$

Obdobná pravidla budeme užívat i u řídicích struktur diskutovaných v dalších kapitolách. Je-li počet zanoření příliš velký a text programu se blíží konci řádku, zmenšíme velikost odsazení na jeden znak.

V předchozí kapitole jsme se zmínili o tom, že před vyhrazeným slovem **end** ve složeném příkazu se může (ale nemusí) psát středník. Je výhodné tento středník před **end** psát *vždy*. Ukazuje se, že častou opravou programu při ladění je vložení příkazu na konec složeného příkazu před **end**. Přitom se často zapomene doplnit středník k původnímu poslednímu (nyní předposlednímu) příkazu. Důsledným psaním středníku před **end** se této chyby vyvarujeme.

Celý proces vytváření programu je vhodné *dokumentovat*. Nejen pro vlastní potřebu, ale i pro předání programu jinému programátorovi (což je při týmové práci běžná situace). Výhodné je, když dokumentace je nedílnou součástí programu ve formě *komentářů*.

Kromě komentářů v těle programu vysvětlujících všechny příkazy, jejichž význam není na první pohled zřejmý, jimi popisujeme zejména význam všech objektů (např. proměnných) při deklaraci. Dále je dobré vytvářet u programů jednotné záhlaví obsahující slovní popis činnosti programu a tvar vstupních i výstupních údajů.

Při zápisu řídicích struktur je důležité, z hlediska čitelnosti programu, označovat vnořené příkazy komentáři popisujícími činnost příkazu. Vnořené příkazy jsou většinou složené a tvoří logický celek. Je výhodné, je-li začátek i konec složeného příkazu označen stejným komentářem. Usnadní se tím v programu identifikace odpovídajících si vyhrazených slov **begin** a **end**.

Například příkaz pro určení ceny vstupenek v kině ze str. 48 by mohl být zapsán takto (tento příkaz oproti předchozí variantě navíc tiskne cenu vstupenek):

```
if (RADA >= 1) and (RADA <= 15) then
  begin { řada se v kině nachází }
    if RADA = 1 then
      CENA := 7  { 1. řada }
    else if (RADA >= 2) and (RADA <= 6) then
      CENA := 8  { 2.–6. řada }
    else
      CENA := 10;   { 7.–15. řada }
    writeln ('Cena vstupenky je ',CENA:1,' Kcs');
  end { řada se v kině nachází }
else { jiná řada }
  writeln ('Tato rada v kine neni');
```

Také je možné označovat koncové **end** poznámkou obsahující vyhrazené slovo **then**, **else** nebo jiné, jímž varianta začínala, např.

```
if logický výraz then
  begin

  end { then }
else
  begin

  end { else }
```

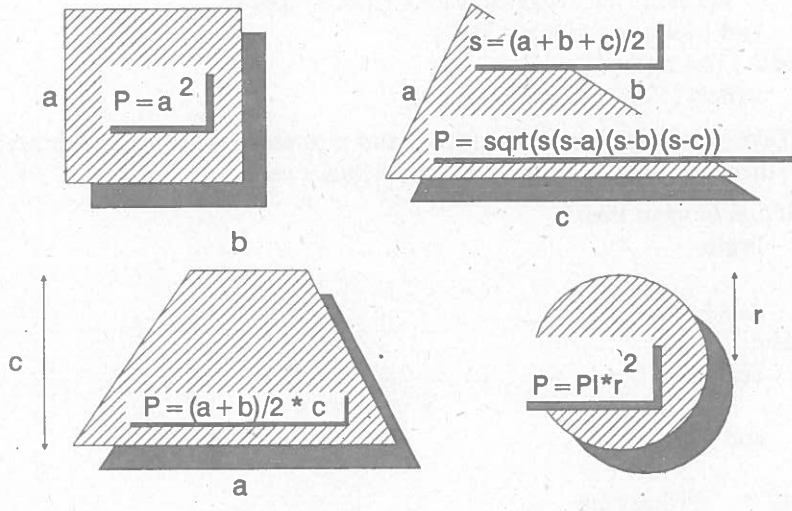
4.1.3 Příkaz case

Již víme, že podmíněný příkaz popisuje větvení programu na základě hodnoty logického výrazu. Logický výraz může nabývat dvou hodnot *false* nebo *true*, proto jedním podmíněným příkazem (s jedním výpočtem logického výrazu) je možné větvit pouze do dvou větví. Větvení do více než dvou variant je nutné zapsat vnořenými podmíněnými výrazy.

Využijeme-li k větvení vypočtené hodnoty ordinálního typu (zatím známe ordinální typy *integer* a *boolean*, další budeme definovat v kap. 5), je možné rozvětvit program na tolik variant, kolik hodnot může daný typ nabýt. Takové větvení je možné zapsat řídicí strukturou nazvanou *příkaz case* nebo *přepínač*. Má tento tvar (in case of = v případě, že):

case *výraz* **of**
  konstanta$_1$: *příkaz*$_1$;
  konstanta$_2$: *příkaz*$_2$;
  :
  konstanta$_n$: *příkaz*$_n$;
end

Výraz i *konstanty* musí být téhož typu. Každá *konstanta* musí vyjadřovat jinou hodnotu.



Obr.4.1 Obsahy plošných obrazců

Užití příkazu case ukážeme na příkladu. Vytvořme program, který bude počítat obsahy různých plošných obrazců (čtverce, trojúhelníka, lichoběžníka a kruhu). Na začátku nabídne program uživateli seznam obrazců, jejichž obsah je schopen vypočítat ve tvaru:

VYPOCET OBSAHU PLOSNEHO OBRAZCE

1. CTVEREC

2. TROJUHELNIK
3. LICHOBEZNIK
4. KRUH

Zadej poradove cislo obrazce:

Uživatel pak zadá jednu z hodnot 1, 2, 3, 4. Výpočet obsahu každého z obrazců se provádí podle jiného vztahu (obr. 4.1). Pro čtverec stačí délka strany a a obsah P je a^2. Pro trojúhelník je možné např. zadat délku všech jeho stran a, b, c a obsah počítat podle Heronova vzorce

$$P = \sqrt{[\,s\,(s-a)\,(s-b)\,(s-c)\,]},$$

kde

$$s = (a + b + c)\,/\,2.$$

Pro lichoběžník je možné zadat délky základen a, b a výšku c. Obsah se vypočte podle vztahu

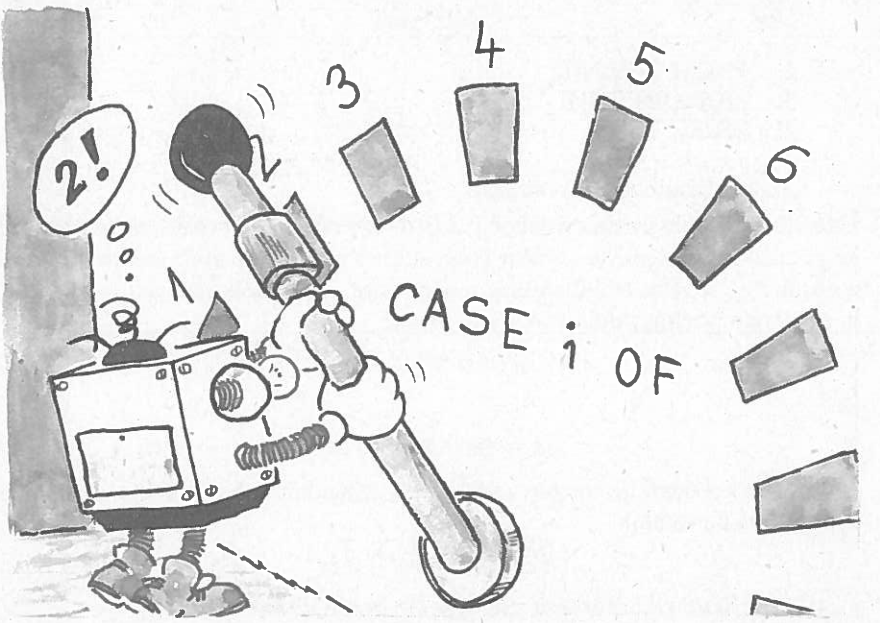
$$P = ((a + b)/2) * c.$$

Obsah kruhu o poloměru r se vypočte podle vztahu

$$P = \pi r^2.$$

V Pascalu bychom zadání vstupních údajů a výpočet obsahů zapsali takto:

```
case OBRAZEC of
1:
  begin { čtverec }
    writeln ('Zadej delku strany ctverce: ');
    readln(A);
    writeln ('Obsah ctverce o strane ',A:6:4,' je ',sqr(A):6:4);
  end; { čtverec }
2:
  begin { trojúhelník }
    writeln ('Zadej delky vsech stran trojuhelnika');
    writeln ('na jeden radek oddelene mezerami:');
    readln(A,B,C);
    S := (A + B + C) / 2;
    writeln ('Obsah trojuhelnika o stranach ',A:6:4,', ',
             B:6:4,', ',C:6:4);
    writeln ('je ', sqrt(S * (S-A) * (S-B) * (S-C)):6:4);
  end; { trojúhelník }
```




```
3:
   begin { lichoběžník }
      writeln ('Zadej delky zakladen a vysku lichobeznika');
      writeln ('na jeden radek oddelene mezerami:');
      readln(A,B,C);
      writeln ('Obsah lichobeznika o zakladnach ',A:6:4,', ',B:6:4);
      writeln ('a vysce ',C:6:4);
      writeln ('je ',(A + B)/2 * C:6:4);
   end; { lichoběžník }
4:
   begin { kruh }
      writeln ('Zadej polomer kruhu:');
      readln(R);
      writeln ('Obsah kruhu o polomeru ',R:6:4,' je ',PI * sqr(R):6:4);
   end; { kruh }
end; { case }
```

PI je zde pojmenovaná konstanta udávající hodnotu Ludolfova čísla, A, B, C, R a S jsou proměnné typu *real*. Jednu z uvedených variant výpočtu chceme vybrat na základě celočíselné hodnoty. Tato hodnota bude předem přečtena do celočíselné proměnné OBRAZEC. Pro rozvětvení užijeme příkazu case.

I při zápisu příkazu case je vhodné dodržovat formát zápisu s odsazením vnořených příkazů vpravo. Koncové **end**, obdobně jako u složeného příkazu, je na stejné úrovni jako **case**. Podobně jako u složeného příkazu, budeme psát středník za všechny varianty přepínače, včetně poslední.

Nabude-li výraz uvedený za vyhrazeným slovem **case** při běhu programu hodnoty, která není totožná s hodnotou žádné z konstant příkazu case, nastane chyba při spuštění programu. Pokud však programátor chce v případě nenalezení odpovídající varianty provést jistý příkaz (a neskončit provádění programu chybou), není to v Pascalu možné.

Proto většina tvůrců překladačů doplnila do příkazu case možnost zápisu větve, která se provádí v případě, že ani jedna za zadaných konstant nemá hodnotu výrazu. Ustálilo se užití vyhrazeného slova **else** (jako u podmíněného příkazu) nebo **otherwise** (otherwise = jinak), kterým se tato varianta uvozuje.

Varianta **else** není povinná. Pokud není uvedena, má přepínač původní vlastnosti.

S variantou **else** má přepínač např. tvar:

case *výraz* **of**
  konstanta$_1$: *příkaz*$_1$;
  konstanta$_2$: *příkaz*$_2$;
  :
  konstanta$_n$: *příkaz*$_n$;
  else
    příkaz;
end

V našem programu můžeme variantu **else** rovněž použít pro odhalení chybně zadaného čísla obrazce:

```
case OBRAZEC of
1:
  begin { čtverec }
    :
  end; { čtverec }
```

```
  2:
    begin { trojúhelník }
      :
    end; { trojúhelník }
  3:
    begin { lichoběžník }
      :
    end; { lichoběžník }
  4:
    begin { kruh }
      :
    end; { kruh }
  else { chybně zadané identifikační číslo obrazce }
    writeln ('Chybne zadane poradove cislo obrazce: ',OBRAZEC:4);
  end; { case }
```

Nabude-li proměnná jiné hodnoty než 1, 2, 3 nebo 4, bude vypsáno varovné hlášení

Chybne zadane poradove cislo obrazce: *xxxx*

kde *xxxx* je chybné číslo.

Pokud se provádí stejná varianta pro několik různých hodnot výrazu, je možné napsat konstanty těchto hodnot společně, oddělené čárkami:

```
case výraz of
  konstanta11, ... konstanta1k : příkaz1;
  konstanta21, ... konstanta2l : příkaz2;
  :
  konstantam1, ... konstantamn : příkazm;
  else
    příkazm+1;
end
```

Poznamenejme, že podmíněný příkaz

```
if logický výraz then
  příkaz1
else
  příkaz2
```

lze příkazem case zapsat takto:

```
case logický výraz of
  false: příkaz2;
  true : příkaz1;
end { case }
```

Na závěr odstavce o příkazu case uveďme program 4.2, který bude počítat obsahy zadaných obrazců.

```
program OBSAHY (input, output);
{ Program provádí výpočet obsahů čtyř plošných útvarů; čtverce,
trojúhelníka, lichoběžníka a kruhu.

  Vstup: Po výzvě
       Zadej poradove cislo obrazce:

       se zadá číslo 1, 2, 3 nebo 4. Číslo určuje útvar, jehož plochu
       budeme počítat. Dále podle vybraného obrazce program vyžaduje
       příslušný počet vstupních údajů.

  Výstup:
       Podle vybraného obrazce:

       Obsah ctverce o strane xxxx je xxxx
       Obsah trojuhelnika o stranach xxxx je xxxx
       Obsah lichobeznika o zakladnach xxxx a vysce xxxx je xxxx
       Obsah kruhu o polomeru xxxx je xxxx

       V případě chybně zadaného čísla obrazce se vypíše

       Chybne zadane poradove cislo obrazce xxxx

}
const
  PI = 3.1415926536;        { Ludolfovo číslo }

var
  OBRAZEC : integer;        { pořadové číslo obrazce }
  A,B,C,R : real;           { vstupní hodnoty }
  S       : real;           { pomocná hodnota }
```

```
begin { OBSAHY }
  writeln ('VYPOCET OBSAHU PLOSNEHO OBRAZCE');
  writeln;
  writeln (' 1.  CTVEREC');
  writeln (' 2.  TROJUHELNIK');
  writeln (' 3.  LICHOBEZNIK');
  writeln (' 4.  KRUH');
  writeln;
  writeln ('Zadej poradove cislo obrazce: ');
  readln (OBRAZEC);        { čtení hodnoty určující obrazec }
  case OBRAZEC of
  1:
    begin { čtverec }
      writeln ('Zadej delku strany ctverce: ');
      readln(A);
      writeln ('Obsah ctverce o strane ',A:6:4,' je ',sqr(A):6:4);
    end; { čtverec }
  2:
    begin { trojúhelník }
      writeln ('Zadej delky vsech stran trojuhelnika');
      writeln ('na jeden radek oddelene mezerami:');
      readln(A,B,C);
      S := (A + B + C) / 2;
      writeln ('Obsah trojuhelnika o stranach ',A:6:4,', ',B:6:4,', ',C:6:4);
      writeln ('je ', sqrt(S * (S-A) * (S-B) * (S-C)):6:4);
    end; { trojúhelník }
  3:
    begin { lichoběžník }
      writeln ('Zadej delky zakladen a vysku lichobeznika');
      writeln ('na jeden radek oddelene mezerami:');
      readln(A,B,C);
      writeln ('Obsah lichobeznika o zakladnach ',A:6:4,', ',B:6:4);
      writeln ('a vysce ',C:6:4);
      writeln ('je ',(A + B)/2 * C:6:4);
    end; { lichoběžník }
  4:
    begin { kruh }
      writeln ('Zadej polomer kruhu:');
      readln(R);
```

```
        writeln ('Obsah kruhu o polomeru ',R:6:4,' je ',PI * sqr(R):6:4);
      end; { kruh }
      else { chybně zadané identifikační číslo obrazce }
        writeln ('Chybne zadane poradove cislo obrazce: ',OBRAZEC:4);
    end; { case }
  end. { OBSAHY }
```

Program 4.2

Všechny reálné hodnoty jsou vypisovány formátem 6:4, tj. v desetinném tvaru, šest znaků celkem, z toho 4 za desetinnou tečkou a jeden znak před tečkou. Pokud má vypisovaná hodnota více platných míst než jednu před desetinnou tečkou, vypíše se na minimální nutný počet znaků. Tak nevzniknou nadbytečné mezery před číslem.

4.2 Cykly

V popisu výrazu je možné si všimnout, že Pascal neobsahuje operátor pro umocňování (pouze standardní funkci *sqr* pro umocňování na druhou). Pokud bychom chtěli tuto operaci do svého programu zapsat, museli bychom ji naprogramovat.

Umocňování

$$X^K,$$

kde X je celé číslo a K celé nezáporné číslo, bychom mohli popsat algoritmem

```
VYSLEDEK := 1;   { nastavení počáteční hodnoty }
opakuj K-krát                                          (4.1)
  VYSLEDEK := VYSLEDEK * X;
```

Zápis *opakuj K-krát* není pascalskou konstrukcí. Zaveďme nyní řídicí struktury umožňující popis opakované činnosti v programu.

Jazykové konstrukci pro zápis opakované činnosti říkáme *cyklus*. Cyklus dovoluje popsat opakované provádění jednoho nebo více příkazů. Jako jediná řídicí struktura umožňuje zvyšovat počet prováděných příkazů v programu v závislosti na vstupních údajích a vypočtených hodnotách.

4.2.1 Cyklus for

Cykly se v programech vyskytují často a je možno je zapsat několika způsoby. V úseku programu (4.1) chceme vyjádřit, že daný příkaz se má opakovat právě K-krát. Pro vyjádření cyklu *s pevným počtem opakování* existuje řídicí struktura nazvaná *cyklus for*. Zapišme pomocí cyklu for úsek (4.1)

```
VYSLEDEK := 1;   { nastavení počáteční hodnoty }
for I := 1 to K do
  VYSLEDEK := VYSLEDEK * X;
```

Cyklus for má tvar

for *proměnná* := *výraz*$_1$ **to** *výraz*$_2$ **do**
  příkaz

V cyklu for musí být vždy použita *proměnná*, která řídí (určuje) počet opakování cyklu (v našem příkladě má jméno I). Proměnná může být libovolného ordinálního datového typu. Této proměnné se říká *řídicí proměnná cyklu*.

Vidíme, že zápis cyklu for má několik částí. V části

for *proměnná* := *výraz*$_1$

se řídicí *proměnné* přiřadí (formou přiřazovacího příkazu) *počáteční hodnota* určená *výrazem*$_1$. Je to hodnota, kterou bude mít řídicí proměnná před započetím cyklu.

Hodnota výrazu

to *výraz*$_2$

slouží k ukončení cyklu. Vypočte se před zahájením cyklu a je vždy *před* provedením *příkazu* zapsaného v třetí části

do
  příkaz

porovnávána s hodnotou řídicí proměnné. Je-li hodnota řídicí proměnné menší nebo rovna hodnotě *výrazu*$_2$, je *příkaz* uvedený za vyhrazeným slovem **do** (nazvaný *tělo cyklu*) proveden. V opačném případě je cyklus ukončen.

Vždy po provedení těla cyklu je řídicí proměnné přiřazena hodnota, která v uspořádané množině hodnot datového typu následuje za současnou hodnotou řídicí proměnné. Pro celá čísla se k řídicí proměnné přičte hodnota 1.

V našem příkladu bude opakovaně prováděn příkaz

```
VYSLEDEK := VYSLEDEK * X;
```

V okamžiku, kdy hodnota I > K, se cyklus ukončí.

Je třeba si uvědomit, že jak počáteční, tak koncová hodnota řídicí proměnné jsou výrazy. Jejich hodnota nemusí být známa již při psaní programu, ale může být vypočtena. Z toho plyne, že i počet opakování cyklu může být určen až v průběhu výpočtu. Nelze proto cyklus obecně nahradit opakovaným zápisem jeho těla.

V jednotlivých opakovaných provedeních těla cyklu roste hodnota řídicí

proměnné vždy o jedničku, dokud není překročena konečná hodnota. Je-li konečná hodnota menší než hodnota počáteční již na počátku provádění cyklu, není tělo cyklu provedeno *ani jednou*.

Variantou cyklu for je zápis (down to = dolů k)

for *proměnná* := *výraz*$_1$ **downto** *výraz*$_2$ **do**
  příkaz

kdy hodnota řídicí proměnné je po každém provedení těla cyklu snížena na hodnotu o jedničku nižší. Cyklus končí, je-li hodnota řídicí proměnné menší než hodnota *výrazu*$_2$.

Vytvořme nyní program, který bude obsahovat cyklus for pro výpočet faktoriálu zadaného čísla. Faktoriál $N!$ přirozeného čísla je definován vztahem

$$N! = 1 * 2 * 3 * \ldots * N.$$

Výpočet faktoriálu je možné realizovat cyklem for např. takto:

```
NFAKTORIAL := 1; { počáteční hodnota pro násobení }
for I:=2 to N do
  NFAKTORIAL := NFAKTORIAL * I;
```

Povšimněme si, že na rozdíl od předchozího příkladu je řídicí proměnná cyklu použita v těle cyklu (ve výrazu). Takové použití je možné, není však dovoleno přiřazovat řídicí proměnné cyklu v těle cyklu hodnotu (např. přiřazením, popř. čtením ze vstupu). Hodnota řídicí proměnné také není po ukončení cyklu definována a nelze ji dále využít.

Při výpočtu faktoriálu je navíc nutno mít na paměti, že hodnota $N!$ roste se zvětšujícím se N velmi rychle, takže již pro relativně malé N (při šestnáctibitovém zobrazení celého čísla již pro $N=8$) může $N!$ přesáhnout maximální hodnotu *maxint* a výpočet je chybný.

```
program FAKTORIAL (input, output);
{ Program vypočte hodnotu faktoriálu přirozeného čísla
        N! = 1 * 2 * 3 * 4 * ... * N
pro zadané N
Vstup: Po výzvě
  Zadej prirozene cislo ...

  se zadává přirozené číslo N ( pro maxint = 32767 je maximální N = 7 )
Výstup:
  N! = xxxxx
}
```

```
var
  NFAKTORIAL,              { hodnota faktoriálu }
  N,
  I: integer;              { řídicí proměnná cyklu }
begin { FAKTORIAL }
  writeln('VYPOCET FAKTORIALU PRIROZENEHO CISLA N');
  writeln;
  writeln('Zadej prirozene cislo mensi nez 8:');
  readln (N);              { vstup hodnoty N }
  NFAKTORIAL := 1;         { počáteční hodnota pro násobení }
  for I:=2 to N do
    NFAKTORIAL := NFAKTORIAL * I;
  writeln('N! = ',NFAKTORIAL:5);
end. { FAKTORIAL }
```

Program 4.3

4.2.2 Cyklus while

Druhou variantou cyklu je cyklus s obecným logickým výrazem. Na základě pravdivosti *logického výrazu* (podobně jako u podmíněného příkazu) se tělo cyklu provede, nebo se cyklus ukončí. Tvar zápisu je:

while *logický výraz* **do**
  příkaz

Příkaz v těle cyklu se provádí tak dlouho, dokud je *logický výraz* v záhlaví *pravdivý* (má hodnotu *true*). Vyhodnocení *logického výrazu* se provádí vždy *před* provedením těla cyklu. Je-li *logický výraz* nepravdivý již na počátku, neprovede se *příkaz* v těle cyklu *ani jednou*. Tuto variantu cyklu nazýváme *cyklus while* (while = zatímco).

Chceme např. vytvořit program, který by počítal aritmetický průměr z předem neznámého počtu reálných hodnot. Zadáme-li nulovou hodnotu, bude to znamenat, že vstup hodnot je ukončen. Byla-li zadána alespoň jedna nenulová hodnota, požadujeme výstup ve tvaru

Aritmeticky prumer hodnot je *xxxx*

Předpokládáme však také, že nemusí být zadána ani jedna nenulová hodnota (již první zadaná hodnota bude nulová), např. omylem. V takovém případě požadujeme výpis

Nebyla zadana ani jedna platna hodnota

Aritmetický průměr určíme jako součet hodnot dělený jejich počtem.

V našem programu bude součet hodnot reprezentovat reálná proměnná SOUCET a počet hodnot bude uložen v celočíselné proměnné POCET. Poslední přečtená hodnota bude uložena v reálné proměnné HODNOTA.

Čtení a součet hodnot vyjádříme pak cyklem while

```
readln (HODNOTA);
while HODNOTA < > 0.0 do { cyklus pro vstup a součet hodnot }
  begin
    SOUCET : = SOUCET + HODNOTA;
    POCET : = POCET + 1;
    readln (HODNOTA); { čtení další hodnoty }
  end;
```

Každý cyklus v programu vyžaduje provedení tzv. *inicializace cyklu*; jsou to příkazy, které připravují počáteční podmínky provádění cyklu. Inicializace může být částečně obsažena v hlavičce cyklu (např. u cyklu for) a obsahuje přiřazení počátečních hodnot proměnným, které budou v cyklu měněny nebo jsou předmětem vyhodnocení logického výrazu v záhlaví cyklu. Inicializace se provádí pro každé provedení cyklu pouze jednou.

V našem příkladu budou proměnné SOUCET i POCET v inicializaci cyklu vynulovány a do proměnné HODNOTA se přečte první hodnota zadaná uživatelem.

Po ukončení cyklu se testuje, zda počet nenulových vstupních hodnot je větší než nula, vypočte se a vypíše průměr (v desetinném tvaru na 5 desetinných míst). Je-li počet vstupních hodnot nulový, vypíše se varovné hlášení.

```
program PRUMER (input, output);
{ Program provádí výpočet aritmetického průměru zadaných reálných
hodnot.

  Vstup: Po výzvě
    Zadavej hodnoty ...

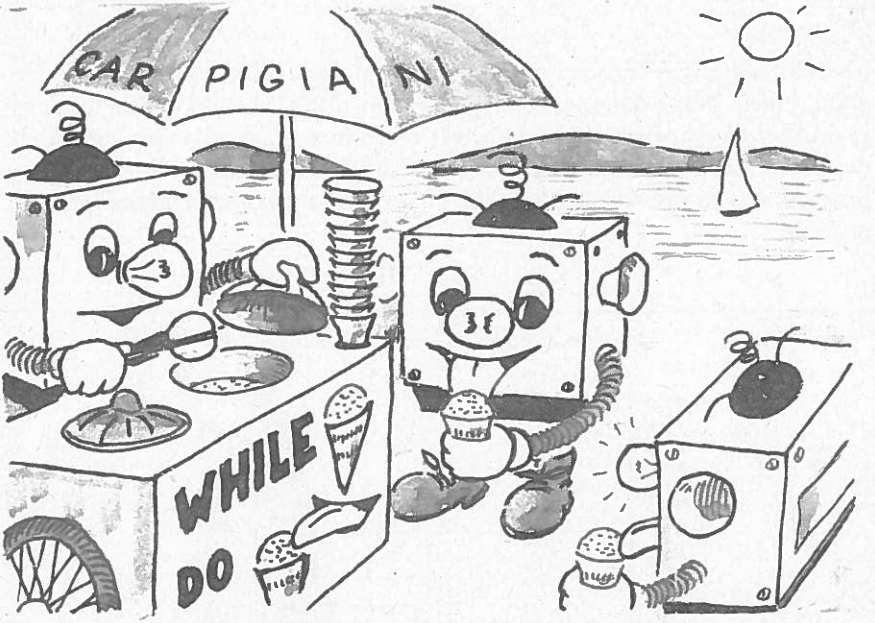
      se zadávají reálná čísla, každé na zvláštní řádek. Zadávání se
      ukončí nulovou hodnotou.

  Výstup:
    Aritmeticky prumer hodnot je xxxxxx
```

```
    Pokud nebyla zadána žádná nenulová hodnota, vypíše se
      Nebyla zadana zadna platna hodnota
}
var
  HODNOTA,                          { právě přečtená hodnota }
  SOUCET      : real;               { součet hodnot }
  POCET       : integer;            { počet hodnot }
begin { PRUMER }
  writeln ('VYPOCET ARITMETICKEHO PRUMERU');
  writeln;
  writeln ('Zadavej hodnoty, kazdou na zvlastni radek:');
  writeln ('(vstup ukonci zadanim nulove hodnoty )');
  POCET   := 0;                     { počet zadaných hodnot }
  SOUCET := 0;                      { součet zadaných hodnot }
  readln (HODNOTA);                 { čtení první zadané hodnoty }
  while HODNOTA <> 0.0 do           { cyklus pro vstup a součet hodnot }
    begin
      SOUCET := SOUCET + HODNOTA;
      POCET  := POCET + 1;
      readln (HODNOTA);             { čtení další hodnoty }
    end;
  if POCET > 0 then { byla zadána alespoň jedna platná hodnota }
    writeln ('Aritmeticky prumer hodnot je ',SOUCET/POCET:7:5)
  else  { není zadána ani jedna platná hodnota }
    writeln ('Nebyla zadana zadna platna hodnota');
end. { PRUMER }
```

Program 4.4

Tvar logického výrazu v záhlaví cyklu while není omezen. Může proto obsahovat i proměnné, jejichž hodnoty jsou v těle cyklu libovolně měněny. Tvůrce programu musí zajistit, aby *logický výraz* v průběhu provádění programu nabyl alespoň jednou hodnoty *false*. Znamená to, že *logický výraz* v záhlaví cyklu musí obsahovat alespoň jednu proměnnou a hodnota alespoň jedné z proměnných uvedených v *logickém výrazu* v záhlaví cyklu musí být při provádění těla cyklu měněna. V opačném případě dojde k tomu, že výpočet programu neskončí. Toto tzv. *zacyklení* programu je častou chybou zejména u začátečníků. Proto pokud se výpočet neukončí za *rozumnou* dobu, je obvyklé hledat chybu u cyklu while (popř. u jeho zrcadlového obrazu, cyklu repeat).



4.2.3 Cyklus repeat

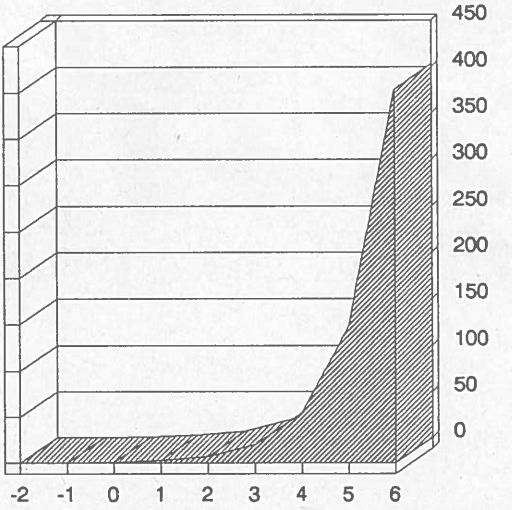
Třetí a poslední typ cyklu je stejně jako cyklus while řízen libovolným logickým výrazem. Nazývá se *cyklus repeat* (repeat = opakuj, until = dokud ne) a má tvar:

repeat
  příkaz$_1$;
  příkaz$_2$;
  :
  příkaz$_n$
until *logický výraz*

Příkazy uzavřené v příkazových závorkách

repeat
until *logický výraz*

tvoří složený příkaz se všemi dosud popsanými vlastnostmi a reprezentují tělo cyklu repeat. Tělo cyklu se provádí tak dlouho, dokud je *logický výraz* uvedený za vyhrazeným slovem **until** nepravdivý (má hodnotu *false*). Jakmile získá hodnotu *true*, provádění cyklu se ukončí. Vyhodnocování *logického výrazu* se provádí vždy *po* provedení těla cyklu. Proto se tělo cyklu vždy *alespoň jednou* provede.

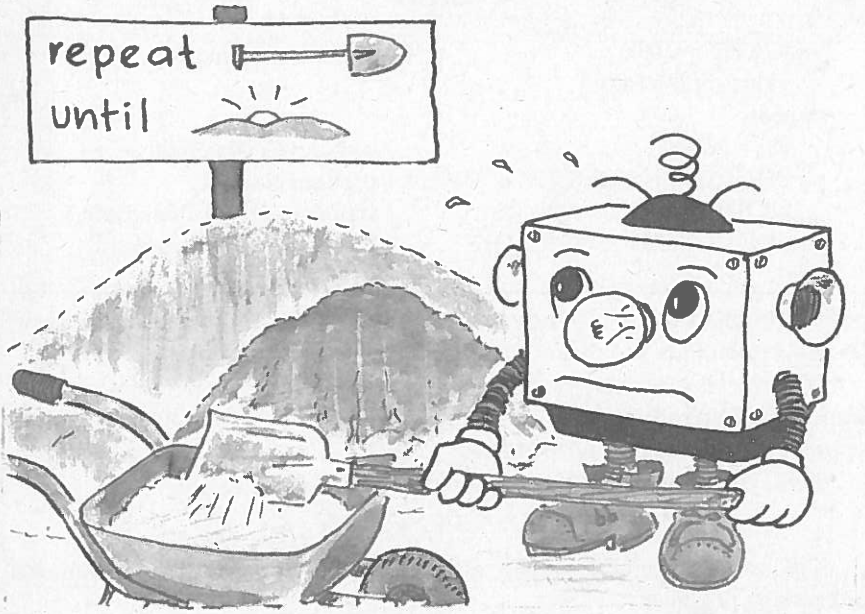


Obr.4.2 Funkce exp X

Uveďme si příklad užití cyklu repeat. Víme, že Pascal poskytuje uživateli některé standardní funkce. Počítač neuchovává hodnoty funkce ve formě tabulky, protože by to bylo pamětově náročné a s omezenou přesností, ale k výpočtu funkcí používá např. součtu mocninných řad. Ukažme si, jak se na počítači realizuje např. výpočet funkce exp X (obr 4.2); v Pascalu je označena jako *exp*(X).

Pro výpočet exp X je odvozen součet mocninné řady (např. v [17]) :

$$\exp X = 1 + \frac{X}{1!} + \frac{X^2}{2!} + \frac{X^3}{3!} + \ldots .$$



Vzhledem k tomu, že absolutní hodnoty jednotlivých členů řady jsou od jistého členu řady se stoupající mocninou stále menší, blíží se výsledná hodnota součtu řady hodnotě exp X po stále se zmenšujících úsecích. Chceme-li vypočítat hodnotu funkce se zadanou přesností ε, je třeba porovnat vždy absolutní hodnotu vypočteného členu řady s hodnotou ε. Je-li hodnota členu řady menší než zadaná přesnost ε, je možné cyklus ukončit.

Zápis cyklu pro výpočet exp X bude mít tuto strukturu. X je hodnota argumentu, EPSILON přesnost výpočtu a CLEN obsahuje hodnotu vypočteného členu řady. SOUCET je celková hodnota, v níž se ukládá součet členů řady. Uvedené proměnné jsou typu *real*. K je pořadové číslo členu, počítáno od nuly (K je typu *integer*).

```
{ inicializace cyklu }
CLEN := 1.0;                    { první člen řady }
K := 0;                         { pořadové číslo členu }
```

```
SOUCET := 0.0;                          { inicializace součtu }
{ cyklus součtu řady }
repeat
  K := K + 1;                           { pořadové číslo členu + 1 }
  SOUCET := SOUCET + CLEN;              { přičtení členu }
  CLEN := CLEN * X / K;                 { výpočet dalšího členu řady }
until abs(CLEN) < EPSILON;              { dokud člen >= přesnost }
```

Před započetím cyklu se hodnota nultého členu řady (tj. hodnota 1.0) přiřadí proměnné CLEN. Hodnota součtu řady SOUCET je na začátku vynulována. Hodnota K pro uchování pořadového čísla členu se nastaví na nulu.

V těle cyklu se vždy nejprve upraví počitadlo členů K (zvýší se o 1). Vypočtený člen řady se pak přičte k celkovému součtu řady SOUCET. Potom se provede výpočet nového členu řady CLEN.

Další člen řady se počítá z předchozího členu podle vztahu

$$CLEN_K = CLEN_{K-1} * X / K.$$

Cyklus se provádí tak dlouho, dokud absolutní hodnota členu není menší než zadaná přesnost.

Nyní již můžeme vytvořit celý program pro výpočet funkce exp *X*. Pro kontrolu budeme vypisovat rovněž hodnotu standardní pascalské funkce *exp*(X) pro zadaný argument.

```
program ENAXTOU (input,output);
{ Výpočet funkce exp X součtem mocninné řady
Vstup: Po výzvě
     Zadej argument X a presnost ...
   obě hodnoty na jeden řádek oddělené mezerami. Přesnost se udává jako
   reálné číslo, určující maximální hodnotu posledního členu řady.
Výstup:
   exp X = y1
   kontrolni hodnota = y2
}
var
   X,                                   { hodnota argumentu }
   EPSILON,                             { přesnost výpočtu }
   SOUCET,                              { součet mocninné řady }
   CLEN    : real;                      { člen řady }
   K       : integer;                   { pořadové číslo členu řady }
```

```
begin { ENAXTOU }
  writeln ('VYPOCET FUNKCE exp X');
  writeln;
  { čtení vstupních hodnot }
  writeln ('Zadej argument X a presnost');
  writeln ('na jeden radek oddelene mezerou.');
  readln (X, EPSILON);
  { inicializace cyklu }
  CLEN := 1.0;                          { první člen řady }
  K := 0;                               { pořadové číslo členu }
  SOUCET := 0.0;                        { inicializace součtu }
  { cyklus součtu řady }
  repeat
    K := K + 1;                         { pořadové číslo členu + 1 }
    SOUCET := SOUCET + CLEN; { přičtení členu }
    CLEN := CLEN * X / K;               { výpočet dalšího členu řady }
  until abs(CLEN) < EPSILON;            { dokud člen > = přesnost }
  { výstup výsledných hodnot }
  writeln ('exp ',X:9:7,' = ', SOUCET:9:7);
  writeln ('kontrolni hodnota = ', exp(X):9:7);
end. { ENAXTOU }
```

Program 4.5

4.3 Cvičení

1. Vytvořte program pro řešení obecné kvadratické rovnice

$$ax^2 + bx + c = 0 \qquad (a <> 0)$$

pro libovolné hodnoty b, c. Program by měl provádět výpočet reálných i komplexních kořenů $x_{1,2}$.

2. Newtonovou iterační metodou vytvořte program pro výpočet druhé odmocniny z nezáporného čísla A s přesností ε.
Návod: Použijte vztahu

$$x_{i+1} = 0{,}5\,(x_i + A / x_i),$$
$$\varepsilon = |\, x_{i+1} - x_i \,|.$$

Na začátku výpočtu zvolte počáteční hodnotu $x_0 = 1$. Liší-li se dvě následující vypočtená x (v absolutní hodnotě) o méně než ε, výpočet ukončete.

3. Čísla na vstupu jsou složky vektorů A a B v pořadí

$$n\ a_1\ b_1\ a_2\ b_2\ a_3\ b_3\ \ldots\ a_n\ b_n.$$

Vytvořte program, který vypočte skalární součin vektorů

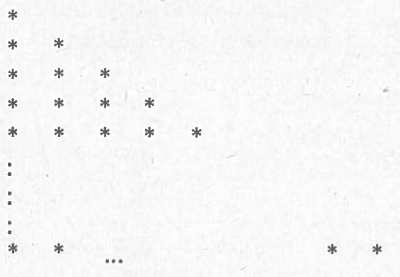
$$A\ .\ B\ =\ \text{součet}\ a_i\ .\ b_i\ \text{pro}\ i = 1, 2, \ldots, n.$$

4. Vytvořte program, který zjistí, zda zadané celé číslo n je prvočíslo. Výsledkem programu bude hlášení

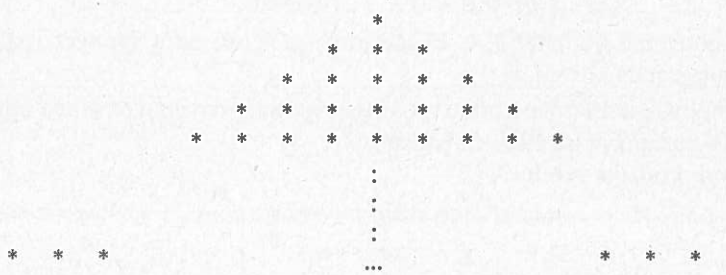
Cislo n je (resp. neni) prvocislo

Návod: Zjistěte postupně zbytek po dělení čísla n čísly 2,3, ..., n–1. Není-li zbytek ani jednou nulový, jde o prvočíslo. Je-li zbytek nulový, nejde o prvočíslo a výpočet je možné ukončit.

5. Vytvořte program, který přijme na vstupu celé číslo VYSKA a nakreslí obrazec. Počet řádků je dán hodnotou VYSKA.



6. Modifikujte příkl. 5 tak, aby výsledný obrazec měl tvar



5 ZNAKY, VÝČTOVÉ TYPY, INTERVALY

5.1 Znaky a kódy

V předchozích kapitolách jsme poznali řetězce znaků. Jsou to posloupnosti znaků jisté délky. Prozatím jsme je využívali pouze v příkazech *write*, resp. *writeln*. Znaky jsou prvky každého textu a zpracování textů je kromě číselných výpočtů další charakteristickou oblastí práce počítačů. K tomu, aby počítač mohl se znaky pracovat, musí je umět převést do svého vnitřního tvaru a v tomto tvaru s nimi provádět operace.

V kap. 1 byl vysvětlen způsob kódování znaků. Každému znaku odpovídá jiná dvojková kombinace a té zase celočíselná hodnota. Říkáme, že znak je zakódován jistým *ordinálním číslem* (pro každý znak jiným).

V kódu jsou znaky uspořádány podle svých ordinálních čísel. Všechny běžně používané kódy zaručují, že *písmena* jsou uspořádána *vzestupně*, tj.

ordinální číslo *A* < ordinální číslo *B* < ... < ordinální číslo *Z*,

ordinální číslo *a* < ordinální číslo *b* < ... < ordinální číslo *z*.

Není však zaručeno, že mezi sousedními písmeny v abecedě není v kódu další znak (tj. posloupnost ordinálních čísel písmen *nemusí být souvislá*). U *číslic* je zaručeno, že jsou uspořádány *vzestupně* a posloupnost ordinálních čísel pro číslice je *souvislá*.

ordinální číslo *1* < ordinální číslo *2* < ... < ordinální číslo *9*.

Pro vyjádření znakových hodnot existuje v Pascalu datový typ *char* (**char**acter = znak). Hodnotami typu *char* jsou všechny znaky, které lze na počítači zpracovávat. Některé ze znaků se na displeji nebo na tiskárně nezobrazují.

Například v kódu ASCII jsou znaky s ordinálními čísly 0 až 31 nazývány *řídicími znaky* a jsou obvykle nezobrazitelné. Řídicí znaky označujeme anglickými zkratkami, např. BS, LF a CR. BS (back space = posun zpět) má v kódu ASCII ordinální číslo 8 a vyjadřuje posun v textu o jeden znak zpět. Podobně LF (line feed = posun o řádek) má ordinální číslo 10 a představuje posun na další řádek. CR (carriage return = návrat vozíku) má ordinální číslo 13 a označuje návrat na začátek řádku. Úplný popis všech řídicích znaků kódu

ASCII lze nalézt v příloze B. Řídicí znaky se na klávesnici obvykle zapisují stisknutím speciální klávesy (např. klávesy označené BS). Druhou možností je stisknout klávesu *Ctrl* současně s dalším znakem, podobně jako vstup velkých písmen lze dosáhnout stisknutím klávesy *Shift* (přepínače velkých písmen) společně s klávesou určující písmeno.

Tak např. znak BS můžeme zapsat stisknutím klávesy BS nebo také *Ctrl* H. Stisknutím klávesy RETURN, resp. ENTER dosáhneme obvykle vstupu znaků CR a LF. Znak LF samotný můžeme zapsat jako *Ctrl* J, znak CR jako *Ctrl* M. Také všechny kombinace s *Ctrl* lze nalézt v příloze B. Přesto, že řídicí znaky jsou nezobrazitelné, jsou platnými hodnotami datového typu *char*.

Podobně jako u datových typů *integer* a *real* je možno deklarovat proměnné typu *char*. Například zápis

```
var
  ZNAK, NOVYZNAK: char;
```

deklaruje dvě proměnné ZNAK a NOVYZNAK typu *char*.

Pro přímý zápis znakové hodnoty do programu slouží *znakové literály*. Mají stejný tvar jako řetězce, tj. začínají a končí apostrofem. Na rozdíl od řetězců mohou obsahovat pouze *jediný* znak. Je-li znakový literál ohraničen apostrofy, vzniká, podobně jako u řetězců, problém s vyjádřením hodnoty samotného apostrofu. Proto, podobně jako u řetězců, musíme tento znak zdvojit. Znakový literál s hodnotou apostrof obsahuje tedy čtyři za sebou napsané apostrofy ''''. Dva krajní ohraničují literál a dva prostřední vyjadřují jediný apostrof.

Tedy

'A' 'b' '0' '#' '''' '%'

jsou znakové literály vyjadřující znaky A b 0 # *znak apostrof* a %. Nezobrazitelné znaky nelze ve tvaru literálu zapsat. Existují-li znakové literály, mohou existovat i pojmenované znakové konstanty, např.

```
const
  VELKEA     = 'A';     { velké písmeno A }
  KONECVETY  = '.';     { tečka na konci věty }
```

Bylo řečeno, že znaky musí být v paměti počítače reprezentovány posloupností bitů (obvykle osmi), které říkáme *slabika*. Je výhodné, můžeme-li k danému znaku získat číselnou hodnotu jeho kódové kombinace (ordinální číslo znaku) a naopak k danému ordinálnímu číslu jeho znak. Existují dvě funkce pro převod znaku na jeho ordinální číslo a naopak. Funkce

ord(X),

kde X je znaková hodnota, vyjadřuje celočíselnou hodnotu ordinálního čísla znaku X. Funkce

chr(Y),

kde Y je celočíselná hodnota, vyjadřuje znakovou hodnotu zakódovanou ordinálním číslem Y. Platí tedy (pro znak X)

X = *chr* (*ord* (X)).

V kódu ASCII platí např.

ord('!')	= 33,	popř.	*chr*(33)	= '!'
ord('0')	= 48		*chr*(48)	= '0'
ord('A')	= 65		*chr*(65)	= 'A'
ord('a')	= 97		*chr*(97)	= 'a'

Podobně jako u číselných datových typů, je možné znakové hodnoty číst ze vstupu příkazem *read*, resp. *readln* a zapisovat je na výstup příkazy *write*, resp. *writeln*. Při čtení znakové hodnoty se ze vstupu do každé proměnné přečte právě jeden znak.

Například

```
readln (ZNAK, NOVYZNAK);
write ('+', ZNAK, NOVYZNAK :5);
writeln ('–':10, ZNAK:SIRKA);
```

Je možné užívat formátovaný výstup znakových hodnot. Pokud není formát výstupu :*celočíselný výraz* zadán, zapisuje se znak na jedinou pozici. Je-li zadána šířka hodnotou *celočíselného výrazu*, předcházejí na výstupním zařízení vystupujícímu znaku mezery tak, aby byla vyplněna zadaná šířka výstupu. Zápis '–':10 způsobí tedy výstup 9 mezer následovaných znakem –.

Velmi často prováděnou činností na počítači je převod celočíselné hodnoty vyjádřené posloupností znaků představujících číslice na celočíselnou hodnotu typu *integer* a naopak. Připomeňme, že hodnota čísla složeného z číslic a_1, a_2, ... , a_n je pro daný základ soustavy z

$$a_1 z^{n-1} + a_2 z^{n-2} + \ldots + a_{n-1} z + a_n.$$

Jakákoliv číselná hodnota vstupuje (vystupuje) do (z) počítače jako posloupnost číslic. Číslice mají podle daného kódu přiřazena jistá ordinální čísla (např. v kódu ASCII postupně od '0' jsou to čísla 48, 49, ... , 57). Číslo 534 je např. vyjádřeno posloupností tří znaků '5', '3' a '4' s ordinálními čísly v ASCII 53, 51 a 52. Čteme-li takové číslo ze vstupního zařízení, získáme postupně 3 znaky s uvedenými ordinálními čísly.

Chceme-li s číslem 534 pracovat jako s celočíselnou hodnotou, musíme je nejprve z posloupnosti znaků převést na jedinou celočíselnou hodnotu. Říkáme, že je nutné provést *konverzi* ze znakového tvaru na číselný. Nutnost provádění konverze si často ani neuvědomíme, neboť ji za nás automaticky provádějí příkazy *read*, *readln*, resp. *write*, *writeln* při čtení, resp. zápisu číselné hodnoty (čteme do proměnných typu *integer*, *real* a zapisujeme hodnoty výrazů těchto typů). Pokud bychom tyto příkazy neměli, museli bychom při programování vstupů a výstupů konverzi provádět.

Vytvořme nyní program, který bude číst čísla ze vstupu a bude je převádět na celočíselnou hodnotu při zadaném základu soustavy od 2 do 10. Pro čtení tedy nebude používat konverze poskytované standardně (Pascalem pouze pro desítkovou soustavu). Program přečte ze vstupu posloupnost znaků ukončenou mezerou (předpokládáme, že to budou číslice). Posloupnost znaků převede na celé číslo a číselnou hodnotu vypíše (už s využitím konverze příkazu *writeln* pro celá čísla) v desítkové soustavě.

Algoritmus konverze využívá skutečnosti, že posloupnost ordinálních čísel pro číslice je souvislá a uspořádaná vzestupně od nuly. Číselnou hodnotu číslice získáme tak, že od ordinálního čísla dané číslice odečteme ordinální číslo znaku '0'. Například pro číslici '5' je ordinální číslo 53. Ordinální číslo znaku '0' je 48 a 53 – 48 = 5, což je právě celočíselná hodnota dané číslice. Platí tedy

$$\text{číselná hodnota číslice X} = ord\,(\text{X}) - ord\,(\text{'0'}).$$

Číslice čteme postupně odleva doprava. Uvažujeme-li polynom

$$H = a_1 z^{n-1} + a_2 z^{n-2} + \ldots + a_{n-1} z + a_n,$$

pak nejdříve čteme a_1, pak a_2 atd. Nakonec přečteme a_n. Vztah pro výpočet hodnoty při zadaném základu jsme již uvedli (předchozí polynom). Můžeme jej však také rozepsat do tvaru tzv. *Hornerova schématu*:

$$\begin{aligned} H_0 &= 0, \\ H_1 &= H_0 z + a_1, \\ H_2 &= H_1 z + a_2, \\ &\vdots \\ H_n &= H_{n-1} z + a_n = H. \end{aligned}$$

Čteme-li např. číslo 534, pak pro základ 10 jsou vypočtené hodnoty

$$\begin{aligned} H_0 &= 0, \\ H_1 &= 0 \,.\, 10 + 5 = 5, \\ H_2 &= 5 \,.\, 10 + 3 = 53, \\ H_3 &= 53 \,.10 + 4 = 534 = H. \end{aligned}$$

Jednotlivé řádky Hornerova schématu můžeme snadno zapsat jako tělo pascalského cyklu. Při výpočtu se vždy nejdříve vynásobí předchozí hodnota

proměnné H základem a přičte se k ní hodnota právě přečtené číslice. Po ukončení cyklu je v proměnné H celočíselná hodnota přečteného čísla s daným základem soustavy.

Cyklus ukončíme, když přečtený znak nebude číslicí. Předpokládáme také, že již první znak nemusí být číslicí. V tom případě bude výsledná hodnota 0.

```
H := 0;                  { počáteční hodnota čísla }
read(CISLICE);           { první znak čísla }
while (CISLICE >= '0') and (CISLICE <= '9') do
  begin { čtení jednotlivých číslic }
    H := H * ZAKLAD + ord (CISLICE) - ord ('0');
    read(CISLICE);
  end; { čtení jednotlivých číslic }
```

Právě čtená číslice je ve znakové proměnné CISLICE, základ soustavy v celočíselné proměnné ZAKLAD.

V celém programu 5.1 pro konverzi se před provedením cyklu navíc kontroluje, zda daný základ soustavy je v intervalu <2,10>. Pokud není, je vypsáno varovné hlášení

Zaklad *xx* neni pripustny

a proveden výpočet pro základ 10.

```
program KONVERZE (input, output);
{ Program provádí konverzi znakového tvaru celého kladného čísla
  s daným základem na celočíselnou hodnotu.

  Vstup: Po výzvě

       Zadej zaklad a cislo:

  se zadá na jednom řádku základ (číslo 2–10) a na dalším řádku číslo
  v dané soustavě ukončené znakem, který není číslicí (např. mezerou)
  Výstup:
      Hodnota cisla: xxx
}
var
  ZAKLAD,                        { zadaný základ }
  H          : integer;          { celočíselná hodnota }
  CISLICE    : char;             { právě přečtený znak }
```

```
begin { KONVERZE }
  writeln('KONVERZE CELEHO KLADNEHO CISLA');
  writeln;
  writeln ('Zadej zaklad a cislo:');
  readln (ZAKLAD);  { čtení základu }
  if (ZAKLAD < 2) or (ZAKLAD > 10) then
    begin { nedovolený základ }
      writeln ('Zaklad ',ZAKLAD:1,' neni pripustny');
      writeln ('Byl dosazen zaklad 10');
      ZAKLAD := 10;
    end; { nedovolený základ }
  H := 0;              { počáteční hodnota čísla }
  read(CISLICE);  { první znak čísla }
  while (CISLICE >= '0') and (CISLICE <= '9') do
    begin { čtení jednotlivých číslic }
      H := H * ZAKLAD + ord (CISLICE) - ord ('0');
      read(CISLICE);
    end;  { čtení jednotlivých číslic }
  writeln('Hodnota cisla: ',H:1);
end. { KONVERZE }
```

Program 5.1

Je vidět, že se znakovými hodnotami není možné provádět aritmetické operace (sčítání, násobení atd.). To je možné pouze s jejich ordinálními čísly:

$$H := H * ZAKLAD + ord\,(CISLICE) - ord\,('0')$$

Kromě operací *ord* a *chr* je možné se znaky provádět ještě operace získání předchůdce (**pred**ecessor = předchůdce) a následníka (**succ**essor = následník). Vzhledem k tomu, že existuje uspořádání znaků, má každý znak (mimo prvního) svého předchůdce. Jeho hodnotu je možné získat funkcí

$$pred(X),$$

kde X je znaková hodnota. Hodnotou funkce je znak s ordinálním číslem $ord(X) - 1$. Obdobnou funkcí je

$$succ(X),$$

která určuje následníka libovolného znaku X (mimo posledního). Je to znak s ordinálním číslem $ord(X) + 1$. Platí

$$pred(X) = chr\,(ord\,(X) - 1),$$

$$succ(X) = chr\,(ord\,(X) + 1).$$

Vzhledem k tomu, že množina ordinálních čísel pro písmena je uspořádaná, avšak není nutně souvislá (souvislost je vyžadována pouze u číslic), nemusí vždy platit (např. v kódu ASCII následující rovnosti platí, avšak v kódu EBCDIC nikoliv)

$$succ('I') = 'J' \qquad \text{nebo} \qquad pred('S') = 'R',$$

ale musí vždy platit

$$succ('0') = '1' \qquad \text{nebo} \qquad pred('8') = '7'.$$

5.2 Jiné nenumerické hodnoty

Všechny až dosud uvažované datové typy *integer*, *char*, *boolean* a *real* nám poskytuje přímo jazyk Pascal (jsou standardní). Znamená to, že kdykoliv začneme psát libovolný program, máme tyto datové typy i s jejich operacemi k dispozici.

Hodnoty uvedených typů jsou z hlediska operací nad těmito typy jednoduché a dále nedělitelné (čísla, znaky ap.), proto těmto typům říkáme *jednoduché datové typy*.

Jednoduché datové typy jsou rozděleny na dvě skupiny. První skupinu tvoří datový typ *integer* a spolu s ním všechny jednoduché typy, jejichž hodnoty lze převést na celá (ordinální) čísla funkcí *ord*. Všem takovým datovým typům říkáme *ordinální*.

Druhou skupinu tvoří samostatně datový typ *real*, u něhož zobrazení hodnot na celá čísla funkcí *ord* neexistuje.

Ordinální datové typy mají některé společné vlastnosti dané zobrazením jejich hodnot na hodnoty typu *integer*, které je třeba zdůraznit.

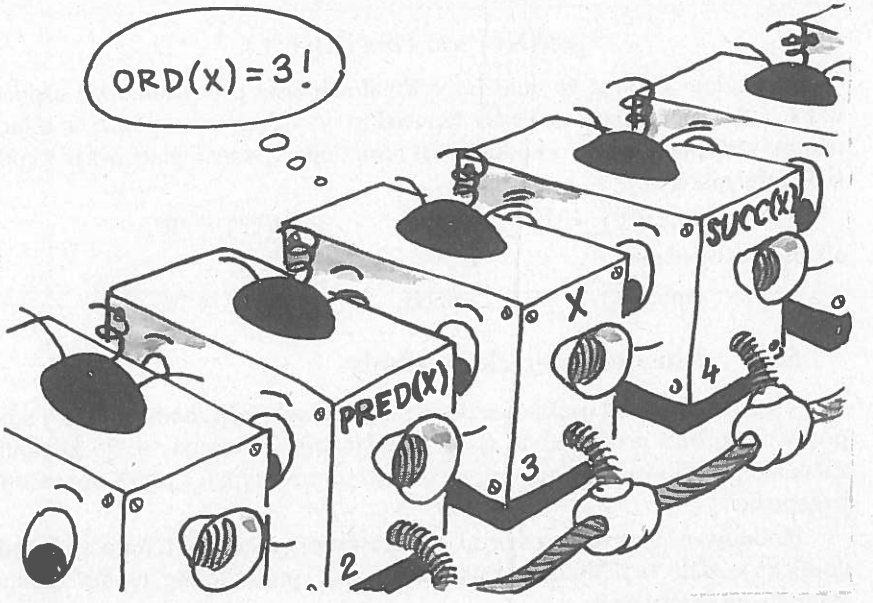
Hodnoty ordinálního typu jsou *uspořádané*. Proto nad všemi ordinálními typy jsou definovány relační operace

$$=,\ <>,\ <,\ >,\ <=,\ >=$$

s výsledkem typu *boolean*. Každý ordinální typ má *konečný počet hodnot*. Pro celá čísla je definována standardní pojmenovaná celočíselná konstanta

```
const
  maxint = ... ;   { hranice celočíselných hodnot }
```

která je dána zobrazením celého čísla v operační paměti počítače. Typ *integer* má pak hodnoty v intervalu $< -maxint, maxint >$.
Při zobrazení celého čísla se znaménkem na 16 bitů má *maxint* hodnotu 32767.



Pro všechny hodnoty X ordinálního typu (mimo nejmenší hodnotu) je definována funkce *předchůdce pred*(X). Pro všechny hodnoty X (mimo největší hodnotu) je definována funkce *následníka succ*(X).

Ke každé hodnotě X ordinálního typu je přiřazena právě jedna celočíselná hodnota *ord*(X), tzv. *ordinální číslo*. Pro celá čísla platí

$$ord(X) = X.$$

Pro ostatní ordinální typy jsou ordinální čísla přiřazena od 0 ve směru uspořádání (nejmenší hodnota má ordinální číslo 0, další 1 atd.).

Následující logické výrazy jsou pravdivé (mají hodnotu *true*):

succ(–23) = –22,	*succ*('0') = '1',	*pred*(*true*) = *false*,
pred(0) = –1,	*pred*('9') = '8',	*succ*(*false*) = *true*,
ord(19) = 19,	'A' < 'Z',	*false* < *true*,
25 < = 103,	'a' < = 'b',	*true* > = *false*,
ord('a') = 97 {pro ASCII},		*ord*(*true*) = 1.

Hodnoty datového typu *real* jsou pouze uspořádané a jsou nad nimi definovány operace porovnání =, < >, <, < =, >, > = s výsledkem *boolean*.

5.2.1 Výčtové typy

Mimo standardní ordinální datové typy *integer*, *char* a *boolean* je možné popsat a užít v programu další uživatelem deklarované datové typy vesměs určené pro nenumerické výpočty.

Představme si např., že náš program má zpracovávat údaje o barvách. Obvyklý postup ve starších programovacích jazycích spočíval ve vytvoření pomocné převodní tabulky mezi barvami a celými čísly:

bílá	0
žlutá	1
modrá	2
zelená	3
červená	4
hnědá	5
šedá	6
černá	7

Tuto tabulku měl programátor při vytváření programu před sebou a kódování barev prováděl sám. Program pracoval místo s barvami s celými čísly. Například test pro zjištění, zda barva X je pestrá, byl napsán takto (PESTRA je logická proměnná, X je celočíselná proměnná, nepestré barvy jsou bílá, šedá a černá):

PESTRA := (X<>0) **and** (X<>6) **and** (X<>7)

Z takto zapsaného programu nelze bez pomocné tabulky nebo velmi pečlivého textu v poznámkách vyčíst jeho význam. Při kódování barev se navíc snadno dělají chyby a program se špatně ladí.

Obdobných případů, kdy chceme zpracovávat počítačem objekty, které tvoří *konečnou*, popř. *uspořádanou množinu*, je mnoho:

- dny v týdnu (pondělí, úterý, středa, čtvrtek, pátek, sobota, neděle),
- barvy na kartách (kříže,listy,kára,srdce),
- typy aut (Trabant, Škoda, Lada, Mercedes, Rolls-Royce),
- pohádkové bytosti (Šmudla, Kejchal, Brumla, Stydlín, Štístko, Prófa, Rejpal, Sněhurka)

atd.

Je proto výhodnější umožnit zápis jmen objektů přímo do výrazu v pascalském programu, tj. test pestrosti barvy zapisovat např. takto:

PESTRA := (X<>BILA) **and** (X<>SEDA) **and** (X<>CERNA)

Program je přehlednější a čtivější i bez rozsáhlých komentářů.

Datový typ proměnné X bude popsán vyjmenováním všech jeho hodnot, proto se mu říká *výčtový typ*. Jeho hodnoty jsou reprezentovány jmény uvažovaných objektů (identifikátory). Popis typu provedeme vyjmenováním všech identifikátorů představujících jeho hodnoty, např.:

- (BILA, ZLUTA, MODRA, ZELENA, CERVENA, HNEDA, SEDA, CERNA),
- (PONDELI, UTERY, STREDA, CTVRTEK, PATEK, SOBOTA, NEDELE),
- (KRIZE, LISTY, KARA, SRDCE),
- (TRABANT, SKODA, LADA, MERCEDES, ROLLSROYCE),
- (SMUDLA, KEJCHAL, BRUMLA, STYDLIN, STISTKO, PROFA, REJPAL, SNEHURKA).

Stejně jako jsou pojmenovány standardní datové typy *integer*, *char* a *boolean*, mohou být pojmenovány i nové uživatelské typy jmény BARVY, DNY, KARTY, AUTA a POHADKA.

Deklarace typů je v programu umístěna mezi deklaraci konstant a deklaraci proměnných. Je uvedena vyhrazeným slovem **type** a má tvar:

type
identifikátor$_1$ = *typ*$_1$;
identifikátor$_2$ = *typ*$_2$;
:
identifikátor$_n$ = *typ*$_n$;

Typem může být popis nově vytvářeného typu nebo identifikátor dříve deklarovaného, nebo i standardního typu. Je tedy možné zapsat např.

```
type
  BARVY    = (BILA, ZLUTA, MODRA, ZELENA, CERVENA,
    HNEDA, SEDA, CERNA);
  DNY      = (PONDELI, UTERY, STREDA, CTVRTEK, PATEK,
    SOBOTA, NEDELE);
  KARTY    = (KRIZE, LISTY, KARA, SRDCE);
  AUTA     = (TRABANT, SKODA, LADA, MERCEDES,
    ROLLSROYCE);
  POHADKA  = (SMUDLA, KEJCHAL, BRUMLA, STYDLIN,
    STISTKO, PROFA, REJPAL, SNEHURKA);
  CELACISLA = integer;
  BARVICKY = BARVY;
```

Datový typ není nutné pojmenovávat. Popis datového typu je možné uvést přímo v deklaraci proměnné:

```
var
  KANASTA : (KRIZE, LISTY, KARA, SRDCE);
```

Výčtový typ je ordinálním typem. Uspořádání je dáno pořadím zápisu hodnot ve vyjmenovaném seznamu. První hodnota v seznamu je nejmenší a poslední největší. Ordinální čísla jsou hodnotám přiřazena postupně od 0 do $n-1$ (kde n je počet hodnot výčtového typu) rovněž podle pořadí v seznamu. Jsou definovány relační operace pro porovnání, funkce *pred*, *succ* a *ord*. Jiné operace nelze užít.

Rovněž *není možné* hodnoty výčtového typu získat přímo ze vstupu příkazem *read* nebo *readln* a předávat je na výstup příkazem *write* nebo *writeln*. Proto *není možné* napsat do programu příkaz (X je proměnná typu BARVY)

readln(X)

s tím, že při výskytu řetězce čtyř znaků BILA na vstupním zařízení se proměnné X přiřadí hodnota BILA. Rovněž není možné zapsat

writeln(X)

s tím, že při hodnotě proměnné X = ZLUTA se na výstupním zařízení objeví řetězec pěti znaků ZLUTA.

Příkladem výčtového typu může být také typ popisující goniometrické funkce sin, cos, tg a cotg.

```
type
  GONFUNKCE = (SINUS, COSINUS, TANGENS, COTANGENS);
```

Tohoto typu můžeme použít např. v příkazu case:

```
case FUNKCE of
  SINUS      : X := sin(Y);
  COSINUS    : X := cos(Y);
  TANGENS    : X := sin(Y)/cos(Y);
  COTANGENS  : X := cos(Y)/sin(Y);
end; { case }
```

FUNKCE je proměnnou datového typu GONFUNKCE a X,Y jsou reálné proměnné

```
var
  FUNKCE : GONFUNKCE;
  X, Y   : real;
```

Na základě hodnoty proměnné FUNKCE se provede výpočet jedné ze čtyř goniometrických funkcí. Argument musí být předem uložen v proměnné Y, výsledek bude v proměnné X.

Poznamenejme, že využití identifikátorů pascalských standardních funkcí *sin* a *cos* v popisu výčtového typu je principiálně možné:

```
type
  GONFUNKCE = ( sin, cos, TANGENS, COTANGENS);
```

avšak identifikátory *sin* a *cos* budou od této chvíle označovat jiný objekt než funkce sinus a cosinus a tyto standardní funkce již nebudeme moci v programu použít. Toto a podobné využití standardních identifikátorů pro označení nestandardních objektů programu se proto *nedoporučuje*.

Standardní datový typ *boolean* je vlastně výčtovým typem

```
type
  boolean = (false, true);
```

pomineme-li skutečnost, že pro něj navíc existují operace **not**, **and** a **or**.

5.3 Intervaly

Počet možných hodnot datového typu může být, zvláště u standardních typů, pro některé aplikace příliš velký. Patrné je to často u datového typu *integer* s rozsahem <– *maxint, maxint* >, kdy často využíváme jen nepatrný interval, např. od 0 do 127 ap. Zdálo by se, že využíváme-li pouze podmnožinu hodnot datového typu, není nutné tuto skutečnost v programu zdůrazňovat. Je-li program odladěn, je skutečně jedno, zda jsme užili pro definici datového typu celou dostupnou množinu hodnot, nebo jen její podmnožinu. Při ladění programu nám však definice podmnožiny hodnot umožňuje odhalit takové chyby, kdy výsledek výrazu má mít hodnotu z jisté podmnožiny hodnot, ale ve skutečnosti má hodnotu jinou a chybnou.

Mimo jiné i z těchto důvodů umožňuje Pascal deklarovat jako datové typy intervaly hodnot jiných ordinálních typů. Interval je popsán minimální a maximální hodnotou nového typu. Jsou to konstanty původního typu a nazýváme je *meze* intervalu. Interval pak obsahuje všechny hodnoty původního datového typu uvnitř mezí, včetně mezí samotných. Meze intervalu jsou v popisu typu odděleny dvěma tečkami. Uvažujeme-li např. výčtové typy uvedené jako příklady na počátku této kapitoly a standardní datové typy, můžeme psát:

```
type
  NEZAP    = 0 .. maxint;     { nezáporná – z integer }
  KLADNA   = 1 .. maxint;     { kladná – z integer }
```

```
CISLICE    = '0' .. '9';   { číslice – interval z char }
PESTRE     = ZLUTA .. HNEDA; { pestré barvy – z BARVY }
VIKEND     = SOBOTA .. NEDELE; {víkend – interval z DNY}
TRPASLICI  = SMUDLA .. REJPAL; { trpaslíci – z POHADKA }
```

Interval si zachovává všechny vlastnosti původního datového typu mimo rozsah hodnot a jméno. Ordinální čísla zůstávají rovněž původní (hodnotám datového typu interval nejsou přidělována ordinální čísla znovu od 0).

Pokud je při vytváření programu zřejmé, že některá proměnná bude nabývat hodnot pouze z intervalu ordinálního typu, budeme ji vždy deklarovat jako proměnnou datového typu interval. Nejčastěji je třeba vytvářet intervaly standardních datových typů *integer* a *char*.

Například v programu provádějícím umocňování celého čísla na nezáporný exponent

```
VYSLEDEK := 1;
for I := 1 to K do
  VYSLEDEK := VYSLEDEK * X;
```

budou deklarovány proměnné takto:

```
var
  VYSLEDEK, X  : integer;
  K            : NEZAP;    { interval <0, maxint> }
  I            : KLADNA;   { interval <1, maxint> }
```

Hodnota exponentu K nemůže být záporná a řídicí proměnná cyklu I nabývá pouze kladných hodnot.

5.4 Cvičení

1. Čísla čtená ze vstupu tvoří koeficienty polynomu

$$a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \ldots + a_{n-1}x + a_n.$$

Čísla jsou na vstupu v pořadí: $n\ a_0\ a_1\ a_2 \ldots a_{n-1}\ a_n$.

Hodnota n je celočíselná a koeficienty polynomu jsou reálná čísla. Vyhodnoťte polynom Hornerovým schématem a vytiskněte výsledek.

2. Vytvořte program, který zašifruje čtené znaky tak, že pro každý znak vypíše jeho ordinální číslo. Vytvořte program pro dešifrování (pro daná ordinální čísla bude vypisovat odpovídající znaky).
3. Program pro konverzi upravte tak, aby zadaný základ mohl být v rozsahu <2,16>.

Návod: Znaky pro vyjádření číslic s hodnotami 10 – 15 budou písmena

a = 10, b = 11, c = 12, d = 13, e = 14, f = 15. Například číslo

ff

má v soustavě se základem 16 hodnotu

$$15 * 16^1 + 15 * 16^0 = 255.$$

Zadáme-li tedy vstupní posloupnost znaků, např.

23a

a základ soustavy 11, bude výsledná hodnota vypočtena podle vztahu

$$2 * 11^2 + 3 * 11^1 + 10 * 11^0 = 285.$$

Znak *a* označuje v jedenáctkové soustavě hodnotu 10.

4. Vytvořte program pro výpočet hodnoty goniometrické funkce pro zadaný argument. Rozšiřte datový typ GONFUNKCE z předchozí kapitoly o další možné funkce a využijte popsaného příkazu case.
5. Vytvořte program, který bude provádět konverzi reálného čísla při vstupu. Program bude číst reálné číslo v desetinné notaci zapsané na jednom vstupním řádku. Předpokládejte, že před číslem může být znaménko + nebo –, popř. může být číslo i bez znaménka.
   Návod: Číslice za desetinnou tečkou je nutné převést na číselnou hodnotu (stejně jako u celých čísel). Číselné hodnoty budou postupně násobeny hodnotami 0.1, 0.01, 0.001 atd. Část čísla před desetinnou tečkou se převádí obvyklým způsobem.
6. Upravte příkl. 5 tak, aby bylo možno číst také reálná čísla v semilogaritmickém tvaru.

6 PROCEDURY A FUNKCE

Programy, které jsme doposud vytvářeli, nebyly svým rozsahem příliš velké. Řešení problémů bylo jednoduché a nepřesáhlo obvykle jednu stránku textu programu. Nyní postupně přejdeme ke složitějším programům, jejichž řešení bude vyžadovat i větší počet dílčích algoritmů, přičemž tyto algoritmy nebudou vždy jednoduché.

Je proto třeba mít k dispozici prostředky, které umožní přirozeně rozčlenit hlavní problém na dílčí podproblémy a dovolí tyto dílčí podproblémy samostatně řešit.

Metoda postupného rozčlenění hlavního problému na dílčí, samostatně definované úkony není samoúčelná. Pro rozsáhlejší programy je naopak nezbytná. Proto v této i dalších kapitolách zavedeme jazykové prostředky, které rozklad řešeného problému umožní. Jsou to *procedury* a *funkce*.

6.1 Procedury

Zkusme nyní řešit tento problém. Máme dáno datum ve tvaru

den
měsíc
rok

kde

- den je celé číslo v intervalu 1 .. 31,
- měsíc je celé číslo v intervalu 1 .. 12,
- rok je letopočet (budeme předpokládat, že je větší než 1923)

a budeme chtít vědět, o který den v týdnu jde. Výsledkem programu bude tedy výpis, např.

17.7.1982 je sobota

Kromě toho chceme vědět pořadové číslo dne v zadávaném roce, tedy ještě výpis

Je to *xxx*. den roku

Od programu budeme požadovat, aby prováděl jednoduchou kontrolu

vstupních údajů a nepřipustil zadání hodnoty dne, měsíce a roku mimo povolené intervaly. Řešení tohoto problému není zcela jednoduché.

Určíme nejdříve den v minulosti, o kterém víme, jaký den v týdnu to byl (např. 1.1.1924 bylo úterý). Musíme také vědět, že nebudeme chtít zadávat jako vstupní údaje datum před tímto dnem. Pak stačí znát počet dnů, který uplynul od 1.1.1924 do zadaného data včetně. Vyhodnocením výrazu

počet dnů **mod** 7

získáme hodnoty z intervalu 0 .. 6, určující přímo dny v týdnu. Bylo-li 1.1.1924 úterý, pak výsledek

0 udává pondělí
1 udává úterý
2 udává středu
:
6 udává neděli

Kdyby počet dnů v jednotlivých měsících byl stejný (např. 31) a neexistovaly přestupné roky, bylo by možné vypočítat počet dnů, které uplynuly od 1.1.1924 do zadaného data

den. měsíc. rok

podle jednoduchého výrazu

počet dnů = den + (měsíc–1) * 31 + (rok–1924) * 365.

Bohužel je však počet dnů v jednotlivých měsících nepravidelný, a to

- leden, březen, květen, červenec, srpen, říjen a prosinec mají 31 dnů,
- duben, červen, září a listopad mají 30 dnů,
- únor má 28, v přestupném roce 29 dnů.

Počet dnů v roce je buď 365, nebo v přestupném roce 366 dnů. Proto pro výpočet počtu dnů nelze jednoduchý vztah využít. Připomeňme, že přestupný rok je ten, jehož letopočet je dělitelný 4, avšak nesmí být zároveň dělitelný 100. Je-li ale rok dělitelný 400 (např. rok 2000), pak je přestupným rokem, tj.

(rok **mod** 4 = 0) **and** (rok **mod** 100 < > 0) **or** (rok **mod** 400 = 0).

V našem programu rozdělíme problém do tří částí:

- *čtení a kontrola vstupních údajů*,
- *výpočet počtu dnů*, které uplynuly od známého data,
- *výpis výsledků*.

Čtení a kontrola vstupních údajů přečte vstupní údaje a naplní proměnné DEN, MESIC a ROK. Jejich hodnoty udávají zadané datum. *Výpočet počtu*

dnů na základě hodnot proměnných DEN, MESIC a ROK určí počet dnů, který uplynul od 1.1.1924 do zadaného data a toto číslo uloží do proměnné POCET. Rovněž vypočte pořadové číslo dne v roce a uloží je do proměnné POCETVROCE. *Výpis výsledků* na základě hodnot proměnných DEN, MESIC, ROK, POCET a POCETVROCE vypíše výsledky v požadovaném tvaru.

Kromě toho budeme chtít, aby se program po zpracování jednoho data neukončil, nýbrž dal uživateli možnost výpočtu dne v týdnu a pořadového čísla dne v roce i pro další vstupní údaje.

Užité proměnné můžeme deklarovat takto:

```
var
  DEN          : TYPDEN;     { den v měsíci }
  MESIC        : TYPMESIC;   { měsíc }
  ROK          : TYPROK;     { rok }
  POCET        : NEZAP;      { počet dnů od 1.1.1924 }
  POCETVROCE   : DENVROCE;   { pořadové číslo dne v roce }
```

Typy proměnných TYPDEN, TYPMESIC, TYPROK, NEZAP a DENVROCE budeme deklarovat později. Zatím vystačíme pouze s jejich identifikátory bez znalosti konkrétních typů.

Schématicky bychom náš program mohli nyní zapsat takto:

```
program DATUM (input, output);
type
  TYPDEN       = ... ;    { typ dne v měsíci }
  TYPMESIC     = ... ;    { typ měsíce }
  TYPROK       = ... ;    { typ roku }
  NEZAP        = ... ;    { nezáporné číslo }
  DENVROCE     = ... ;    { typ pořadového čísla dne v roce }
var
  DEN          : TYPDEN;     { den v měsíci }
  MESIC        : TYPMESIC;   { měsíc }
  ROK          : TYPROK;     { rok }
  POCET        : NEZAP;      { počet dnů od 1.1.1924 }
  POCETVROCE   : DENVROCE;   { pořadové číslo dne v roce }
  POKRACUJ     : integer;    { příznak pokračování }
begin { DATUM }
  repeat { dokud má uživatel zájem }
    čtení vstupních údajů;
      { naplní hodnoty proměnných DEN, MESIC, ROK }
```

```
    výpočet počtu dnů a pořadí;
      { z hodnot proměnných DEN, MESIC a ROK vypočte
        hodnoty proměnných POCET a POCETVROCE }
    výpis výsledků;
      { na základě hodnot proměnných DEN, MESIC, ROK,
        POCET a POCETVROCE vypíše výsledky }
    writeln ('Chces pokracovat (0 = ne, 1 = ano)');
    readln (POKRACUJ);
  until POKRACUJ = 0;
end. { DATUM }
```

Hlavní řídicí strukturou programu je cyklus repeat (podle zadaných údajů se provede alespoň jeden výpočet). V těle cyklu se postupně provádějí všechny tři požadované činnosti (čtení, výpočet a výpis). Poslední dva příkazy *writeln* a *readln* umožní uživateli zadat hodnotu proměnné POKRACUJ, která způsobí opakování těla cyklu (má-li nenulovou hodnotu), nebo cyklus ukončí (má-li nulovou hodnotu).

Popisy výpočtu zapsané kurzívou představují vlastně příkazy, které provádějí jistou akci (podobně jako standardní příkazy Pascalu). Tato akce je však na rozdíl od standardních příkazů definovaná tvůrcem programu a je nutné ji popsat (deklarovat).

Pascal umožňuje vytvářet nové příkazy, jejichž význam je popsán uživatelem formou procedur. Proceduru v nejjednodušší formě si můžeme představit jako složený příkaz, který je pojmenován identifikátorem. Identifikátoru procedury pak můžeme používat jako dalšího příkazu jazyka. Na místě, kde jej v programu uvedeme se provede požadovaný složený příkaz. Uvedení identifikátoru procedury jako příkazu nazýváme *příkazem* (někdy také *voláním*) *procedury*.

Deklarace procedury začíná *hlavičkou* (*záhlavím*) procedury

procedure *identifikátor*;

Pojmenovaný úsek činnosti (*tělo procedury*) je zapsán formou složeného příkazu ohraničeného příkazovými závorkami **begin** a **end**. Deklarace procedury končí středníkem. Deklarací procedury zavádíme její identifikátor jako nový pascalský příkaz. Zápis příkazu, skládajícího se pouze z identifikátoru procedury, způsobí provedení celého složeného příkazu zapsaného v těle procedury.

Pro náš příklad můžeme deklarovat proceduru pro *výpis výsledků*

```
procedure VYPISVYSLEDKU;
{ výpis výsledků v textovém tvaru }
begin { VYPISVYSLEDKU }
  write (DEN:1,'.',MESIC:1,'.',ROK:1,' je ');
  case POCET mod 7 of { podle výsledku mod 7 výpis jména dne }
    0: write ('pondeli');
    1: write ('utery');
    2: write ('streda');
    3: write ('ctvrtek');
    4: write ('patek');
    5: write ('sobota');
    6: write ('nedele');
  end; { case }
  writeln;
  writeln ('Je to ',POCETVROCE:1,'. den v roce');
  writeln;
end; { VYPISVYSLEDKU }
```

a zapisovat pak příkaz procedury

```
VYPISVYSLEDKU
```

který způsobí provedení celého těla procedury VYPISVYSLEDKU. Podobně lze pojmenovat a deklarovat i dvě zbývající části pro výpočet počtu dnů a pořadí v roce (procedura POCETDNU) a pro čtení vstupních údajů (procedura CTIDATUM).

Uvedený postup nám umožňuje rozšiřovat deklaracemi procedur základní množinu standardních příkazů Pascalu v rámci vytvářeného programu. Tím, že jazyk umožňuje deklarovat pojmenované úseky libovolné složitosti, poskytuje prostředek pro rozklad problému na podproblémy. Je dobré nejprve určit pouze popis činnosti procedury a její *vazbu* na okolí. Řešení jejího těla (neboli *jak* to má dělat) je lepší nechat na pozdější fázi návrhu.

I my jsme nejdříve určili popis činnosti jednotlivých procedur, jména a vazbu na okolí (tj. identifikátory a typy všech proměnných, kterými si budou procedury předávat údaje s okolím).

Můžeme si tedy strukturu programu DATUM zapsat s voláními jednotlivých procedur a s vyznačením, kde budou jednotlivé procedury deklarovány.

```
program DATUM (input, output);
type
  TYPDEN    = ... ;  { typ dne v měsíci }
```

```
    TYPMESIC    = ... ;  { typ měsíce }
    TYPROK      = ... ;  { typ roku }
    NEZAP       = ... ;  { nezáporné číslo }
    DENVROCE    = ... ;  { typ pořadového čísla dne v roce }
  var
    DEN             : TYPDEN;    { den v měsíci }
    MESIC           : TYPMESIC;  { měsíc }
    ROK             : TYPROK;    { rok }
    POCET           : NEZAP;     { počet dnů od 1.1.1924 }
    POCETVROCE      : DENVROCE;  { pořadové číslo dne v roce }
    POKRACUJ        : integer;   { příznak pokračování }

  procedure CTIDATUM;
  { čtení data }
  begin { CTIDATUM }
    :
  end; { CTIDATUM }

  procedure POCETDNU;
  { výpočet počtu dnů od 1.1.1924 }
  begin { POCETDNU }
    :
  end; { POCETDNU }

  procedure VYPISVYSLEDKU;
  { výpis výsledků v textovém tvaru }
  begin { VYPISVYSLEDKU }
    :
  end; { VYPISVYSLEDKU }

  begin { DATUM }
    repeat { dokud má uživatel zájem }
      CTIDATUM;          { čtení vstupních údajů }
      POCETDNU;          { výpočet počtu dnů a pořadí }
      VYPISVYSLEDKU;     { výpis výsledků }
      writeln ('Chces pokracovat (0=ne, 1=ano)');
      readln (POKRACUJ);
    until POKRACUJ = 0;
  end. { DATUM }
```

Deklarace procedur se umísťuje mezi deklaraci proměnných a tělo programu. Počet procedur může být libovolný a v těle procedury mohou být uvedeny další příkazy procedur (tj. volání dalších procedur). Vždy však musí platit (podobně jako u proměnných), že procedura užitá v příkazu procedury musí být před tímto místem deklarována.

Nyní již můžeme přistoupit k vlastní deklaraci procedur našeho programu. Dříve ovšem musíme konkrétně deklarovat jednotlivé datové typy, jinak bychom těla procedur nemohli zapsat.

```
type
  TYPDEN    = 1 .. 31;          { typ dne v měsíci }
  TYPMESIC  = 1 .. 12;          { typ měsíce }
  TYPROK    = 1924 .. maxint;   { typ roku }
  NEZAP     = 0 .. maxint;      { nezáporné číslo }
  DENVROCE  = 1 .. 366;         { typ pořadového čísla dne v roce }
```

Pro deklaraci typů jsme užili intervalů typu *integer*, neboť jednotlivé proměnné mohou nabývat pouze některých celočíselných hodnot (DEN hodnoty 1 až 31, MESIC hodnoty 1 až 12 atd.).

Procedura CTIDATUM provádí čtení zadaného data a jednoduchou kontrolu správnosti údajů.

```
procedure CTIDATUM;
{ čtení data }
begin { CTIDATUM }
  writeln ('Zadej den :');      { čtení dne v měsíci }
  repeat { dokud není zadána hodnota v intervalu 1 .. 31 }
    readln (HODNOTA);   { přečtení celočíselné hodnoty }
    if (HODNOTA < 1) or (HODNOTA > 31) then { test }
      writeln ('Zadej cislo v intervalu 1 .. 31');
  until (HODNOTA > = 1) and (HODNOTA < = 31);
  DEN : = HODNOTA;

  writeln ('Zadej mesic :');    { čtení měsíce v roce }
  repeat { dokud není zadána hodnota v intervalu 1 .. 12 }
    readln (HODNOTA);   { přečtení celočíselné hodnoty }
    if (HODNOTA < 1) or (HODNOTA > 12) then { test }
      writeln ('Zadej cislo v intervalu 1 .. 12');
  until (HODNOTA > = 1) and (HODNOTA < = 12);
  MESIC : = HODNOTA;
```

```
    writeln ('Zadej rok :');        { čtení roku }
    repeat { dokud není zadána hodnota >= 1924 }
      readin (HODNOTA);   { přečtení celočíselné hodnoty }
      if HODNOTA < 1924 then { test }
        writeln ('Zadej cislo vetsi nez 1923');
    until HODNOTA >= 1924;
    ROK := HODNOTA;
  end; { CTIDATUM }
```

Procedura obsahuje tři velmi podobné úseky pro čtení dne, měsíce a roku. Každý z úseků vypíše nejprve požadavek, např.

Zadej den :

Následuje cyklus repeat, který obsahuje čtení celočíselné hodnoty do proměnné HODNOTA. Není-li přečtená hodnota v požadovaném intervalu, vypíše se varovné hlášení, např.

Zadej číslo v intervalu 1 .. 31

a tělo cyklu se zopakuje.

Je-li hodnota v požadovaném intervalu, např. platí-li

(HODNOTA >= 1) **and** (HODNOTA <= 31)

cyklus se ukončí. Po ukončení cyklu obsahuje proměnná HODNOTA zkontrolovanou hodnotu, kterou je možno uložit do proměnné DEN, MESIC nebo ROK datového typu interval s tím, že chyba při běhu programu nemůže nastat (ukládaná hodnota bude vždy ležet v požadovaném intervalu).

Jádrem programu je procedura POCETDNU pro výpočet počtu dnů, které uplynuly od 1.1.1924 do zadaného data, a pořadového čísla dne v roce.

```
procedure POCETDNU;
{ výpočet počtu dnů od 1.1.1924 }
begin { POCETDNU }

  { 1. úsek }
  POCET: = DEN;      { nejprve se vloží den v měsíci }

  { 2. úsek }
  I := MESIC - 1;      { počet celých měsíců }
  POCET := POCET + I * 31;
  { odečtení počtu měsíců s menším počtem dnů }
  if MESIC <= 8 then    { leden až srpen }
```

```
    POCET := POCET - I div 2
  else  { září až prosinec }
    POCET := POCET - (I-1) div 2;
  if MESIC > 2 then
    begin { úprava počtu dní února }
      if (ROK mod 4 =0) and (ROK mod 100 < >0)
        or (ROK mod 400 = 0) then
        { přestupný rok }
        POCET := POCET - 1    { 31 - 1 - 1 = 29 }
      else { nepřestupný rok }
        POCET := POCET - 2;   { 31 - 1 - 2 = 28 }
    end; { úprava počtu dní února }
  POCETVROCE := POCET; { nyní je vypočteno pořadové číslo }

  { 3. úsek }
  { výpočet počtu dnů v proběhlých letech }
  I:= (ROK-1924); { počet roků }
  { redukce přestupných let }
  POCET := POCET + (I+3)div 4 - (I+23)div 100
  + (I+323) div 400 + I * 365;
end; { POCETDNU }
```

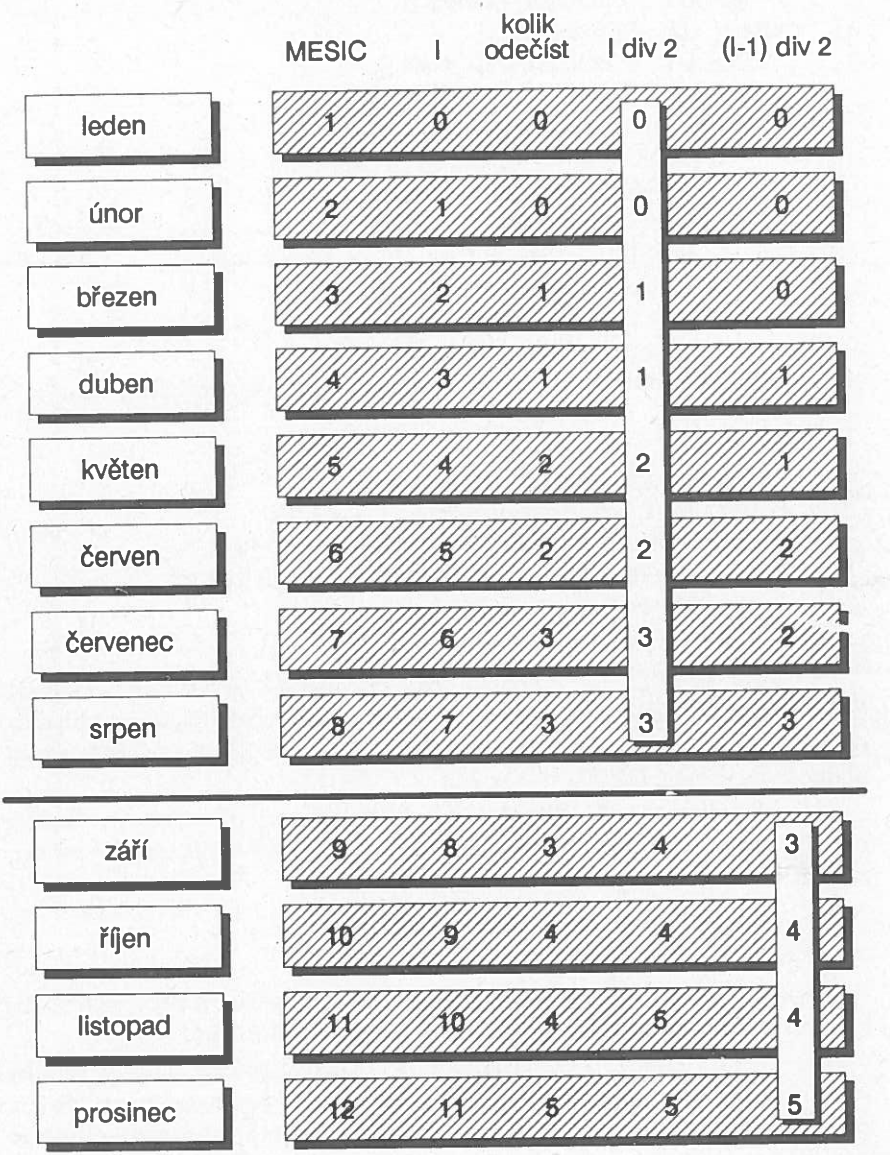
Výpočet se skládá ze tří úseků, přičemž celkový počet dnů se ukládá do proměnné POCET. V prvním úseku se vypočte počet dnů, které uplynuly od začátku zadaného měsíce do zadaného data včetně. Tento počet je roven hodnotě proměnné DEN.

V druhé části se vypočte počet dnů, které uplynuly od začátku zadaného roku do začátku zadaného měsíce. Nejdříve se předpokládá, že všechny měsíce mají 31 dnů a provede se jednoduchý výpočet

```
I := MESIC - 1; { počet celých měsíců }
POCET := POCET + I * 31;
```

Nyní je potřeba odečíst jeden den pro všechny měsíce s počtem dnů 30. Pro jednoduchost zatím předpokládáme, že i únor má 30 dnů.

Na obr. 6.1 vidíme vypočtené hodnoty proměnných MESIC a I pro jednotlivé měsíce. Vidíme také, o kolik je třeba snížit původní počet dnů (v březnu o 1 za únor, v dubnu o 1 za únor, v květnu o 2 za únor a duben atd.). Potíže způsobují letní měsíce červenec a srpen, které porušují pravidelné střídání delších a kratších měsíců. Hodnotu pro odečtení vyjadřuje od ledna do srpna (pro MESIC < = 8) výraz



Obr. 6.1 Úprava počtu dnů pro jednotlivé měsíce

I **div** 2

a od září do prosince (pro MESIC > 8) výraz

(I–1) **div** 2

Odečtení bude tedy provedeno příkazem

```
if MESIC < = 8 then    { leden až srpen }
   POCET : = POCET – I div 2
else                   { září až prosinec }
   POCET : = POCET – (I–1) div 2;
```

Je-li uvažovaný měsíc březen nebo některý další, je nutné provést výpočet určující počet dnů února. V předchozím podmíněném příkazu jsme předpokládali, že únor má 30 dnů. Proto se odečetl z počtu dnů jeden den. Nyní je třeba zjistit, zda jde o přestupný rok. V případě přestupného roku je nutno odečíst ještě jeden den (tím se dosáhne správného počtu 29). Není-li rok přestupný, je nutné odečíst 2 a tím dosáhnout počtu dnů 28.

```
if MESIC > 2 then
   begin { úprava počtu dní února }
      if (ROK mod 4 =0) and (ROK mod 100 < >0)
         or (ROK mod 400 = 0) then
         { přestupný rok }
         POCET : = POCET – 1    { 31 – 1 – 1 = 29 }
      else { nepřestupný rok }
         POCET : = POCET – 2;   { 31 – 1 – 2 = 28 }
   end; { úprava počtu dní února }
```

V tomto okamžiku obsahuje proměnná POCET pořadové číslo dne v roce. Tato hodnota se může uložit do proměnné POCETVROCE.

```
POCETVROCE : = POCET; { nyní je vypočteno pořadové číslo }
```

Zbývá vypočítat počet dnů v letech, které uplynuly od roku 1924 do začátku zadaného roku. Počet let je nutné vynásobit počtem dnů v roce (tj. 365) a zvýšit o počet přestupných roků, které v uvažovaném období proběhly.

```
I: = (ROK–1924); { počet roků }
{ přičtení dnů přestupných let }
POCET : = POCET + (I+3)div 4 – (I+23)div 100 +
(I+323 )div 400 + I * 365;
```

Výraz ROK–1924 udává počet celých let od 1.1.1924 do zadaného data. V tomto období bylo

(ROK–1924 + 3) **div** 4

roků, jejichž letopočet je dělitelný 4 (tolik je třeba přičíst, protože to byly přestupné roky). Tento výraz zaručuje, že pro zadaný rok 1924 se nebude přičítat žádný den za přestupný rok, pro léta 1925, 1926, 1927, 1928 se bude přičítat den za přestupný rok 1924, pro léta 1929, 1930, 1931, 1932 dva dny za přestupné roky 1924 a 1928 atd.

Další část výrazu

(ROK–1924 + 23) **div** 100

udává počet roků, které uplynuly od roku 1924 a jejichž letopočet je dělitelný 100 (tolik je třeba odečíst, protože roky dělitelné stem nejsou přestupné a byl jim v předchozím výrazu přičten neoprávněně den navíc). Pro zadané letopočty 1924 až 2000 se nebude odečítat nic. Pro letopočty 2001 až 2100 se bude odečítat jednička, pro letopočty 2101 až 2200 dvojka atd.

Nakonec je nutné provést ještě úpravu pro roky dělitelné 400, které jsou opět přestupné. Následující část výrazu

(ROK–1924 + 323) **div** 400

udává počet roků, které uplynuly od roku 1924 a jejichž letopočet je dělitelný 400 (tolik je třeba přičíst, protože jim v předchozím výrazu byl odečten neoprávněně den navíc). Pro zadané letopočty 1924 až 2000 se nebude přičítat žádný den. Pro letopočty 2001 až 2400 jeden den atd.

Na konci procedury POCETDNU je v proměnné POCET množství dnů od 1.1.1924 do zadaného data včetně.

Procedura VYPISVYSLEDKU, která kromě tisku textů a celočíselné hodnoty POCETVROCE provádí převod počtu dnů na den v týdnu, již byla uvedena. K jejímu algoritmu poznamenejme, že na rozdíl od měsíců s různým počtem dnů mají týdny vždy 7 dní. Vyčíslením výrazu

POCET **mod** 7

získáme hodnoty 0 .. 6, které v příkazu case způsobí výpis příslušného textu.

Na závěr čl. 6.1 uvedeme úplný tvar programu.

```
program DATUM (input, output);
{ Program ze zadaného data vypočte, o který den v týdnu se jedná a také
pořadové číslo dne v roce. Je možno zadávat data od 1.1.1924.
  Vstup: Po výzvách
       Zadej den:
       Zadej mesic:
```

Zadej rok:

se vždy na nový řádek zadají číslice udávající datum. Vstupní hodnoty se kontrolují na správnost (den musí ležet v intervalu 1 .. 31, měsíc v intervalu 1 .. 12 a rok musí být větší než 1923). Při zadání hodnoty mimo interval se vypíše varovné hlášení

Zadej číslo v intervalu *xx .. yy*

a je nutné zapsat znovu správnou hodnotu. Po ukončení výpočtu se vypíše dotaz

Chces pokracovat (0 = ne, 1 = ano)

Nulová odpověď program ukončí, jiná odpověď způsobí opakování výpočtu.

Výstup:

xx.xx.xxxx je *xxxxxxx*

Je to *xxx*. den v roce

}

```
type
  TYPDEN    = 1 .. 31;          { typ dne v měsíci }
  TYPMESIC  = 1 .. 12;          { typ měsíce }
  TYPROK    = 1924 .. maxint;   { typ roku }
  NEZAP     = 0 .. maxint;      { nezáporné číslo }
  DENVROCE  = 1 .. 366;         { typ pořadového čísla dne v roce }

var
  DEN       : TYPDEN;           { den v měsíci }
  MESIC     : TYPMESIC;         { měsíc }
  ROK       : TYPROK;           { rok }
  POCET     : NEZAP;            { počet dnů od 1.1.1924 }
  POCETVROCE : DENVROCE;        { pořadové číslo dne v roce }
  POKRACUJ  : integer;          { příznak pokračování }
  HODNOTA   : integer;          { právě přečtená a zkontrolovaná hodnota }
  I         : integer;          { pomocná proměnná }
```

```
procedure CTIDATUM;
{ čtení data }
begin { CTIDATUM }
  writeln ('Zadej den :'); { čtení dne v měsíci }
  repeat { dokud není zadána hodnota v intervalu 1 .. 31 }
    readln (HODNOTA); { přečtení celočíselné hodnoty }
    if (HODNOTA<1) or (HODNOTA>31) then { test }
      writeln ('Zadej cislo v intervalu 1 .. 31');
  until (HODNOTA>=1) and (HODNOTA<=31);
  DEN:= HODNOTA;
  writeln ('Zadej mesic :'); { čtení měsíce v roce }
  repeat { dokud není zadána hodnota v intervalu 1 .. 12 }
    readln (HODNOTA); { přečtení celočíselné hodnoty }
    if (HODNOTA<1) or (HODNOTA>12) then { test }
      writeln ('Zadej cislo v intervalu 1 .. 12');
  until (HODNOTA>=1) and (HODNOTA<=12);
  MESIC:=HODNOTA;
  writeln ('Zadej rok :'); { čtení roku }
  repeat { dokud není zadána hodnota >= 1924 }
    readln (HODNOTA); { přečtení celočíselné hodnoty }
    if HODNOTA<1924 then { test }
      writeln ('Zadej cislo vetsi nez 1923');
  until HODNOTA>=1924;
  ROK:= HODNOTA;
end; { CTIDATUM }

procedure POCETDNU;
{ výpočet počtu dnů od 1.1.1924 }
begin { POCETDNU }
  POCET:=DEN;      { nejprve se vloží den v měsíci }
  I:= MESIC-1; { počet celých měsíců }
  POCET:= POCET + I * 31;
  { odečtení počtu měsíců s menším počtem dnů }
  if MESIC <= 8 then   { leden až srpen }
    POCET:= POCET - I div 2
  else          { září až prosinec }
    POCET:= POCET - (I-1) div 2;
  if MESIC > 2 then
    begin { úprava počtu dní února }
```

```
        if (ROK mod 4 =0) and (ROK mod 100 < >0)
          or (ROK mod 400 = 0) then
          { přestupný rok }
          POCET := POCET - 1    { 31 - 1 - 1 = 29 }
        else { nepřestupný rok }
          POCET := POCET - 2;   { 31 - 1 - 2 = 28 }
      end; { úprava počtu dní února }
    POCETVROCE := POCET; { nyní je vypočteno pořadové číslo }
    { výpočet počtu dnů v proběhlých letech }
    I:= (ROK-1924); { počet roků }
    { přičtení počtu přestupných let }
    POCET := POCET + (I+3)div 4 - (I+23)div 100 +
          (I+323)div 400 + I * 365;
end; { POCETDNU }

procedure VYPISVYSLEDKU;
{ výpis výsledků v textovém tvaru }
begin { VYPISVYSLEDKU }
  write (DEN:1,'.',MESIC:1,'.',ROK:1,' je ');
  case POCET mod 7 of { podle výsledku mod 7 výpis jména dne }
    0: write ('pondeli');
    1: write ('utery');
    2: write ('streda');
    3: write ('ctvrtek');
    4: write ('patek');
    5: write ('sobota');
    6: write ('nedele');
  end; { case }
  writeln;
  writeln ('Je to ',POCETVROCE:1,'. den v roce');
  writeln;
end; { VYPISVYSLEDKU }

begin { DATUM }
  repeat { dokud má uživatel zájem }
    CTIDATUM; { čtení vstupních údajů }
    POCETDNU; { výpočet počtu dnů a pořadí }
    VYPISVYSLEDKU; { výpis výsledků }
    writeln ('Chces pokracovat (0 = ne, 1 = ano)');
```

```
    readln (POKRACUJ);
  until POKRACUJ = 0;
end. { DATUM }
```

Program 6.1

6.2 Bloková struktura programu

Algoritmus realizující tělo procedury nemusí být vždy jednoduchý. Může se ukázat, že kromě proměnných pro předávání údajů bude nutné k jeho zápisu ve formě příkazů užít i jiných objektů programu (proměnných nebo i dalších procedur). Povšimněme si např. proměnné HODNOTA datového typu *integer* v programu 6.1, která je pomocnou proměnnou procedury CTIDATUM a pro zbytek programu není třeba. Deklarujeme-li ji mezi proměnnými programu, porušujeme celistvost a izolovanost procedury, kterou bychom rádi zapisovali jako celek, jehož jedinou vazbou na okolí jsou vstupní a výstupní údaje. Takovými soukromými objekty procedury nemusí být pouze proměnné. Mohou to být také konstanty, datové typy, popř. také procedury vzniklé při rozkladu složitého algoritmu těla procedury na podproblémy.

Soukromé objekty procedury by neměly být použitelné mimo ni a jejich jména by neměla kolidovat s ostatními jmény objektů v programu. Například by mělo být možné napsat

```
program PRIKLAD (input, output);
var
  I : integer;
procedure PRVNI;
var
  I : integer;
begin { PRVNI }
  :
end; { PRVNI }
procedure DRUHA;
var
  I : integer;
begin { DRUHA }
  :
end; { DRUHA }
begin { PRIKLAD }
  :
end. { PRIKLAD }
```

V programu PRIKLAD jsou tři celočíselné proměnné I. První z nich je proměnnou programu, druhá soukromou proměnnou procedury PRVNI a třetí soukromou proměnnou procedury DRUHA. Přestože se jmenují stejně, označují různé objekty.

To řeší Pascal *blokovou strukturou programu.*

Blokem se nazývá část programu obsahující deklarace společně se složeným příkazem

const
  deklarace konstant
type
  deklarace datových typů
var
  deklarace proměnných
procedure PROCEDURA1;
  blok;
procedure PROCEDURA2;
  blok;
:
begin
  tělo bloku
end

Kromě složeného příkazu obsahujícího *tělo bloku* může libovolná z uvedených částí bloku chybět.

Program, tak jak jsme jej popsali v kap.2, je vlastně blokem uvedeným hlavičkou

program *identifikátor* (*parametry programu*);

a ukončený tečkou.

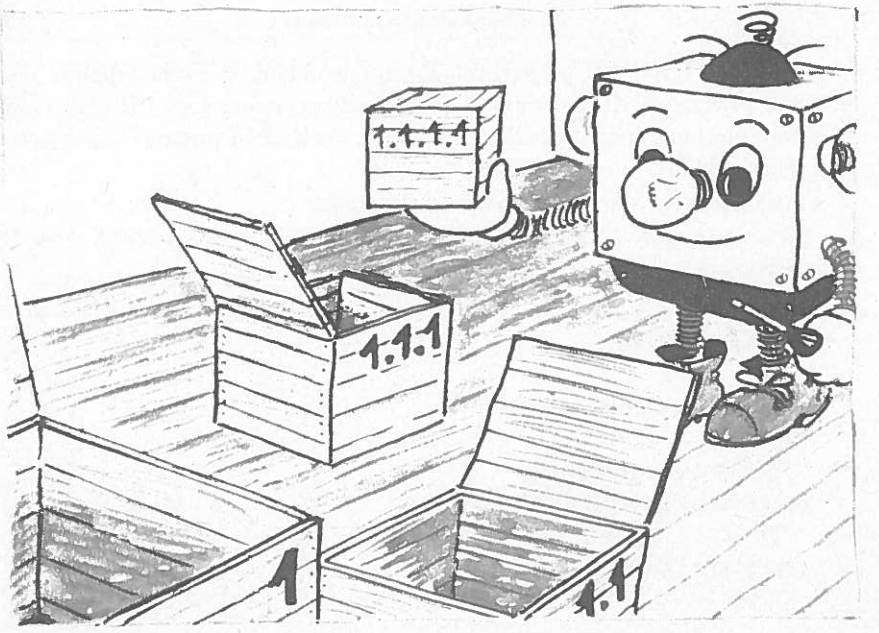
Deklarace procedury má v Pascalu tvar analogický. Pouze hlavička procedury je jiná (prozatím jsme poznali hlavičku procedury bez parametrů):

procedure *identifikátor*;

a deklarace je ukončena středníkem.

Tím, že blok může obsahovat deklaraci procedury, která je opět blokem, může dojít ke vnořování bloků do sebe. Program je blokem na úrovni 1, může obsahovat deklarace procedur na úrovni 2 atd., až do úrovně omezené vlastnostmi konkrétního překladače Pascalu.

Objekty deklarované v jistém bloku mohou být použity pouze v témže bloku (tzn. i v blocích procedur nižší úrovně, které jsou v něm deklarovány). Mimo



blok nejsou viditelné; je tam možno deklarovat jiné objekty se stejnými jmény. V rámci deklarační části jednoho bloku není dovoleno využít jednoho identifikátoru několikrát pro označení různých objektů. Nelze také týž objekt deklarovat vícekrát. Uděláme-li to, dojde k chybě při překladu programu. Ve vnitřních blocích je možno deklarovat objekty týchž jmen, které byly deklarovány na úrovni vyšší. Nastane-li taková situace, platí ta deklarace, která je místu použití nejblíže nadřazená.

Objekt použitý v bloku své deklarace se nazývá *lokální* objekt. Objekt použitý v bloku, jenž je vložen do bloku jeho deklarace se nazývá *nelokální* objekt. Objekt deklarovaný na úrovni 1 (tj. na úrovni programu) nazýváme objekt *globální*.

Program 6.2. ukazuje využití blokové struktury programu a zejména pravidlo viditelnosti objektů, v tomto konkrétním případě proměnných.

```
program BLOKOVASTRUKTURA;
{ Program dokumentuje blokovou strukturu programu. Každá proměnná
je v programu označena čísly,které umožňují rozlišit různé proměnné
označené stejnými identifikátory.

Program není funkční !
}
var
   A{11} : integer;
   B{11} : integer;
   C{11} : integer;

procedure P21;
var
   A{21} : integer;
   B{21} : integer;
begin { P21 }
   B{21}  := 1;
   C{11}  := B{21} + A{21}; { C je globální proměnná }
end; { P21 }

procedure P22;
var
   A{22} : integer;
   C{22} : integer;

   procedure P31;
   var
      A{31} : integer;
      B{31} : integer;
   begin { P31 }
      A{31}  := 1;
      C{22}  := B{31} + A{31}; { C je nelokální proměnná }
   end; { P31 }

begin { P22 }
   A{22}  := 1;
   C{22}  := B{11} + A{22}; { B je globální proměnná }
end; { P22 }
```

```
begin { P11 }
  A{11} := 1;
  B{11} := 1;
  C{11} := B{11};
end. { P11 }
```

Program 6.2

Komentář vyjadřuje vždy úroveň deklarace a pořadí bloku v této úrovni ({22} je druhá úroveň, druhý blok). Takto je vyjádřen vztah deklarací a použití objektů v programu. Do běžného programu se tyto komentáře nezapisují. Při složitější blokové struktuře může pak být nalezení odkazovaného objektu v programu obtížné. Proto raději nepoužíváme stejné identifikátory ani tam, kde nám to jazyk dovoluje.

Ukazuje se, že pro většinu případů stačí dvě deklarační úrovně,a to první úroveň programu a jeho proměnné, typy a konstanty (*globální*) viditelné ve všech procedurách, a druhá úroveň deklarovaných procedur s lokálními proměnnými, typy a konstantami.

Například proměnnou HODNOTA v proceduře CTIDATUM je možné deklarovat jako lokální, stejně jako proměnnou I využívanou lokálně v proceduře POCETDNU programu 6.1.

```
procedure CTIDATUM;
{ čtení data }
var
  HODNOTA : integer; { právě přečtená a zkontrolovaná hodnota }
begin { CTIDATUM }
  :
end; { CTIDATUM }

procedure POCETDNU;
{ výpočet počtu dnů od 1.1.1924 }
var
  I : integer; { pomocná proměnná }
begin { POCETDNU }
  :
end; { POCETDNU }
```

V programu je tedy možné globální proměnnou HODNOTA zrušit; stejně tak i globální proměnnou I deklarovanou nyní v proceduře POCETDNU.

6.3 Parametry

Procedury v té nejjednodušší formě, jak jsme je dosud poznali, spolupracují s okolím tak, že vybírají své vstupní hodnoty z nelokálních proměnných a své výstupní hodnoty do nelokálních proměnných opět ukládají. Každá procedura proto vyžaduje (chce-li vyměňovat hodnoty se svým okolím) jistý počet nelokálních proměnných pro komunikaci. Příkazy pracují v těle procedury přímo s nelokálními proměnnými.

Tento způsob spolupráce procedur s okolím přes proměnné je v Pascalu sice možný, nicméně existuje i další, často výhodnější možnost.

Uvažujme např. proceduru pro umocňování

$$Z = X^K$$

celočíselné hodnoty X na nezápornou hodnotu K. S dosavadními poznatky o procedurách ji můžeme deklarovat např. takto :

```
procedure UMOC;
{ umocňování Z = X umocněno na K }
var
  I : NEZAP;          { řídicí proměnná cyklu }
begin { UMOC }
  Z:=1;               { počáteční hodnotou pro násobení je 1 }
  for I:= 1 to K do
    Z:= Z * X;        { násobení základem }
end; { UMOC }
```

Vidíme, že tato procedura je jednoúčelová. Umí totiž pracovat pouze s proměnnými X, K a Z. Potřebujeme-li však nalézt mocninu hodnot

$$C = A^B$$

stejných datových typů, nastanou potíže. Buďto musíme deklarovat novou proceduru se stejným algoritmem, avšak jinými komunikačními proměnnými (to je nevýhodné z hlediska velikosti programu) nebo je nutné před voláním procedury kopírovat obsahy proměnných A a B do proměnných X a K. Po provedení procedury je opět nutné kopírovat výsledek z proměnné Z do proměnné C. To je sice obvykle méně náročné z hlediska velikosti programu, avšak provádí se zase více příkazů.

Bylo by tudíž praktické mít možnost přiřadit proceduře proměnné, s nimiž bude komunikovat, až v okamžiku jejího volání. Tělo procedury v deklaraci může pak být zapsáno obecněji, bez vazby na konkrétní proměnné programu.

Procedura je modifikována konkrétními proměnnými až v okamžiku volání. Této možnosti zápisu říkáme *parametrizace procedury.*

Parametrizace procedury umožňuje zapsat vedle jména procedury v deklaraci i seznam všech vstupních i výstupních hodnot i proměnných, se kterými procedura pracuje, a jejich datových typů. Tyto hodnoty a proměnné jsou pojmenovány identifikátory, podobně jako řádné proměnné. Takto zapsaným vstupním a výstupním hodnotám říkáme *parametry.* V těle procedury užíváme místo jmen konkrétních proměnných a konkrétních hodnot jména parametrů.

V záhlaví deklarace procedury v okrouhlých závorkách je uveden seznam vstupních i výstupních parametrů procedury. Říkáme jim *formální parametry* pro rozlišení od *skutečných parametrů* zadávaných při volání procedury. Jednotlivé formální parametry jsou odděleny středníkem.

Uvažujme např. proceduru UMOC, nyní zapsanou s parametry:

```
procedure UMOC(ZAKLAD: integer; EXPON: NEZAP;
                        var VYSL: integer);
{ umocňování VYSL = ZAKLAD umocněno na EXPON }
var
  I : NEZAP;
begin { UMOC }
  VYSL: = 1;          { počáteční hodnotou pro násobení je 1 }
  for I: = 1 to EXPON do
    VYSL: = VYSL * ZAKLAD; { násobení základem }
end; { UMOC }
```

V hlavičce procedury je možné sloučit deklaraci několika formálních parametrů téhož datového typu podobně jako při deklaraci proměnných. Například v proceduře pro součet dvou komplexních čísel by mohla hlavička vypadat takto (KOMPLEX je typ komplexního čísla):

```
procedure SOUCET (A,B: KOMPLEX; var C: KOMPLEX);
```

Každý formální parametr (podobně jako proměnná) má určen datový typ zápisem identifikátoru typu za dvojtečkou, např.

```
var VYSL: integer
```

V záhlaví procedury není možné uvádět celé popisy datových typů, pouze *identifikátory* datových typů. Není proto možné deklarovat formální parametr A typu interval takto:

```
procedure PRIKLAD (A: 1..100);
```

nýbrž je třeba datový typ z hlavičky procedury deklarovat samostatně, např.

```
type
  INTERVAL = 1..100;

procedure PRIKLAD (A: INTERVAL);
begin { PRIKLAD }
  :
end; { PRIKLAD }
```

Vstupní parametry speciálně neoznačujeme

```
ZAKLAD: integer;
EXPON : NEZAP;
```

zatímco *výstupní parametry* označujeme již známým vyhrazeným slovem **var**:

```
var VYSL: integer;
```

Formální parametry procedury můžeme použít v těle procedury stejným způsobem jako proměnné stejného datového typu, ale pouze v těle té procedury, v jejímž záhlaví se nacházejí. Říkáme, že jejich platnost je *lokální* v těle procedury.

Tělo procedury UMOC s formálními parametry se (až na záměnu jmen X, K a Z za ZAKLAD, EXPON a VYSL) neliší od těla téže procedury bez parametrů, má však z hlediska programu jiný význam. Zatímco v prvním případě se procedura UMOC volala příkazem

```
UMOC
```

a její tělo popisovalo akce s konkrétními proměnnými X,K a Z, druhý případ vyžaduje volání příkazem

```
UMOC(X,Y,Z)
```

nebo

```
UMOC(A,B,C)
```

Teprve příkaz procedury definitivně určí, se kterými skutečnými proměnnými bude tělo procedury pracovat. Proměnným v závorkách příkazu volání procedury říkáme *skutečné parametry*. Skutečný parametr nahradí při provedení příkazu procedury ten formální parametr, který je v hlavičce procedury zapsán na odpovídající pozici (první skutečný nahradí první formální atd.). Skutečné parametry jsou v příkazu volání procedury odděleny čárkou.

Skutečným *vstupním* parametrem procedury je *výraz*. Proměnná nebo konstanta, stojící v parametru samostatně, je rovněž výrazem. Oba tyto objekty mohou tedy také být skutečným vstupním parametrem procedury. Je požadován takový datový typ výrazu, který by bylo možno přiřadit do proměnné

stejného datového typu, jako má odpovídající formální parametr.

Skutečným *výstupním* parametrem *musí* být *proměnná* téhož typu, který je uveden u odpovídajícího formálního parametru.

Uvažujeme-li tedy proceduru UMOC s hlavičkou

```
procedure UMOC( ZAKLAD   : integer;
                EXPON    :NEZAP;
                var VYSL : integer);
```

je možné jako skutečné vstupní parametry ZAKLAD a EXPON zadat i celočíselné výrazy. Skutečným výstupním parametrem VYSL musí být vždy jen celočíselná proměnná. Předpokládáme-li existenci celočíselných proměnných X, Y a Z, může mít příkaz procedury UMOC např. tvar

```
UMOC((X + Y) * 2, X div 3, Z)
```

nebo

```
UMOC( 2, 3, Z)
UMOC(Y + 1, 2, X)
```

avšak není přípustný zápis

```
UMOC( (X + Y) * 2, X div 3, Z + 1)
```

neboť skutečným výstupním parametrem VYSL nesmí být výraz Z + 1.

Zavedení parametrů má četné výhody. Procedura se stává víceúčelovou a její modifikace pro užití s konkrétními proměnnými daného typu je snadná. Dále procedury s parametry umožňují deklarovat v programu relativně nezávislé úseky, včetně jejich rozhraní pro komunikaci s okolím. Jejich přínosem je možnost rozčlenit problém na množství samostatných, snadno pochopitelných i realizovatelných dílů s jasnou vazbou na okolí a tělem, které je pro okolí procedury chápáno jako rozšíření příkazů jazyka.

Mimo to procedury umožňují opakující se úsek programu jednou deklarovat a pak jej podle potřeby vyvolat.

Styk procedury s okolím lze v Pascalu provádět jak nelokálními či globálními proměnnými, tak parametry. Většinou se dává přednost jasné definici vazby na okolí v parametrech, zvláště u složitějších programů a také (jak uvidíme v kap. 11) při použití abstraktních datových typů.

Kromě víceúčelovosti tím zajistíme proceduře nezávislost na ostatních deklaracích. Procedura také nemůže změnit hodnoty jiných proměnných, deklarovaných mimo ni, než těch, které jsou uvedeny jako výstupní parametry.

Shrňme nyní ještě jednou popis vstupních a výstupních parametrů. Parametry

- *vstupní* neoznačujeme,
- *výstupní* označujeme **var**.

Parametry označené vyhrazeným slovem **var** nazýváme také *parametry volané odkazem*. Mají tu vlastnost, že formální parametr nahrazuje v těle procedury vždy tu konkrétní proměnnou, která je pak uvedena jako skutečný parametr. Je-li formálnímu parametru v těle procedury přiřazena hodnota, *změní se* i hodnota skutečného parametru. Proto musí být skutečným parametrem tohoto typu vždy proměnná.

Parametry neoznačené nazýváme také *parametry volané hodnotou*. Mají tu vlastnost, že při volání procedury se vytvoří v proceduře lokální proměnná daného typu a do této proměnné se před začátkem provádění těla procedury přiřadí hodnota skutečného parametru. Po ukončení těla procedury se pracovní proměnná *zruší*. Proto je skutečným parametrem volaným hodnotou *výraz*, neboť se před použitím vyhodnotí a jeho hodnota se přiřadí. Parametru volanému hodnotou můžeme sice v těle procedury přiřazovat hodnotu a tuto hodnotu v těle procedury i používat, avšak mimo tělo procedury se přiřazení hodnoty neprojeví. Ve skutečnosti totiž pracujeme s lokální pomocnou proměnnou, která po provedení těla procedury zanikne. Proto je nutné věnovat pozornost chybám, kdy bylo opomenuto deklarovat výstupní parametr jako parametr volaný odkazem. Parametr volaný hodnotou nemůže okolí procedury změnit a ta pak nedává očekávaný výsledek.

Pokud bychom např. zapomněli k výstupnímu parametru VYSL procedury UMOC připsat označení **var**

```
procedure UMOC(ZAKLAD: integer; EXPON: NEZAP;
                          VYSL: integer);
{ umocňování VYSL = ZAKLAD umocněno na EXPON }
begin { UMOC }
:
end; { UMOC }
```

bude parametru VYSL při volání procedury přiřazena výsledná hodnota. Protože však jde o parametr volaný hodnotou, hodnota skutečného parametru (přestože půjde o proměnnou) se *nezmění*.

Vidíme, že svými vlastnostmi jsou parametry volané odkazem určeny k tomu, aby byly užívány jako výstupní, a parametry volané hodnotou jako vstupní.

Je-li ovšem parametr dané procedury jak vstupní, tak výstupní, není nutné jej uvádět v záhlaví dvakrát, ale pouze jednou jako parametr volaný odkazem.

Program 6.3 je variantou programu uvedeného pod označením 6.1. Využívá však procedur s parametry. Zadání je shodné, proto jej neuvádíme.

```
program DATUM (input, output);
{ Úvodní poznámka shodná s programem 6.1 }
type
  TYPDEN    = 1 .. 31;          { typ dne v měsíci }
  TYPMESIC  = 1 .. 12;          { typ měsíce }
  TYPROK    = 1924 .. maxint;   { typ roku }
  NEZAP     = 0 .. maxint;      { nezáporné číslo }
  DENVROCE  = 1 .. 366;         { typ pořadového čísla dne v roce }
var
  DEN       : TYPDEN;           { den v měsíci }
  MESIC     : TYPMESIC;         { měsíc }
  ROK       : TYPROK;           { rok }
  POCET     : NEZAP;            { počet dnů od 1.1.1924 }
  POCETVROCE : DENVROCE;        { pořadové číslo dne v roce }
  POKRACUJ  : integer;          { příznak pokračování }

procedure CTIDATUM (var DEN   : TYPDEN;      { den v měsíci }
  var MESIC : TYPMESIC; var ROK : TYPROK);   {měsíc a rok }
{ čtení data }
var
  HODNOTA : integer; { právě přečtená a zkontrolovaná hodnota }

procedure CTIHODNOTU (DOLNI,HORNI:NEZAP);
{ čtení hodnoty s kontrolou intervalu }
begin { CTIHODNOTU }
  repeat { dokud není zadána hodnota v intervalu DOLNI..HORNI }
    readln (HODNOTA);   { přečtení celočíselné hodnoty }
    if (HODNOTA < DOLNI) or (HODNOTA > HORNI) then { test }
      writeln ('Zadej cislo v intervalu ',DOLNI:1, ' .. ',HORNI:1);
  until (HODNOTA > = DOLNI) and (HODNOTA < = HORNI);
end; { CTIHODNOTU }
begin { CTIDATUM }
  writeln ('Zadej den :');  { čtení dne v měsíci }
  CTIHODNOTU (1,31);
  DEN := HODNOTA;
  writeln ('Zadej mesic :'); { čtení měsíce v roce }
```

```
    CTIHODNOTU (1,12);
    MESIC := HODNOTA;
    writeln ('Zadej rok :'); { čtení roku }
    CTIHODNOTU (1924, maxint);
    ROK := HODNOTA;
end; { CTIDATUM }

procedure POCETDNU ( DEN    : TYPDEN; { den v měsíci }
                     MESIC : TYPMESIC; { měsíc v roce }
                     ROK    : TYPROK; { rok }
                     var POCET :NEZAP; { počet dnů od 1.1.1924 }
                     var POCETVROCE :DENVROCE); { pořadové číslo }
{ výpočet počtu dnů od 1.1.1924 }

var
    I : integer; { pomocná proměnná }
begin { POCETDNU }
    POCET:=DEN;      { nejprve se dá den v měsíci }
    I := MESIC - 1; { počet celých měsíců }
    POCET := POCET + I * 31;
    { odečtení počtu měsíců s menším počtem dnů }
    if MESIC < = 8 then    { leden až srpen }
      POCET := POCET - I div 2
    else                   { září až prosinec }
      POCET := POCET - (I-1) div 2;
    if MESIC > 2 then
      begin { úprava počtu dní února }
        if (ROK mod 4 =0) and (ROK mod 100 < >0)
          or (ROK mod 400 = 0) then
          { přestupný rok }
          POCET := POCET - 1    { 31 - 1 - 1 = 29 }
        else { nepřestupný rok }
          POCET := POCET - 2; { 31 - 1 - 2 = 28 }
      end; { úprava počtu dní února }
    POCETVROCE := POCET; { nyní je vypočteno pořadové číslo }
    { výpočet počtu dnů v proběhlých letech }
    I:= (ROK-1924); { počet roků }
    { přičtení počtu přestupných let }
    POCET := POCET + (I+3) div 4 - (I+23) div 100
```

```
            + (I+323)div 400 + I * 365;
  end; { POCETDNU }

  procedure VYPISVYSLEDKU (DEN : TYPDEN; { den v měsíci }
                           MESIC : TYPMESIC; { měsíc v roce }
                           ROK : TYPROK; { rok }
                           POCET : NEZAP; { počet dnů }
                           POCETVROCE : DENVROCE); { pořadí }
  { výpis výsledků v textovém tvaru }
  begin { VYPISVYSLEDKU }
    write (DEN:1,'.',MESIC:1,'.',ROK:1,' je ');
    case POCET mod 7 of { podle výsledku mod 7 výpis jména dne }
      0: write ('pondeli');
      1: write ('utery');
      2: write ('streda');
      3: write ('ctvrtek');
      4: write ('patek');
      5: write ('sobota');
      6: write ('nedele');
    end; { case }
    writeln;
    writeln ('Je to ',POCETVROCE:1,'. den v roce');
    writeln;
  end; { VYPISVYSLEDKU }

  begin { DATUM }
    repeat { dokud má uživatel zájem }
      CTIDATUM (DEN, MESIC, ROK); { čtení vstupních údajů }
      POCETDNU (DEN, MESIC, ROK, POCET,POCETVROCE);
      { výpočet počtu dnů a pořadí }
      VYPISVYSLEDKU (DEN, MESIC, ROK, POCET,
        POCETVROCE);
      { výpis výsledků }
      writeln ('Chces pokracovat (0 = ne, 1 = ano)');
      readln (POKRACUJ);
    until POKRACUJ = 0;
  end. { DATUM }
```

Program 6.3

Kromě toho, že procedury CTIDATUM, POCETDNU a VYPISVYSLEDKU komunikují nyní s okolím přes parametry, došlo ještě k jedné změně. V proceduře CTIDATUM bylo trojnásobné opakování obdobného úseku (označme jej úsek 6.1)

```
repeat { dokud není zadána hodnota v intervalu DOLNI..HORNI }
  readln (HODNOTA);   { přečtení celočíselné hodnoty }
  if (HODNOTA < DOLNI) or (HODNOTA > HORNI) then { test }
    writeln ('Zadej cislo v intervalu ',DOLNI:1,' .. ',HORNI:1);
until (HODNOTA > =DOLNI) and (HODNOTA < =HORNI);
```

nahrazeno deklarací procedury CTIHODNOTU s parametry DOLNI a HORNI udávajícími meze povoleného intervalu při čtení a třemi příkazy procedury CTIHODNOTU. Výsledek předává procedura CTIHODNOTU v nelokální proměnné HODNOTA.

Vidíme, že proměnná HODNOTA je z hlediska procedury CTIHODNOTU nelokální, z hlediska procedury CTIDATUM lokální a v hlavním programu ji není možné používat.

6.4 Metoda návrhu shora dolů

V kap. 4 jsme poznali tři typy řídicích struktur

- posloupnost příkazů (složený příkaz),
- větvení (podmíněný příkaz a příkaz case),
- cyklus (cykly for, while a repeat).

Povšimněme si jejich jedné důležité vlastnosti. Všechny řídicí struktury mají tvar jediného příkazu. Tato vlastnost není pro řídicí struktury samozřejmá. Je však velmi výhodná, neboť umožňuje snadno aplikovat metodu návrhu *shora dolů*. Tuto metodu nyní stručně popíšeme.

Mají-li *všechny* řídicí struktury v programu uvedenou vlastnost, lze snadno dosáhnout toho, aby každý logický celek v programu měl také tvar jediného příkazu. Takový celek lze také jednoduše realizovat jako proceduru.

Například v programu DATUM je ze složeného příkazu (úsek 6.1) vytvořena procedura CTIHODNOTU.

Každou deklarovanou proceduru si můžeme představit jako nový příkaz zapisovaný do programu uvedením jeho jména.

Program DATUM pro výpočet dne v týdnu zapsaný s použitím příkazů procedur by měl např. tento tvar:

```
program DATUM (input, output);
var
  DEN      : TYPDEN;          { den v měsíci }
  MESIC    : TYPMESIC;        { měsíc }
  ROK      : TYPROK;          { rok }
  POCET    : NEZAP;           { počet dnů od 1.1.1924 }
  POCETVROCE : DENVROCE;      { pořadové číslo dne v roce }
  POKRACUJ : integer;         { příznak pokračování }
begin { DATUM }
  repeat { dokud má uživatel zájem }
    CTIDATUM (DEN, MESIC, ROK); { čtení vstupních údajů }
    POCETDNU (DEN, MESIC, ROK, POCET,POCETVROCE);
      { výpočet počtu dnů a pořadí }
    VYPISVYSLEDKU (DEN, MESIC, ROK, POCET,
      POCETVROCE);   { výpis výsledků }
    writeln ('Chces pokracovat (0=ne, 1=ano)');
    readln (POKRACUJ);
  until POKRACUJ = 0;
end. { DATUM }
```

Takto zapsaný program nelze sice bez chyby přeložit překladačem Pascalu, popisuje však velmi dobře strukturu algoritmu na vyšší úrovni abstrakce, než poskytuje jazyk Pascal. Je to vlastně program pro počítač, který je schopen realizovat nové příkazy procedur jako standardní příkazy a má nové datové typy jako standardní datové typy.

Při vytváření programu může vzniknout postupně několik úrovní abstrakce, od nejobecnější úrovně vyjádřené např. touto formou:

```
program DATUM (input, output);
begin { DATUM }
  PROVEDALGORITMUS;
end. { DATUM }
```

do úrovně, kde jsou již všechny datové typy i procedury deklarovány (program 6.3 v předchozím čl. 6.3).

Výhodou návrhu shora dolů je, že umožňuje logicky strukturovaný návrh programu směrem od hlavních funkčních bloků postupně až ke konkrétním příkazům. Tvůrce programu není rozptylován ve fázi návrhu globální struktury programu podrobnostmi, na které se může zaměřit při rozpracovávání konkrétních dílčích celků. Jednotlivé úrovně abstrakce tvoří dobrou dokumentaci vývoje programu.

Další výhodou tohoto způsobu návrhu je snadná zaměnitelnost logických celků vzhledem k jejich jednoduché vazbě k okolí (mají tvar jediného příkazu). Nevyhovuje-li činnost nebo algoritmus některého logického celku, lze jej nahradit jiným, provádějícím tutéž činnost lépe. Případné chyby v algoritmu lze tedy snadno odstranit.

Zvýšenou pozornost je naopak nutné věnovat návrhu *vzájemné výměny údajů* procedur mezi sebou. Procedury si mezi sebou předávají údaje, což mohou provádět prostřednictvím parametrů nebo nelokálních proměnných. Zvolený způsob komunikace (včetně např. konkrétních jmen datových typů i globálních proměnných, resp. počtu, datového typu i druhu parametrů) je závazný a nelze jej později změnit bez hlubšího přepracování (často i celého) programu. Při zápisu příkazu procedury do abstraktního programu nemusí být proto vždy určeno, jakými příkazy bude později realizován, ale musí být jednoznačně určen datový typ a způsob příjmu vstupních údajů. Totéž musí být pro daný úsek známo i o údajích výstupních.

Ke každému z uvedených příkazů procedur našeho příkladu lze nyní uvést deklaraci. Dosud nejsou také deklarovány datové typy parametrů a globálních proměnných. V okamžiku deklarace těla procedury, které s parametry provádí operace, musíme datové typy přesně popsat.

K deklaraci procedury CTIDATUM je nutné určit datové typy parametrů TYPDEN, TYPMESIC a TYPROK.

Potom lze deklarovat proceduru CTIDATUM (program 6.3). K deklaraci procedury POCETDNU je třeba dalších typů NEZAP a DENVROCE. Deklarace procedury POCETDNU má tvar z programu 6.3.

K deklaraci procedury VYPISVYSLEDKU není nutné užít žádné další datové typy.

Takto obdržíme program na nízké úrovni abstrakce, ve kterém není deklarována pouze procedura CTIHODNOTU použitá v proceduře CTIDATUM. V dalším kroku snižování abstrakce doplníme deklaraci této procedury. Tím obdržíme konečný tvar programu 6.3.

Systematické rozdělení (strukturalizace) problému na podproblémy s určením jejich vzájemných vazeb není často jednoduché. Kdo tento postup ovládá, ten může být dobrým a tvůrčím programátorem. Zápis algoritmu v Pascalu (nebo jiném programovacím jazyce) je po strukturalizaci jen běžnou rutinou. Programovací jazyk chápeme jen jako vyjadřovací prostředek a jeho zvládnutí je samozřejmostí.

6.5 Funkce

Jak již víme, procedura v pascalském programu pojmenovává část programu a její vyvolání (aktivace) příkazem volání procedury způsobí provedení jisté dílčí akce. Deklarací procedur rozšiřujeme základní množinu příkazů jazyka.

Obdobná situace se nabízí u výrazu. Rádi bychom rozšířili základní množinu operací a standardních funkcí výrazu novými, námi definovanými funkcemi.

Každá standardní operace i funkce počítá ve výrazu hodnotu jistého jednoduchého datového typu. Tuto vlastnost by měly mít i nově definované funkce. Tím se budou lišit od procedur, které jako výsledek hodnotu nevracejí.

Z hlediska předávání parametrů (u funkcí užíváme též názvu argumenty) jsou procedury i funkce totožné.

Pascal umožňuje deklarovat libovolné funkce vracející jako výsledek jednoduchý datový typ (popř., jak později uvidíme, i typ ukazatel). Funkce se deklarují stejně jako procedury, až na dvě odchylky.

První odchylkou je záhlaví funkce, které má tvar

function *jméno* (*formální parametry*): *typ výsledku*;

Od záhlaví procedury se liší vyhrazeným slovem **function**, což není tak podstatné jako to, že za seznamem *formálních parametrů* je zapsán datový *typ výsledku* funkce. Typ výsledku musí být (podobně jako typ formálního parametru) zadán identifikátorem (ne popisem) typu. Nelze proto napsat např. záhlaví

function DEJBARVU (I: *integer*): (ZELENA,BILA,MODRA);

nýbrž

```
type
  BARVY = (ZELENA, BILA, MODRA);
function DEJBARVU (I: integer): BARVY;
```

Druhou odlišností je způsob vytvoření hodnoty výsledku funkce (ten u procedur není třeba). Tělo funkce je realizováno stejně jako tělo procedury, tj. složeným příkazem. Chceme-li v těle funkce dát najevo, že daný výraz tvoří výsledek funkce, zapíšeme do posloupnosti příkazů při deklaraci funkce přiřazovací příkaz

identifikátor := *výraz*

kde na levé straně přiřazovacího příkazu je jméno funkce, jejíž tělo vytváříme. Procedura UMOC z této kapitoly tedy může být přirozeněji zapsána jako funkce s argumenty ZAKLAD a EXPON, vracející hodnotu typu *integer*.

```
function UMOC (ZAKLAD: integer; EXPON: NEZAP): integer;
{ ZAKLAD umocněn na EXPON }
var
   VYSL: integer;  { hodnota výsledku }
   I  : NEZAP;     { řídicí proměnná cyklu }

begin { UMOC }
   VYSL: = 1;
   for I : = 1 to EXPON do
      VYSL: = VYSL * ZAKLAD;
   UMOC: = VYSL;      { přiřazení výsledku funkce }
end; { UMOC }
```

Identifikátoru funkce (s případnými skutečnými parametry) lze užít ve výrazu stejným způsobem jako proměnné téhož typu. Zápis identifikátoru funkce ve výrazu způsobí *aktivaci* těla funkce a výpočet hodnoty funkce. Je-li deklarována funkce UMOC, je možné zapisovat např. výrazy

```
X + UMOC (Y, 2)
UMOC( (UMOC (X,1) + UMOC (X,2)), X * Z)
```

Zápis funkce ve výrazu způsobí tedy její vyvolání a výpočet hodnoty. Z toho také plyne, že tělo funkce UMOC nelze zapsat takto:

```
function UMOC (ZAKLAD: integer; EXPON: NEZAP): integer;
{ ZAKLAD umocněn na EXPON }
var
   I : NEZAP;  { řídicí proměnná cyklu }
begin { UMOC }
   UMOC: = 1;
   for I : = 1 to EXPON do
      UMOC: = UMOC * ZAKLAD;
end; { UMOC }
```

Zápis funkce ve výrazu

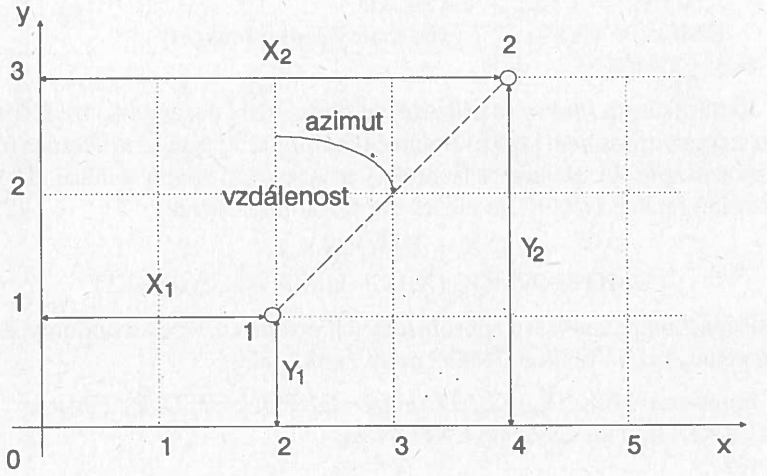
```
UMOC * ZAKLAD
```

znamená totiž z hlediska Pascalu pokus o její novou aktivaci, a to ještě s chybným počtem skutečných parametrů (má mít dva a nemá žádný). Proto pře-

kladač ohlásí chybu: *počet skutečných parametrů neodpovídá počtu formálních parametrů v deklaraci funkce.*

Hodnotu uloženou v těle funkce jako její výsledek nelze již ve výrazu v těle téže funkce využít, nemůže být zároveň i pracovní proměnnou. Je-li hodnota výsledku i dále potřeba, je nutné vždy zavést jinou lokální pomocnou proměnnou (v našem případě VYSL).

Aktivace funkce téhož jména, kterou právě deklarujeme (volání sebe sama), je v Pascalu povolena. Podobně je tomu i u procedur. Takovému volání říkáme *rekurzívní volání* funkce nebo procedury. Algoritmy, využívající rekurze, jsou často velmi elegantní a krátké. Podrobněji se jimi budeme zabývat v čl. 6.6.



Obr. 6.2 Souřadnice bodu

Užití funkcí si ukážeme na příkladu. Umístění bodu na ploše se čtvercovou sítí souřadnic můžeme určit souřadnicemi X a Y vzhledem k počátku souřadnic (obr. 6.2). Pro lepší představu může být uvažovanou plochou např. mapa, kde počátkem souřadnic bude levý dolní roh. Dolní okraj mapy představuje osu x a levý svislý okraj osu y. Souřadnice X bodu je vzdálenost bodu od osy y ve směru osy x. Souřadnice Y bodu je dána vzdáleností bodu od osy x ve směru osy y.

Na obr. 6.2 má bod *1* souřadnice $X_1 = 2$, $Y_1 = 1$ a bod *2* souřadnice $X_2 = 4$, $Y_2 = 3$.

Můžeme také uvažovat měřítko mapy. Zapisuje se obvykle ve tvaru

$$1 : m.$$

Každá vzdálenost z změřená na mapě v měřítku $1 : m$ je ve skutečnosti délkou $z * m$.

Chceme vytvořit program, kterému zadáme souřadnice bodů *1* a *2* na mapě a měřítko mapy. Výsledkem výpočtu bude skutečná vzdálenost bodů *1* a *2* při zadaném měřítku. Dále bude vypočten azimut bodu *2* vzhledem k bodu *1*. Jsme-li tedy např. v terénu bez vhodných orientačních bodů na místě o souřadnicích X_1, Y_1 a známe souřadnice cíle X_2, Y_2, program vypočte vzdálenost cíle a azimut pochodu (azimut počítáme od směru na sever ve smyslu otáčení hodinových ručiček). Výstup azimutu požadujeme ve stupních a minutách (nikoliv v radiánech) od 0 do 359 stupňů 59 minut.

Jak vypočtená vzdálenost, tak azimut budou reálné hodnoty a k jejich výpočtu využijeme funkcí.

Pro výpočet vzdálenosti budeme využívat funkci VZDALENOST, deklarovanou hlavičkou

function VZDALENOST (X, Y: *real*; MERITKO: KLAD): *real*;

která počítá vzdálenost bodu o souřadnicích X a Y od počátku souřadnic v měřítku 1:MERITKO. Budeme-li chtít počítat vzdálenost dvou bodů se souřadnicemi X1, Y1 a X2, Y2, můžeme zadat jako skutečné parametry rozdíly, např.

VZDALENOST (X2–X1, Y2–Y1, MERITKO)

kde MERITKO je proměnná typu KLAD udávající měřítko. Jelikož parametry X a Y funkce VZDALENOST jsou volány hodnotou, můžeme výrazy jako skutečné parametry použít.

Pro výpočet azimutu vytvoříme funkci

function AZIMUT (X, Y: *real*) : *real*;

která bude počítat azimut bodu X, Y vzhledem k počátku souřadnic. Výsledek funkce bude ve stupních, avšak jako jedna reálná hodnota (tedy ne ve tvaru *stupně* : *minuty*). Azimut bodu *2* vzhledem k bodu *1* vypočteme zadáním obdobných skutečných parametrů jako pro funkci VZDALENOST

AZIMUT (X2–X1, Y2–Y1)

Pro tisk azimutu ve tvaru

stupně : *minuty*

vytvoříme proceduru

procedure TISKAZIMUTU (AZIMUT : *real*);

která jako vstup obdrží reálné číslo udávající azimut ve stupních a zobrazí je v požadovaném tvaru. Základní struktura programu bude:

```
program NAVIGACE (input, output);
var
  X1, Y1,              { souřadnice bodu 1 }
  X2, Y2:  real;       { souřadnice bodu 2 }
  MERITKO : KLAD; { měřítko 1:MERITKO }
begin { NAVIGACE }
  writeln ('VYPOCET AZIMUTU A VZDALENOSTI');
  writeln;
  { čtení souřadnic bodů a měřítka }
  writeln ('Zadej souradnice X1, Y1 na jeden radek');
  writeln ('oddelene mezerou :');
  readln (X1,Y1);
  writeln ('Zadej souradnice X2, Y2 na jeden radek');
  writeln ('oddelene mezerou :');
  readln (X2,Y2);
  writeln ('Zadej meritko:');
  readln (MERITKO);
  { výpočet a výpis vzdálenosti }
  writeln ('Vzdalenost: ',VZDALENOST(X2–X1,Y2–Y1,
                MERITKO):8:2);
  { výpočet a výpis azimutu }
  write ('Azimut : ');
  TISKAZIMUTU (AZIMUT(X2–X1,Y2–Y1));
  writeln;
end. { NAVIGACE }
```

Program nejdříve přečte ze vstupu hodnoty souřadnic X1, Y1, X2, Y2 a měřítko MERITKO. Aktivací funkce VZDALENOST získá vzdálenost obou bodů. Jelikož volání funkce reprezentuje přímo hodnotu, je možné výraz

```
VZDALENOST(X2–X1,Y2–Y1,MERITKO)
```

užít přímo jako parametru příkazu *writeln*:

```
writeln ('Vzdalenost: ',VZDALENOST(X2–X1,Y2–Y1,
            MERITKO):8:2);
```

Není tedy nutné užít pomocné proměnné, do které by se nejdříve hodnota přiřadila:

```
I := VZDALENOST(X2–X1,Y2–Y1,MERITKO);
writeln ('Vzdalenost: ', I:8:2);
```

Podobně volání funkce pro výpočet azimutu je přímo zapsáno jako skutečný parametr volaný hodnotou procedury TISKAZIMUTU:

```
TISKAZIMUTU (AZIMUT(X2–X1,Y2–Y1));
```

Nyní můžeme přistoupit k deklaraci funkcí VZDALENOST, AZIMUT a procedury TISKAZIMUTU. Nejdříve deklarujme funkci VZDALENOST. Datový typ KLAD užitý v parametrech můžeme deklarovat takto:

```
type
  KLAD = 1 .. maxint;    { kladné číslo }
```

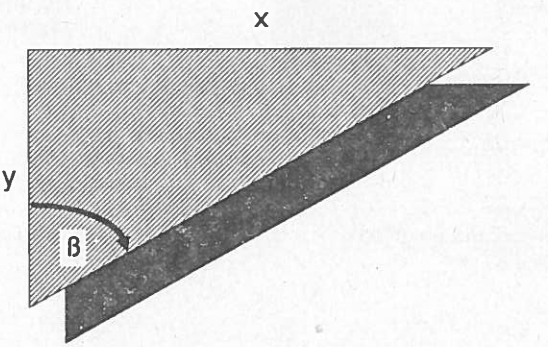
Vlastní algoritmus je výpočtem délky přepony v pravoúhlém trojúhelníku podle Pythagorovy věty

$$\mathrm{DELKA}^2 = \mathrm{X}^2 + \mathrm{Y}^2.$$

V Pascalu má pak výraz pro výpočet délky přepony tvar

```
DELKA := sqrt ( sqr(X) + sqr(Y) ) * MERITKO
```

Délka přepony je vynásobena měřítkem pro získání skutečné vzdálenosti.



Obr. 6.3 Výpočet úhlu

Funkci VZDALENOST budeme tedy deklarovat takto

```
function VZDALENOST (X, Y: real; MERITKO: KLAD): real;
{ výpočet vzdálenosti bodu o souřadnicích X,Y od počátku }
begin { VZDALENOST }
  VZDALENOST := sqrt (sqr(X) + sqr(Y)) * MERITKO;
end; { VZDALENOST }
```

Funkce AZIMUT realizuje výpočet úhlu v pravoúhlém trojúhelníku, známe-li odvěsny (obr. 6.3) podle vztahu

$$\beta = \text{arctg}\,(X/Y).$$

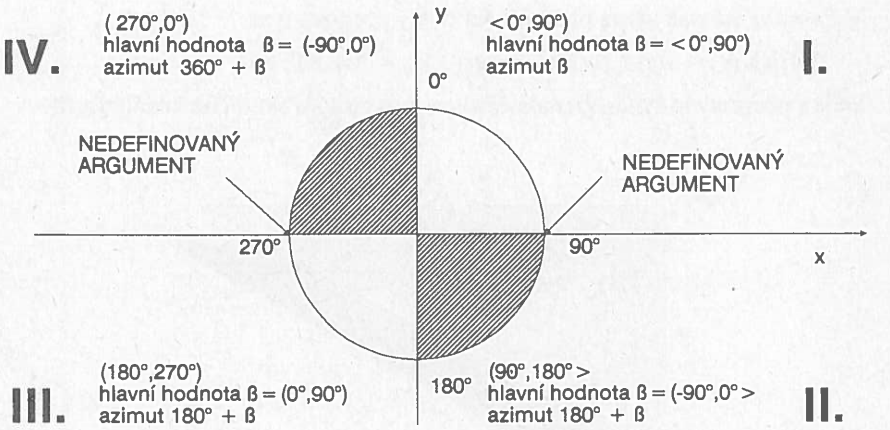
Pascalská funkce *arctan* počítá úhel v radiánech v rozsahu

$$(-\pi/2, \pi/2),$$

což je tzv. hlavní hodnota arctg. Po funkci požadujeme, aby vytvářela výsledek ve stupních v intervalu $<0, 360)$, což vyjádřeno v radiánech je $<0, 2\pi)$. Musíme proto využít mnohoznačnosti funkce arctg, tj.

$$\text{arctg}\,(X/Y) = \beta + n\pi,$$

kde n je libovolné celé číslo a β hlavní hodnota. Navíc pro $Y=0$ není možné argument funkce vyhodnotit (dělení nulou).



Obr. 6.4 Hodnoty funkce arctg

Na obr. 6.4 je vyznačena plocha s osami x a y, které rozdělují plochu do 4 kvadrantů I., II., III. a IV.

Nejdříve se zabývejme případy, kdy $Y = 0$ (argument X/Y má nedefinovanou hodnotu). Pak v případě, že $X > 0$, je výsledkem funkce 90°. Je-li $X < 0$, je výsledkem 270°. Je-li také $X = 0$, nelze úhel vypočítat a výsledkem funkce bude 0°.

Nyní uvažujme jednotlivé kvadranty. V kvadrantu I. je hlavní hodnota *arctg* přímo požadovaným výsledkem. V kvadrantech II. a III. je nutné k výsledku přičíst hodnotu π a v kvadrantu IV. je nutné přičíst 2π. Výsledný úhel vychází v radiánech a je nutné jej převést na stupně. K tomu slouží funkce

function STUPEN (RAD: *real*): *real*;

která pro zadaný argument v radiánech vypočte výsledek ve stupních.

Funkci AZIMUT můžeme deklarovat např. takto:

```
function AZIMUT (X, Y: real): real;
{ výpočet azimutu bodu o souřadnicích X,Y vzhledem
k počátku souřadnic , výsledek je ve stupních ( jako desetinné číslo) }

begin { AZIMUT }
  if Y = 0.0 then
    begin { nespojitost }
      if X > 0.0 then
        AZIMUT := 90.0
      else if X < 0.0 then
        AZIMUT := 270.0
      else { počátek soustavy }
        AZIMUT := 0.0;
    end  { nespojitost }
  else if Y < 0.0 then { pod osou x }
    AZIMUT := STUPEN (PI + arctan (X/Y))
  else if X < 0.0 then { X < 0.0 , Y > 0.0 }
    AZIMUT := STUPEN (2 * PI + arctan (X/Y))
  else { X >= 0.0, Y > 0.0 }
    AZIMUT := STUPEN (arctan (X/Y))
end; { AZIMUT }
```

Zbývá ještě vytvořit funkci STUPEN. Ta využívá při výpočtu vztahu

úhel ve stupních = (úhel v radiánech * 180) / π

Její deklarace bude vypadat takto:

```
function STUPEN (RAD:real): real;
{ převod radiánů na stupně }
begin { STUPEN }
  STUPEN := (RAD * 180) / PI;
end; { STUPEN }
```

Procedura TISKAZIMUTU tiskne hodnotu úhlu zadanou jako reálné číslo ve stupních ve tvaru

stupně : *minuty*

Chceme, aby minuty byly vždy vypsány jako dvě číslice a aby byly v intervalu 00 .. 59. Z reálné hodnoty úhlu AZIMUT se celočíselná hodnota STUPNE, udávající počet stupňů, získá vyhodnocením výrazu

STUPNE := *trunc* (AZIMUT);

Celočíselná hodnota MINUTY, udávající počet minut, se získá vyhodnocením výrazu

MINUTY := *round* (AZIMUT – STUPNE) * 60;

Pokud je vypočtený počet minut roven 60, pak zvýšíme počet stupňů o jeden a počet minut vynulujeme (60 minut = 1 stupeň). Zbývá vyřešit, jak vypsat minuty na dvou znacích tak, aby byly vypsány vždy obě číslice.

Příkaz

write(MINUTY:2)

tento problém neřeší, neboť pro hodnoty MINUTY < 10 vypíše příkaz jako první znak ze dvou tištěných *mezeru* (kap.2). Využijeme proto příkazu

write(MINUTY:1)

který způsobí výpis hodnoty na minimální nutný počet znaků (zde na 1 nebo 2 znaky). V případě, že MINUTY < 10, doplníme zvlášť znak '0' příkazem

```
if MINUTY < 10 then  { úvodní 0 }
   write ('0');
write(MINUTY:1);
```

Nyní můžeme uvést celý program včetně procedury TISKAZIMUTU:

```
program NAVIGACE (input, output);
{ Program vypočte vzdálenost dvou bodů 1, 2 a azimut bodu 2 vzhledem
k bodu 1.

  Vstup: Po výzvě
       Zadej souradnice X, Y na jeden radek ...

    se na jeden řádek zadají souřadnice bodu oddělené mezerou.
    To se provede nejprve pro bod 1, pak pro bod 2. Dále po výzvě
       Zadej meritko:
```

```
      se zadá číslo x určující měřítko mapy 1 : x

   Výstup:
         Vzdalenost: xxxxx.xx
         Azimut : xxx:xx

      Azimut je vypisován ve stupních a minutách, vzdálenost je
      skutečná, přepočtená podle zadaného měřítka
}
const
   PI = 3.1415926536;      { Ludolfovo číslo }

type
   KLAD = 1 .. maxint;     { kladné číslo }

var
   X1, Y1,                 { souřadnice bodu 1 }
   X2, Y2:  real;          { souřadnice bodu 2 }
   MERITKO : KLAD;         { měřítko 1:MERITKO }

function STUPEN (RAD:real): real;
{ převod radiánů na stupně }
begin { STUPEN }
   STUPEN := (RAD * 180) / PI;
end; { STUPEN }

function VZDALENOST (X, Y: real; MERITKO: KLAD): real;
{ výpočet vzdálenosti bodu o souřadnicích X,Y od počátku
souřadnic, přepočtené podle zadaného měřítka }
begin { VZDALENOST }
   VZDALENOST := sqrt (sqr(X) + sqr(Y)) * MERITKO;
end; { VZDALENOST }

function AZIMUT (X, Y: real): real;
{ výpočet azimutu bodu o souřadnicích X,Y vzhledem
k počátku souřadnic , výsledek je ve stupních
( jako desetinné číslo) }
begin { AZIMUT }
   if Y = 0.0 then
```

```
      begin { nespojitost }
        if X > 0.0 then
          AZIMUT := 90.0
        else if X < 0.0 then
          AZIMUT := 270.0
        else { počátek soustavy }
          AZIMUT := 0.0;
      end  { nespojitost }
    else if Y < 0.0 then { pod osou x }
      AZIMUT := STUPEN (PI + arctan (X/Y))
    else if X < 0.0 then { X < 0.0 , Y > 0.0 }
      AZIMUT := STUPEN (2 * PI + arctan (X/Y))
    else { X >= 0.0, Y > 0.0 }
      AZIMUT := STUPEN (arctan (X/Y))
  end; { AZIMUT }

  procedure TISKAZIMUTU (AZIMUT : real);
  { tisk azimutu ve tvaru stupně:minuty }
  var
    STUPNE,
    MINUTY : integer;
  begin { TISKAZIMUTU }
    STUPNE := trunc (AZIMUT); { celá část azimutu = stupně }
    MINUTY := round (AZIMUT - STUPNE) * 60;
    { desetinná část * 60 = minuty }
    if MINUTY = 60 then { nesmí vyjít 60 minut }
      begin { přidání jednoho stupně }
        STUPNE := STUPNE + 1;
        MINUTY := 0;
      end; { přidání jednoho stupně }
    write (STUPNE:5,':');
    if MINUTY < 10 then  { úvodní 0 }
      write ('0');
    write(MINUTY:1);
  end; { TISKAZIMUTU }

  begin { NAVIGACE }
    writeln ('VYPOCET AZIMUTU A VZDALENOSTI');
    writeln;
```

```
    { čtení souřadnic bodů a měřítka }
    writeln ('Zadej souradnice X1, Y1 na jeden radek');
    writeln ('oddelene mezerou :');
    readln (X1,Y1);
    writeln ('Zadej souradnice X2, Y2 na jeden radek');
    writeln ('oddelene mezerou :');
    readln (X2,Y2);
    writeln ('Zadej meritko:');
    readln (MERITKO);
    { výpočet a výpis vzdálenosti }
    writeln ('Vzdalenost: ',VZDALENOST(X2–X1,Y2–Y1,MERITKO):8:2);
    { výpočet a výpis azimutu }
    write ('Azimut  : ');
    TISKAZIMUTU (AZIMUT(X2–X1,Y2–Y1));
    writeln;
end. { NAVIGACE }
```

Program 6.4

6.6 Rekurzívní procedury a funkce

Rekurzívní procedura (funkce) během provádění příkazů svého těla (ještě před jeho ukončením) vyvolá sama sebe. Znovu se tedy začne provádět táž posloupnost příkazů, aniž by původní byla ukončena.

Aktivace sama sebe se může provádět přímo (např. procedura A má ve svém těle opět příkaz procedury A), nebo zprostředkovaně (procedura A volá proceduru B, ta volá C atd., poslední volá opět proceduru A).

Nová aktivace může opět volat sama sebe atd. až do libovolné úrovně.

Má-li rekurzívně aktivovaná procedura nebo funkce lokální proměnné, je při každé aktivaci vytvořena nová sada lokálních proměnných, přičemž při nové aktivaci zůstanou lokální proměnné nadřazených aktivací zachovány. Každá aktivace má tedy své lokální proměnné. K proměnným nadřazených aktivací téže procedury či funkce není přístup možný. Po ukončení aktivace zaniknou odpovídající lokální proměnné a platnými se stanou lokální proměnné nadřazené aktivace.

Existují úkoly, jejichž řešení vede na zápis programů s rekurzívními procedurami, resp. funkcemi. Jsou to problémy, při jejichž rozkladu na podproblémy vznikne problém, který je obdobný původnímu, avšak je v jistém smyslu jednodušší.

Příkladem takového úkolu je výpočet binomického koeficientu $B(n, k)$.

Platí

$$(a+b)^n = B(n,0)\,a^n + B(n,1)\,a^{n-1}b + B(n,2)\,a^{n-2}b^2 + \ldots + B(n,n-1)\,ab^{n-1} + B(n,n)\,b^n.$$

Tedy např. $B(2,0) = 1, B(2,1) = 2, B(2,2) = 1,$

$$(a+b)^2 = a^2 + 2ab + b^2$$

nebo $B(3,0) = 1, B(3,1) = 3, B(3,2) = 3, B(3,3) = 1,$

$$(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3.$$

Pro výpočet hodnoty binomických koeficientů platí vztahy

$$B(n,0) = B(n,n) = 1,$$
$$B(n,k) = B(n-1,k-1) + B(n-1,k).$$

Tedy např. výpočet $B(4,2)$ lze realizovat takto

$$B(4,2) = B(3,1)+B(3,2) = B(2,0)+B(2,1)+B(2,1)+B(2,2) = 1 + B(1,0) + B(1,1) + B(1,0) + B(1,1) + 1 = 1 + 1 + 1 + 1 + 1 + 1 = 6$$

Pro výpočet binomického koeficientu můžeme vytvořit rekurzívní funkci realizující uvedené vztahy:

```
function B (N,K : NEZAP): KLAD;
{ výpočet binomického koeficientu B(n,k) }
begin { B }
  if (K = 0) or (K = N) then
    B:=1
  else
    B := B(N-1, K-1) + B(N-1, K);
end; { B }
```

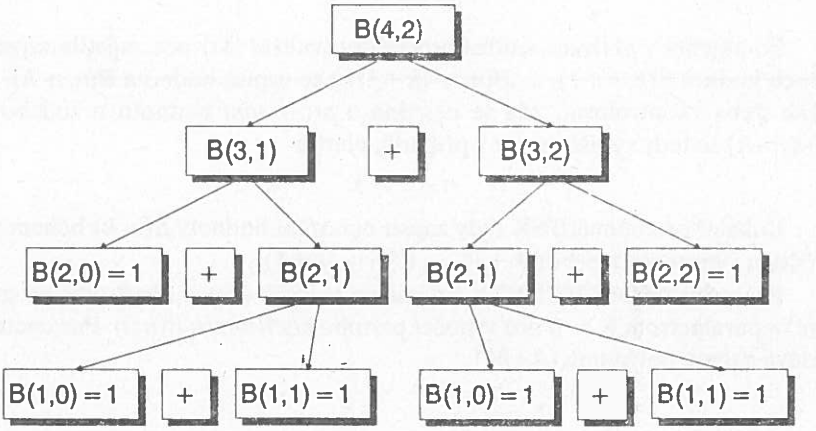
Vyvoláme-li tuto funkci např. s parametry N = 4 a K = 2, dojde k 11 aktivacím funkce B (obr. 6.5). Z toho 6 aktivací B(2,0), B(1,0), B(1,1), B(1,0), B(1,1) a B(2,2) neprovádí rekurzívní volání sebe sama a má hodnotu 1 (jejich součet dává konečný výsledek 6). Zbývajících 5 aktivací funkce B provádí rekurzívní volání sebe sama

B(4,2) = B(3,1) + B(3,2),
B(3,1) = B(2,0) + B(2,1),
B(3,2) = B(2,1) + B(2,2),
B(2,1) = B(1,0) + B(1,1),
B(2,1) = B(1,0) + B(1,1).

Pro binomické koeficienty platí také vztah

$$B(n,k) = B(n,n–k).$$

Tohoto vztahu využijeme při řešení dalšího příkladu. Chceme vypočítat všechny binomické koeficienty pro dané N a vypsat je za sebou na jeden řádek.



Obr. 6.5 Výpočet binomických koeficientů

Abychom nemuseli počítat dvojice $B(n,k)$ a $B(n,n–k)$ se stejnou hodnotou dvakrát využijeme lokálních proměnných rekurzívní procedury.

```
procedure KOEFICIENTY (N,K : NEZAP);
{ tisk koeficientů v řádku }
var
  BNK : KLAD;                    { vypočtené B(n,k) }
begin  { KOEFICIENTY }
  BNK := B(N,K);                 { výpočet a úschova B(n,k) }
  write (BNK : DELKAVYPISU);{ výpis B(n,k) }
  if (N–2 * K) > 1 then { dosud nejsme v polovině }
    KOEFICIENTY (N, K+1);
  if (N–K) > K then              { existuje B(n,n–k) }
    write (BNK : DELKAVYPISU);{ B(n,n–k) = B(n,k) }
end; { KOEFICIENTY }
```

Na začátku procedury je vždy vypočten koeficient $B(n,k)$. Pro další využití je vypočtená hodnota uložena do lokální proměnné BNK. Vypočtená hodnota

$B(n, k)$ je vypsána. Potom je rekurzívně aktivována procedura KOEFICIENTY s parametrem K zvětšeným o jedničku, tj. provede se výpočet $B(n, k+1)$. Jelikož platí $B(n, k) = B(n, n-k)$, je možné počítat pouze část hodnot. Pro n liché počítáme pouze polovinu, pro n sudé ještě o jednu (*prostřední*) hodnotu více. Přesněji řečeno, aktivace se provádí, dokud

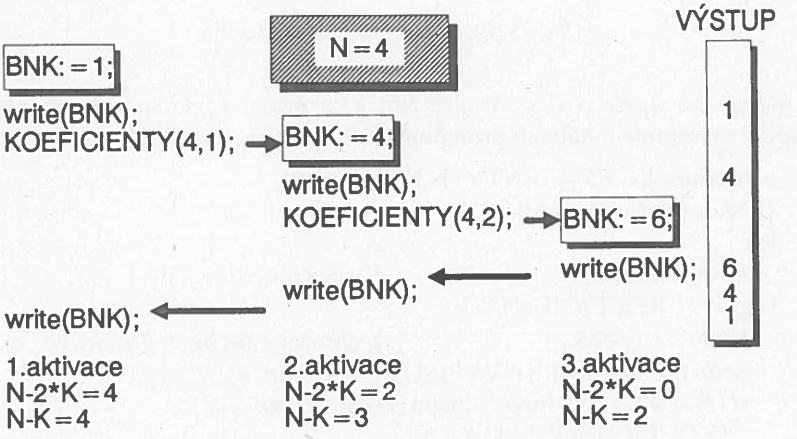
$$n - 2k > 1.$$

Po návratu z aktivace vnitřní procedury (vnitřní aktivace zajistila vypsání všech hodnot $B(n, k+1)$, ... ,$B(n, n-(k+1))$) se vypíše hodnota $B(n, n-k)$. Je však třeba zkontrolovat, zda se nejedná o *prostřední* hodnotu u sudého n. $B(n, n-k)$ se tedy vypíše pouze v případě, platí-li

$$n - k > k.$$

Lokální proměnná BNK tedy zajistí uchování hodnoty $B(n, k)$ během výpočtu a výpisu hodnot $B(n, k+1)$, ..., $B(n, n-(k+1))$.

Proceduru KOEFICIENTY voláme na nejvyšší úrovni (na úrovni programu) s parametrem K = 0 pro výpočet prvního koeficientu $B(n,0)$. Parametr N udává stupeň polynomu $(a+b)^n$.



Obr. 6.6 Aktivace procedury KOEFICIENTY

Například pro N = 4 (obr. 6.6) dojde ke třem aktivacím procedury KOEFICIENTY s hodnotami BNK = 1, 4 a 6. Výstup bude

1 4 6 4 1

Uvedeme nyní celý program pro výpočet binomických koeficientů.

```
program KOEFIC (input, output);
{ Program vypočte všechny binomické koeficienty pro zadané n.

  Vstup: Po výzvě
       Zadej stupen polynomu :

       se zadá celé číslo udávající stupeň polynomu, pro něž počítáme
       koeficienty.

  Výstup:
       B(n,0) B(n,1) B(n,2) .. B(n, n–2) B(n, n–1) B(n, n)

       kde B(n, k) je binomický koeficient
}
const
  DELKAVYPISU = 6;      { délka výpisu jednoho koeficientu }

type
  KLAD = 1 .. maxint;   { kladná čísla }
  NEZAP = 0 .. maxint;  { nezáporná čísla }
var
  N : NEZAP;            { stupeň polynomu }
function B (N,K : NEZAP): KLAD;
{ výpočet binomického koeficientu B(n, k) }
begin { B }
  if (K = 0) or (K = N) then
    B:=1
  else
    B := B(N–1, K–1) + B(N–1, K);
end; { B }

procedure KOEFICIENTY (N,K : NEZAP);
{ tisk koeficientů v řádku }
var
  BNK : KLAD;  { vypočtené B(n, k) }
begin { KOEFICIENTY }
  BNK := B(N,K); { výpočet a úschova B(n, k) }
  write (BNK : DELKAVYPISU); { výpis B(n, k) }
  if (N–2 * K) > 1 then   { dosud nejsme v polovině }
```

```
      KOEFICIENTY (N, K + 1);
    if (N–K) > K then  { existuje B(n, n–k) }
      write (BNK : DELKAVYPISU); { B(n, n–k) = B(n, k) }
end; { KOEFICIENTY }

begin { KOEFIC }
  writeln('BINOMICKE KOEFICIENTY');
  writeln;
  { zadání vstupní hodnoty }
  writeln('Zadej stupen polynomu :');
  readln (N);
  { výpis koeficientů }
  KOEFICIENTY(N,0);
  writeln;
end. { KOEFIC }
```

Program 6.5

Příkladem rekurzívního volání procedury je také konverze celého kladného čísla o základu *z* na posloupnost znaků (které lze např. vypsat na obrazovku). Opačný převod jsme již prováděli v programu KONVERZE. Připomeňme, že pro základ 10 provádí obě tyto konverze automaticky příkaz *read*, *readln*, resp. *write*, *writeln*. Chceme-li ovšem vypsat či přečíst číslo o jiném základu, musíme konverzi naprogramovat.

Jde tedy o to, jak transformovat celočíselnou (budeme uvažovat pouze nezáporné hodnoty) hodnotu *H* pro výpis.

Algoritmus je poměrně jednoduchý. Hodnotu *H* celočíselně dělíme základem. Zbytek po dělení je číselnou hodnotou *poslední* vypisované číslice. Podíl opět dělíme základem a zbytek udává předposlední číslici. To provádíme tak dlouho, dokud výsledek dělení není nulový.

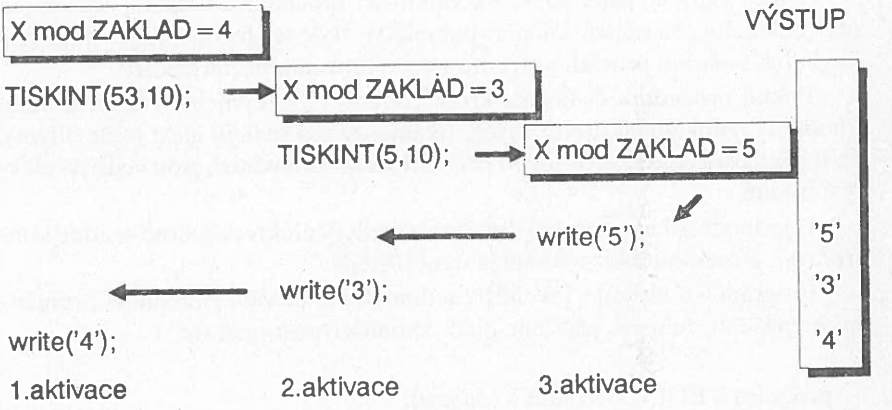
Mějme např. číslo 534. Základem bude pro jednoduchost 10. V prvním kroku je zbytek po dělení 10 roven 4 (tj. číselná hodnota číslice '4'). Potom dělíme 53 se zbytkem 3. Nakonec dělíme 5 a zbytkem je 5.

Problémem převodu je, že hodnoty číslic získáváme v opačném pořadí, než ve kterém je třeba je tisknout. Můžeme však využít parametru X volaného hodnotou, který hodnotu dělence uchová po dobu výpočtu ostatních zbytků. Procedura může mít tvar

```
procedure TISKINT(X     : NEZAP; { tištěná hodnota }
                  ZAKLAD : ZAKLADY);{ základ soustavy }
```

```
{ výpis celočíselné hodnoty po znacích při zadaném základu soustavy }
begin { TISKINT }
  if X > 0 then
    begin
      TISKINT(X div ZAKLAD,ZAKLAD);
      write ( chr(X mod ZAKLAD + ord('0')))
    end;
end; { TISKINT }
```

Parametr X udává současnou hodnotu dělence (při první aktivaci je X rovno převáděné hodnotě). ZAKLAD udává základ soustavy.



Obr. 6.7 Aktivace procedury TISKINT

Pokud X > 0, je nejdříve rekruzívně aktivována procedura TISKINT s parametrem X rovným podílu celočíselného dělení

X **div** ZAKLAD

V okamžiku, kdy parametr X je roven 0, je aktivováno právě tolik procedur TISKINT, kolik bude mít vypisované číslo znaků, a v parametrech X jednotlivých aktivací procedury jsou uloženy dělenci X. Poslední procedura s parametrem X = 0 ukončí rekurzívní rozvoj. Při návratu se nejprve vypíše znak odpovídající zbytku po dělení posledního dělence X

X **mod** ZAKLAD

(tj. první číslice převedeného čísla) atd. Jelikož ukončení činnosti procedur probíhají v opačném pořadí než aktivace, vypíšou se zbytky v opačném pořadí (což právě potřebujeme).

Poznamenejme, že znak (číslice) odpovídající celočíselné hodnotě X **mod** ZAKLAD získáme standardní funkcí

chr (X **mod** ZAKLAD + *ord*('0')),

neboť znaky '0' až '9' jsou uspořádané a jejich ordinální čísla souvislá (kap. 5).

Průběh výpočtu při výpisu hodnotu 534 v desítkové soustavě je uveden na obr. 6.7.

6.7 Vedlejší efekty

V této kapitole jsme popsali komunikaci procedur a funkcí s okolím jak přes nelokální proměnné, tak přes parametry. Bylo též řečeno, že je vhodnější, když styk s okolím provádí procedura nebo funkce přes parametry.

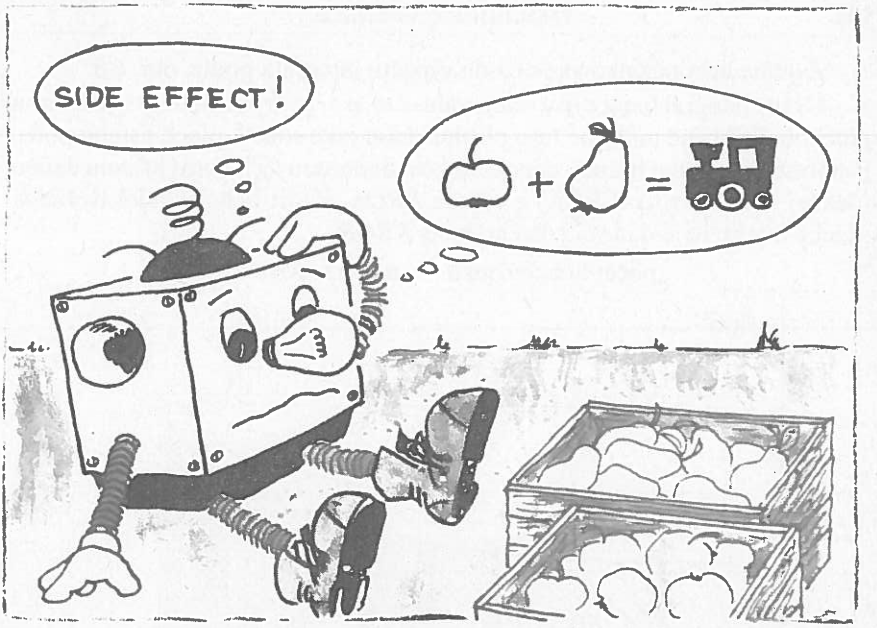
Pokud procedura či funkce kromě hodnoty výstupních parametrů mění i hodnoty nelokálních proměnných, říkáme, že má *vedlejší efekt* (side effect). Je-li předávání údajů prováděno přes nelokální proměnné, jsou vedlejší efekty úmyslné.

U jednodušších programů dokážeme vedlejší efekty poměrně snadno kontrolovat, u rozsáhlejších systémů je to obtížnější.

Program 6.6 ukazuje, jak může nedomyšlené užívání globálních proměnných způsobit, že např. přestane platit komutativnost operace +.

```
program VEDLEJSIEFEKT (output);
{ Program ukazuje, jak vlivem vedlejšího efektu funkce přestává platit
komutativnost operace +.
  Vstupy programu nejsou.
  Výstupy programu jsou dvě různé hodnoty zdánlivě téhož výrazu
}
var
  A: integer;

function D(X: integer): integer;
begin { D }
  D:= A+X;
  A:= A+1; { vedlejší efekt }
end; { D }
```




```
begin { VEDLEJSIEFEKT }
   A:=1; writeln( D(1) + D(A) );
   A:=1; writeln( D(A) + D(1) );
end. { VEDLEJSIEFEKT }
```

Program 6.6

Program 6.6 vypíše dvě hodnoty 6 a 5 pro dva stejné výrazy lišící se pouze pořadím operandů operace sčítání.

Užíváme-li tedy výměnu údajů přes nelokální proměnné, je nutné se podobným obratům vyhnout.

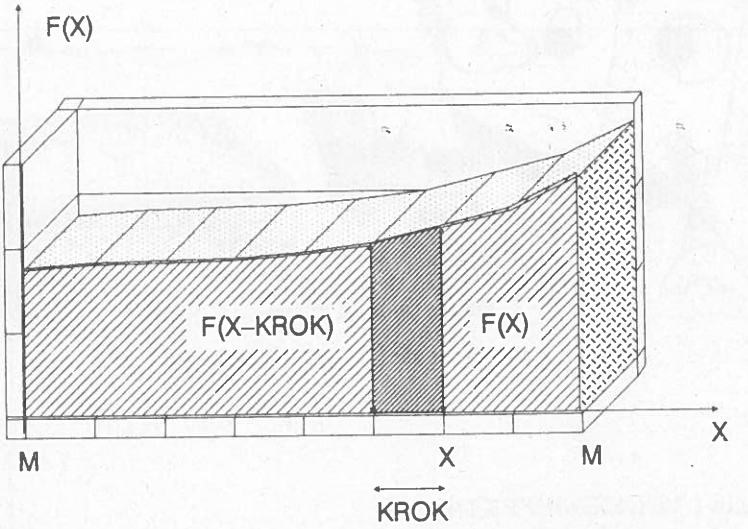
6.8 Funkcionální a procedurální parametry

Uvažujme nyní následující problém. Chceme vypočítat určitý integrál dvou funkcí $f(x)$ a $g(x)$ s reálným argumentem i výsledkem v zadaném intervalu $<m, n>$.

Zvolme lichoběžníkovou metodu výpočtu integrálu podle obr. 6.8.

Určitý integrál funkce $f(x)$ v intervalu $<m, n>$ je určen slabě vyšrafovanou plochou. Přibližně můžeme tuto plochu získat jako součet ploch jistého počtu lichoběžníků s dolní hranou danou funkční hodnotou $f(x)$, horní hranou danou funkční hodnotou $f(x–KROK)$ a výškou *KROK*. Počet lichoběžníků tvořících plochu integrálu je dán velikostí hodnoty *KROK*

$$\text{počet lichoběžníků} = (n - m) / KROK.$$



Obr. 6.8 Lichoběžníková metoda výpočtu určitého integrálu

Čím menší je hodnota *KROK*, tím více lichoběžníků vyplňuje plochu a tím více se také součet jejich ploch blíží skutečné velikosti vyšrafované plochy.

Pro náš program zvolme funkci $f(x) = \exp(x)$ a funkci $g(x) = 1/x$. Je možné zvolit i jiné funkce, na principu metody se nic nezmění.

Vytvořený program vyhodnocuje velikost plochy součtem ploch lichoběžníků (podle úmluvy budeme funkce f a g v programu označovat velkými písmeny; jsou to nestandardní identifikátory):

$$(F(X–KROK) + F(X)) / 2 * KROK.$$

Součet provádíme v cyklu while. Hodnota KROK je konstantní. Funkční hodnota X je na počátku nastavena na hodnotu M + KROK. Tím se zajistí, že první vypočtená plocha lichoběžníka bude

$$(F(M) + F(M+KROK)) / 2 * KROK$$

Po výpočtu plochy jednoho lichoběžníka se hodnota proměnné X zvýší o KROK. Cyklus je ukončen, dosáhne-li hodnota proměnné X horní hranici intervalu N. Výsledná velikost plochy je zobrazena na obrazovce.

V každém kroku výpočtu je vždy F(X–KROK) rovno hodnotě F(X) v předchozím kroku. Proto je zbytečné počítat v každém kroku obě hodnoty. Hodnotu F(X) vždy uchováme v proměnné MINULA a využijeme ji jako F(X–KROK) v dalším kroku. Tak zajistíme, že funkce F se vyhodnocuje při každém průchodu tělem cyklu pouze jednou. Totéž se provádí i pro funkci G.

Program řešící tento problém je označen 6.7.

```
program INTEGRAL(input, output);
{ Program provádí výpočet určitého integrálů lichoběžníkovou metodou
pro funkce f(x) = exp(x) a g(x) = 1 / x v zadaném intervalu <m, n>.

Vstup programu:
      Po výzvě
   Zadej interval

   se zapíší dvě reálné hodnoty na jeden řádek. Udávají meze intervalu
   a jsou odděleny alespoň jednou  mezerou. První hodnota musí být
   menší než druhá.

Výstup programu:
   INTEGRAL f(x) = yyyyyy.yy
   INTEGRAL g(x) = zzzzzz.zz
}
const
   KROK     = 0.01;        { krok metody }
var
   SOUCET       : real;     { součet ploch lichoběžníků }
   X            : real;     { argument }
   M,N          : real;     { meze intervalu }
   HODNOTA      : real;     { vypočtená funkční hodnota }
   MINULA       : real;     { předchozí funkční hodnota }

function F(X:real):real; { první funkce }
begin { F }
   F:=exp(X);
end; { F }
```

```
function G(X:real):real; { druhá funkce }
begin { G }
  G:=1/X;
end; { G }

begin { INTEGRAL }
  writeln('VYPOCET INTEGRALU');
  writeln;
  writeln('Zadej interval');
  readln(M,N);
  { výpočet integrálu funkce F }
  SOUCET:=0;          { nastavení počátečních hodnot }
  MINULA:= F(M);
  X:= M + KROK;   { dolní mez intervalu }
  while X <= N do
    begin          { cyklus pro součet obsahů lichoběžníků }
      HODNOTA:= F(X);
      SOUCET:=SOUCET + (MINULA + HODNOTA)/2 * KROK;
      MINULA:= HODNOTA;
      X:= X + KROK;
    end;        { cyklus }
  writeln('INTEGRAL f(x)= ',SOUCET:8:2);
  { vypočet integrálu funkce G }
  SOUCET:=0;          { nastavení počátečních hodnot }
  MINULA:= G(M);
  X:= M + KROK;   { dolní mez intervalu }
  while X <= N do
    begin          { cyklus pro součet obsahů lichoběžníků }
      HODNOTA:= G(X);
      SOUCET:=SOUCET+(MINULA + HODNOTA)/2 * KROK;
      MINULA:= HODNOTA;
      X:= X + KROK;
    end;        { cyklus }
  writeln('INTEGRAL g(x)= ',SOUCET:8:2);
end. { INTEGRAL }
```

Program 6.7

V těle programu 6.7 se dvakrát opakují algoritmicky shodné části pro součet ploch lichoběžníků, které se liší pouze v identifikátoru funkce F a G.

Bylo by zřejmě výhodné vytvořit pro výpočet určitého integrálu proceduru, jejímž parametrem by nebyla hodnota výrazu nebo proměnná (jak jsme si zvykli), ale funkce. Obdobných problémů, kdy je třeba parametrizovat funkci nebo proceduru jinou funkcí nebo procedurou, je více. Proto Pascal umožňuje mimo parametry volané hodnotou a odkazem deklarovat i tzv. funkcionální a procedurální parametry.

Do části formálních parametrů funkce nebo procedury je možné uvést formální parametr nového typu, který má tvar hlavičky jiné funkce nebo procedury. S identifikátorem procedury (funkce) uvedené v parametrech je dovoleno v těle deklarované procedury (funkce) pracovat jako se skutečnou procedurou (funkcí).

Je-li funkce předávána jako parametr, nazýváme ji *funkcionálním parametrem*, předávání procedury *procedurálním parametrem*.

Vazba na konkrétní funkci nebo proceduru se provede při vyvolání dosazením skutečného identifikátoru funkce nebo procedury jako skutečného parametru. Je ovšem nutné, aby formálním i skutečným parametrem byla procedura nebo funkce se stejným počtem a datovým typem svých parametrů. Skutečné a formální parametry si musí odpovídat i co do druhu (volání hodnotou, odkazem, popř. opět funkcionální a procedurální parametry).

Pro náš program INTEGRAL by bylo zřejmě vhodné vytvořit funkci SOUCET pro výpočet určitého integrálu a parametrizovat ji jednak hranicemi intervalu (parametry volané hodnotou), jednak funkcí, pro niž integrál počítáme (funkcionální parametr).

Deklarace by měla tvar:

```
function SOUCET (M,N: real;function F(X:real):real):real;
{ výpočet určitého integrálu funkce F lichoběžníkovou metodou
v intervalu <M,N> }
var
  S         : real;  { součet ploch lichoběžníků }
  X         : real;  { argument }
  HODNOTA   : real;  { vypočtená funkční hodnota }
  MINULA    : real;  { předchozí funkční hodnota }
begin { SOUCET }
  S:= 0.0;           { nastavení počátečních hodnot }
  MINULA := F(M);
  X := M + KROK; { dolní mez intervalu }
  while X <= N do
    begin { cyklus pro součet obsahů lichoběžníků }
```

```
        HODNOTA := F(X);
        S := S + (MINULA + HODNOTA)/2 * KROK;
        MINULA := HODNOTA;
        X := X + KROK;
      end; { cyklus }
    SOUCET := S;
end; { SOUCET }
```

V záhlaví je jako funkcionální formální parametr uvedena celá hlavička funkce F

function F(X: *real*): *real*;

a v těle funkce SOUCET se s identifikátorem F pracuje jako se skutečnou funkcí. Při aktivaci funkce SOUCET zápisem

SOUCET(M,N,G)

se dosadí jméno skutečné funkce (zde je to jméno G) a s funkcí tohoto jména se skutečně pracuje.

Celý program INTEGRAL po úpravě je uveden jako program 6.8. Na rozdíl od všech ostatních programů není možné jej přeložit překladačem Turbo Pascalu, neboť tento překladač nedovoluje používat funkcionální a procedurální parametry.

```
program INTEGRAL(input, output);
{ Úvodní komentáře jsou totožné s programem 6.7 }
const
  KROK    = 0.01;        { krok metody }
var
  M,N     : real;        { meze intervalu }

function F(X: real): real; { první funkce }
begin { F }
  F:=exp(X);
end; { F }

function G(X: real): real; { druhá funkce }
begin { G }
  G:=1/X;
end; { G }
```

```
function SOUCET (M,N: real; function F(X: real): real): real;
{ výpočet určitého integrálu funkce F lichoběžníkovou metodou
v intervalu M,N }
var
  S          : real;  { součet ploch lichoběžníků }
  X          : real;  { argument }
  HODNOTA    : real;  { vypočtená funkční hodnota }
  MINULA     : real;  { předchozí funkční hodnota }

begin { SOUCET }
  S:= 0.0;            { nastavení počátečních hodnot }
  MINULA := F(M);
  X := M + KROK;  { dolní mez intervalu }
  while X < = N do
    begin { cyklus pro součet obsahů lichoběžníků }
      HODNOTA := F(X);
      S := S + (MINULA + HODNOTA)/2 * KROK;
      MINULA := HODNOTA;
      X := X + KROK;
    end; { cyklus }
  SOUCET := S;
end; { SOUCET }

begin { INTEGRAL }
  writeln('VYPOCET INTEGRALU');
  writeln;
  writeln('Zadej interval');
  readln(M,N);
  writeln('INTEGRAL f(x) = ',SOUCET(M,N,F):8:2);
  writeln('INTEGRAL g(x) = ',SOUCET(M,N,G):8:2);
end. { INTEGRAL }
```

Program 6.8

Jako skutečné funkcionální a procedurální parametry nemohou být použity standardní funkce a procedury. Proto např. v programu 6.8 je standardní funkce *exp obalena* funkcí F, kterou je již možné použít jako skutečný funkcionální parametr.

U některých překladačů Pascalu se používání funkcionálních a procedurálních parametrů může lišit od popsané standardní verze, popř. může tento

typ parametrů chybět. Je proto dobré se v uživatelské příručce přesvědčit o vlastnostech užívaného překladače.

6.9 Direktivy

Závěrem kapitoly o procedurách a funkcích uveďme ještě jeden fakt.

Jak víme, je v Pascalu nutné každý objekt (konstantu, datový typ, proměnnou proceduru i funkci) deklarovat *dříve*, než je objekt použit. Při deklaraci vzájemně rekurzívních procedur a funkcí může nastat situace, kdy tento požadavek nelze splnit. Mějme např. dvě procedury P1 a P2:

```
procedure P1(I: integer);
begin { P1 }
  :
  P2 (2.83);
  :
end; { P1 }
```

```
procedure P2(R: real);
begin { P2 }
  :
  P1 (283);
  :
end; { P2 }
```

Zapíšeme-li v deklaracích jako první proceduru P1, nemůžeme užít v jejím těle procedury P2. Naopak, zapíšeme-li jako první v deklaraci P2, nemůžeme v jejím těle užít P1.

Aby bylo možno deklarovat i takto zapsané procedury, umožňuje Pascal *oddělit hlavičku* procedury i funkce *od jejího těla*. Je možné zapsat nejdříve hlavičku, která namísto těla má uveden identifikátor *forward* (forward = vpřed, vpředu). Tím se považuje procedura či funkce za deklarovanou a je možné ji používat. V libovolném následujícím místě téhož bloku, kde je povoleno deklarovat procedury či funkce, je pak možno doplnit tělo. To je pak uvedeno jen se zkrácenou hlavičkou, která již neobsahuje formální parametry a u funkcí ani typ výsledku (pouze jméno).

Identifikátoru uvedenému místo těla funkce či procedury říkáme *direktiva*. Pro náš příklad bychom tedy mohli napsat

```
procedure P2 (R: real);
  forward;
procedure P1 (I: integer);
begin { P1 }
  :
  P2(2.83);
  :
end; { P1 }
```

```
procedure P2;   { zkrácená hlavička bez parametrů a typu výsledku }
begin { P2 }
  :
  P1 (283);
  :
end; { P2 }
```

U složitějších problémů, kde vzájemná vazba při volání procedur a funkcí není jednoduchá, je možné a také vhodné deklarovat nejprve *všechny* procedury a funkce s direktivou *forward* a teprve potom doplnit deklarace těl. Oboje lze provést ve zcela libovolném a pro nás logickém pořadí.

Direktiva *forward* je jedinou *standardní* direktivou Pascalu. Bývá užívána i další direktiva *external* určující, že tělo procedury nebo funkce se nachází v jiném, zvlášť přeloženém programovém modulu. Standardní Pascal neřeší problém odděleného překladu těl procedur a funkcí. Pro výstavbu rozsáhlejších programových systémů je to však zcela nezbytné. Proto bývá tento problém u většiny překladačů řešen, bohužel však různě. Nejčastěji se využívá direktivy *external*.

Podrobnější popis odděleného překladu programových modulů přesahuje rámec této knihy. Pokud by čtenář chtěl pracovat s Pascalem tímto způsobem, je nutné, aby získal informace v uživatelské příručce konkrétního překladače.

6.10 Cvičení

1. Deklarujte funkci pro výpočet faktoriálu $n!$ podle tzv. Stirlingovy aproximace ($\pi = 3.14159$, $e = 2.71828$)

$$n! = \sqrt{[2\pi n]}\ (n/e)^n.$$

2. Deklarujte funkci pro výpočet třetí odmocniny z a podle vztahu

$$x_{n+1} = 1/3\ (2x_n + a/x_n^2).$$

Jde o iterační vztah. Vychází se z jisté zvolené hodnoty x. Podle udaného vztahu se počítají hodnoty x_{n+1} tak dlouho, dokud se dvě vypočtené hodnoty liší o hodnotu menší, než je zadaná přesnost (viz program ENAXTOU).

3. Deklarujte funkce pro výpočet sinh, cosh a tgh podle vztahů

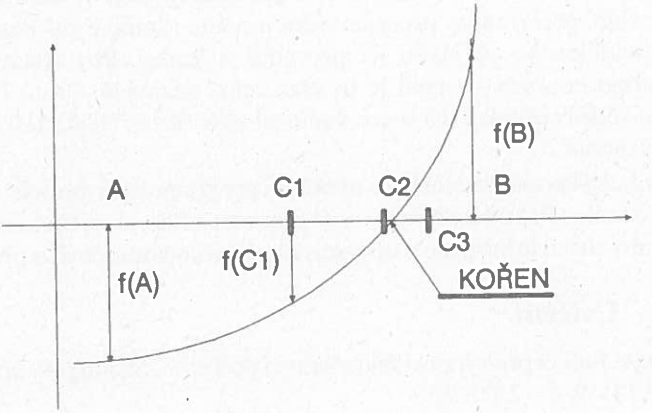
$$\begin{aligned} \sinh(x) &= (\exp(x) - \exp(-x)) / 2, \\ \cosh(x) &= (\exp(x) + \exp(-x)) / 2, \\ \mathrm{tgh}\,(x) &= \sinh(x) / \cosh(x). \end{aligned}$$

4. Deklarujte funkci RAND s jedním celočíselným parametrem a celočíselným výsledkem, která počítá vždy další člen posloupnosti pseudonáhodných čísel podle vztahu

$$x_{i+1} = (1001 * x_i + 333) \textbf{ mod } 1024.$$

Vytvořte program, který vypíše vždy prvních 100 členů posloupnosti pro různě zadaná x_0 (x_0 je volitelné libovolně).

5. Deklarujte logickou funkci JECISLICE, která pro zadaný znak zjistí, zda se jedná o číslici '0' .. '9'.
6. Vytvořte funkci, která nalezne kořen rovnice $f(x) = 0$ metodou půlení intervalu. Jsou zadány meze intervalu, v němž kořen má ležet, přesnost ε a funkce $f(x)$ jako funkcionální parametr.



Obr. 6.9 Metoda půlení intervalu

Návod (obr. 6.9):
Předpokládá se, že funkční hodnoty v mezích intervalu A a B mají opačná znaménka (alespoň jeden kořen v intervalu existuje). Vždy rozpůlíme interval $<A,B>$, čímž obdržíme hodnotu

$$C = (A + B) / 2.$$

Získáme funkční hodnotu $f(C)$ v dělicím bodě. Kořen rovnice leží v té polovině intervalu, v níž mají funkční hodnoty v krajních bodech opačné znaménko. Zúžíme interval na vybranou polovinu a půlení opakujeme. Cyklus ukončíme, je-li rozdíl mezí intervalu (v absolutní hodnotě) menší než ε.

7. Vytvořte program, který vypočte, kolik dnů uběhlo v rozmezí dvou zadaných dat.

7 POLE

V předchozích kapitolách jsme diskutovali základní datové typy a řídicí struktury jazyka Pascal. Nyní přejdeme ke složitějším datovým typům vytvářeným uživatelem. Oblastí pro aplikaci Pascalu bude měření a vyhodnocování výsledků měření.

7.1 Tabulka měření

Měříme-li libovolnou fyzikální veličinu, obvykle neprovádíme měření pouze jednou. Opakujeme je několikrát, abychom vyloučili nahodilou chybu. Naměřené hodnoty zapisujeme do tabulky, která mívá tvar:

1	x_1
2	x_2
3	x_3
4	x_4
.	.
.	.
n	x_n

Uvedená tabulka uchovává hodnoty $x_1, x_2, \ldots, x_n$ získané při n měřeních veličiny x. Hodnoty $x_1, x_2, \ldots, x_n$ můžeme zpracovat některým z klasických způsobů statistického vyhodnocování výsledků (viz např. [24]). *Aritmetický průměr* X je součtem hodnot x_i dělený jejich počtem.

Směrodatnou odchylku aritmetického průměru Y pro $n >= 10$ vypočteme podle vztahu

$$Y = \sqrt{[(y_1^2 + y_2^2 + \ldots + y_n^2)/(n * (n-1))]},$$
$$y_i = x_i - X.$$

Uvedený výpočet je znám mnoha pracovníkům, kteří jej často opakovaně provádějí. Pokusme se jej proto algoritmizovat do formy pascalského programu.

Program pro vyhodnocování n naměřených hodnot může mít strukturu:

```
program MERENI (input, output);
var
  N                    : TYPPOCTU; { skutečný počet hodnot }
```

```
    PRUMER,                          { aritmetický průměr }
    SMODCHYLKA : real;               { směrodatná odchylka }
    NAMHODNOTY,                      { tabulka naměřených hodnot }
    ODCHYLKY   : TABULKA;            { tabulka odchylek od průměru }

begin { MERENI }
  { vstup naměřených hodnot }
  VSTUPHODNOT (N, NAMHODNOTY);
  { výpočet aritmetického průměru }
  PRUMER := VYPOCETPRUMERU(N,NAMHODNOTY);
  { odchylky měření }
  ODCHMERENI (N, PRUMER, NAMHODNOTY, ODCHYLKY);
  { výpočet směrodatné odchylky }
  SMODCHYLKA := VYSMODCHYLKY (N, ODCHYLKY);
  { výstup průměru a směrodatné odchylky }
  VYSTUP (PRUMER, SMODCHYLKA);
end. { MERENI }
```

Procedura VSTUPHODNOT provádí čtení naměřených hodnot, které uloží do proměnné NAMHODNOTY. Jejich skutečný počet bude uložen do proměnné N.

Funkce VYPOCETPRUMERU vypočte průměr naměřených hodnot a uloží jej do proměnné PRUMER. Na základě těchto hodnot provede procedura ODCHMERENI výpočet odchylek jednotlivých měření od aritmetického průměru. Odchylky budou uloženy do proměnné ODCHYLKY.

Funkce VYSMODCHYLKY provede výpočet podle uvedeného vztahu a výsledek uloží do proměnné SMODCHYLKA. Poslední část programu procedura VYSTUP provede výstup vypočtených hodnot.

Datový typ TABULKA (proměnných NAMHODNOTY a ODCHYLKY) bude reprezentovat hodnoty uvedené v tabulce. Každá hodnota datového typu TABULKA je složena z maximálně 100 reálných hodnot označených celými čísly od 1 do 100.

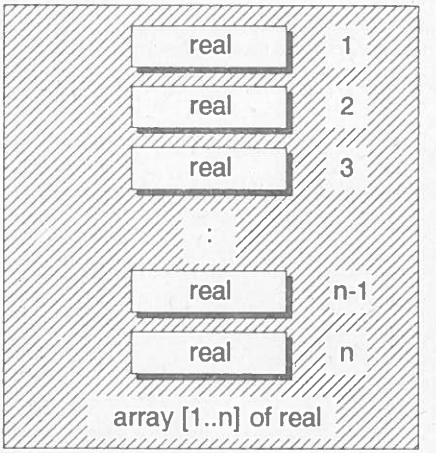
Datové typy, jejichž hodnoty jsou složeny z hodnot obecně jiných datových typů, budeme nazývat *strukturované datové typy* nebo zkráceně *datové struktury*. Existuje více druhů strukturovaných datových typů. Postupně se s nimi seznámíme.

První z nich je (vhodným pro reprezentaci naší tabulky), strukturovaný datový type *pole* (obr. 7.1).

Vidíme, že datový typ pole se skládá z uspořádané množiny hodnot jistého

datového typu (v našem případě typu *real*). Každé takové hodnotě říkáme *složka* pole. Všechny složky pole musí být *stejného* datového typu a jejich počet musí být předem znám.

Abychom mohli s jednotlivými složkami pole pracovat, musíme si je označit. K tomu využijeme hodnot vhodného *ordinálního* typu (v našem případě celočíselného intervalu 1..100). Vzpomeneme-li si na vlastnosti ordinálních



Obr. 7.1 Pole

typů (kap. 5), víme, že množina hodnot ordinálního typu je také (stejně jako množina složek pole) uspořádaná a konečná. Tím, že každé složce pole přiřadíme hodnotu vhodného ordinálního typu, kterému budeme říkat typ *indexu*, budou složky pole jednoznačně označeny.

Počet složek pole je tedy určen množinou hodnot ordinálního typu, přičemž každé hodnotě ordinálního typu je přiřazena hodnota jiného, libovolného typu. Udáním hodnoty ordinálního typu můžeme obdržet přiřazenou hodnotu (složku pole).

Formálněji řečeno, pole je zobrazením množiny hodnot ordinálního typu na množinu hodnot libovolného typu (obr. 7.2).

Složky pole jsou jednoznačně označeny hodnotami indexu. Hodnota 1 označuje (v našem případě) první složku pole, hodnota 10 desátou ap.

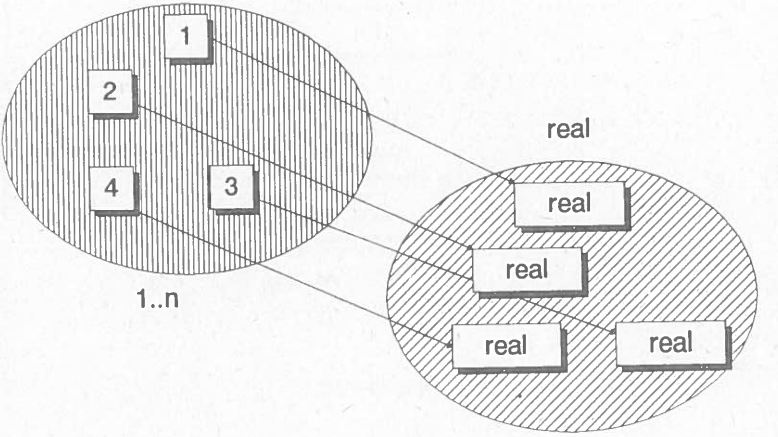
K tomu, abychom mohli pole využít, je nutné popsat v programu jeho datový typ, popř. deklarovat jeho proměnné. Popis typu provedeme zápisem

array [*ordinální typ*] **of** *libovolný typ*

V našem případě můžeme pak deklarovat datový typ tabulka takto

```
type
  TABULKA = array [1..100] of real
```

Popis typu má dvě části. V lomených závorkách je uveden ordinální typ indexu, který určuje uspořádání i počet hodnot (a tím i počet složek pole). Za vyhrazeným slovem **of** je uveden libovolný datový typ složek pole.



Obr. 7.2 Pole jako zobrazení

Typ indexu může být libovolný ordinální typ. Vždy je však třeba mít na paměti, že pro každou hodnotu indexu bude v datovém typu pole existovat jedna složka pole, která pak zaujímá určité místo v paměti počítače. A ta není neomezeně velká. Proto např. pascalsky správný zápis

array [*integer*] **of** *real*

vyhradí při deklaraci proměnné tohoto typu (při hodnotě *maxint* = 32767 a velikosti složky typu *real* 8 slabik) přibližně 524 tisíc slabik. To je požadavek často na hranici splnitelnosti.

Z těchto důvodů se jako typy indexů uplatní nejčastěji intervaly typu *integer* nebo výčtové typy.

Počet složek pole je v Pascalu *pevný*; je určen vlastnostmi datového typu indexu, a tudíž musí být znám již v době zápisu programu. To je v mnoha aplikacích nevýhodné, neboť při proměnném počtu složek pole v jednotlivých běžích programu musíme pole deklarovat tak, aby vyhovovalo největšímu možnému požadavku na počet složek. V našem případě deklarujeme také typ TABULKA jako pole o 100 složkách, ačkoliv 100 hodnot zadáme programu jen ojediněle. Při neuváženém užívání rozměrných polí můžeme zaplnit paměť převážně nevyužitým prostorem.

Podobně jako u popisu předchozích uživatelských datových typů není nutné typ pole pojmenovat a je možné jej zapisovat přímo v místě deklarace proměnných. Například lze napsat

```
var
  NAMHODNOTY,
  ODCHYLKY : array [1..100] of real;
```

Pojmenování typu identifikátorem je však výhodnější, neboť s pojmenovaným typem lze pracovat při návrhu algoritmu metodou shora dolů bez detailní znalosti jeho vnitřní struktury.

S polem jako celkem lze provést pouze operaci *přiřazení*. Chceme-li použít jiné operace s hodnotou pole, je nutné pracovat s jednotlivými složkami pole odděleně. Řekli jsme již, že každá hodnota pole je jednoznačně označena odpovídající hodnotou indexu. Toho využijeme v odkazu na konkrétní složku pole, který má tvar

proměnná typu pole [*výraz typu indexu*]

Tedy např. odkaz na 5. složku pole ODCHYLKY bude mít tvar

ODCHYLKY [5]

Důležité je, že výraz v lomených závorkách nemusí být konstantní. Hodnotu indexu lze spočítat až v průběhu činnosti programu, např. v zápisu

ODCHYLKY [I–1]

Zpracování polí je úzce spjato s využitím cyklu for, neboť jej lze s výhodou využít při opakované činnosti s několika složkami pole. Například součet prvních N hodnot pole NAMHODNOTY do proměnné SOUCET snadno popíšeme cyklem:

```
SOUCET := 0;
for I := 1 to N do
  SOUCET := SOUCET + NAMHODNOTY [I];
```

V našem programu MERENI tedy budeme deklarovat typ TABULKA pro uchování maximálně 100 hodnot takto:

```
const
  MAXN = 100;
type
  TABULKA = array [1..MAXN] of real;
```

Funkci VYPOCETPRUMERU pro výpočet aritmetického průměru složek pole naměřených hodnot NAMHODNOTY typu TABULKA s indexy od 1 do N (N je skutečný počet hodnot v poli) budeme realizovat cyklem for s využitím celočíselné řídicí proměnné I.

```
function VYPOCETPRUMERU (N:TYPPOCTU;     { počet hodnot }
                         var NAMHODNOTY: TABULKA) { hodnoty }
                         : real;
{ výpočet aritmetického průměru }
var
  SOUCET : real;                { součet hodnot }
  I: TYPPOCTU;                  { řídicí proměnná cyklu }

begin { VYPOCETPRUMERU }
  SOUCET := 0.0;
  for I := 1 to N do
    SOUCET := SOUCET + NAMHODNOTY[I];
  VYPOCETPRUMERU := SOUCET / N;
end; { VYPOCETPRUMERU }
```

Známe-li aritmetický průměr (proměnná PRUMER), můžeme vypočítat odchylky jednotlivých měření od aritmetického průměru. V proceduře ODCHMERENI využijeme opět cyklu for :

```
for I := 1 to N do
  ODCHYLKY [I] := NAMHODNOTY [I] – PRUMER;
```

Směrodatnou odchylku vypočteme ve funkci VYSMODCHYLKY rovněž cyklem

```
SOUCET := 0.0;
for I := 1 to N do
  SOUCET := SOUCET + sqr(ODCHYLKY[I]);
VYSMODCHYLKY := sqrt(SOUCET/(N * (N–1)));
```

Při čtení vstupních hodnot v proceduře VSTUPHODNOT je nejprve

získán celkový počet hodnot N a potom jsou cyklem for čteny jednotlivé naměřené hodnoty.

```
readln (N);   { počet naměřených hodnot }
writeln ('Zadej ',N:1,' namerenych hodnot,');
writeln ('kazdou na zvlastni radek.');
for I := 1 to N do { čtení hodnot }
  readln (NAMHODNOTY[I]); { čtení i-té hodnoty }
```

Každá hodnota je zadávána na zvláštním řádku vstupního textu a je ukládána do složky pole NAMHODNOTY [I].

Výsledný program MERENI je označen 7.1.

```
program MERENI (input, output);
{ Vyhodnocení výsledků n měření; výpočet aritmetického průměru
a směrodatné odchylky

Vstup : Po výzvě
        Zadej pocet namerenych hodnot
      zadej skutečný počet hodnot < = 100
      Po výzvě
        Zadej n namerenych hodnot ....
      zadávej naměřené hodnoty každou na zvláštní řádek

Výstup :
        arit. pruměr +/- směrodatná odchylka
}
const
  MAXN = 100;    { maximální počet naměřených hodnot }

type
  TYPPOCTU = 1 .. MAXN; { typ počtu hodnot }
  TABULKA = array [TYPPOCTU] of real; { tabulka hodnot }

var
  N              : TYPPOCTU; { skutečný počet zadaných hodnot }
  PRUMER,                    { aritmetický průměr }
  SMODCHYLKA     : real;     { směrodatná odchylka }
  NAMHODNOTY,                { tabulka naměřených hodnot }
  ODCHYLKY       : TABULKA;  { tabulka odchylek od průměru }
```

```
procedure VSTUPHODNOT (var N:TYPPOCTU;     { počet hodnot }
                       var NAMHODNOTY:TABULKA); { hodnoty }
{ přečtení vstupních hodnot }
var
  I: TYPPOCTU;     { řídicí proměnná cyklu }
begin { VSTUPHODNOT }
  writeln ('ZPRACOVANI VYSLEDKU MERENI');
  writeln;
  writeln ('Zadej pocet namerenych hodnot');
  readln (N);        { počet naměřených hodnot }
  writeln ('Zadej ',N:1,' namerenych hodnot,');
  writeln ('kazdou na zvlastni radek.');
  for I := 1 to N do { čtení hodnot }
    readln (NAMHODNOTY[I]); { čtení i–té hodnoty }
end; { VSTUPHODNOT }

function VYPOCETPRUMERU (N:TYPPOCTU;     { počet hodnot }
                         var NAMHODNOTY: TABULKA) { hodnoty }
                         : real;
{ výpočet aritmetického průměru }
var
  SOUCET : real;   { součet hodnot }
  I: TYPPOCTU;     { řídicí proměnná cyklu }

begin { VYPOCETPRUMERU }
  SOUCET := 0.0;
  for I := 1 to N do
    SOUCET := SOUCET + NAMHODNOTY[I];
  VYPOCETPRUMERU := SOUCET / N;
end; { VYPOCETPRUMERU }

procedure ODCHMERENI (N:TYPPOCTU;              { počet hodnot }
                      PRUMER: real;            { aritm. průměr }
                      var NAMHODNOTY:TABULKA;  { hodnoty }
                      var ODCHYLKY :TABULKA);  { odchylky }
{ výpočet odchylek od průměru }
var
  I: TYPPOCTU;     { řídicí proměnná cyklu }
begin { ODCHMERENI }
```

```
   for I := 1 to N do
      ODCHYLKY[I] := NAMHODNOTY[I] – PRUMER;
end; { ODCHMERENI }

function VYSMODCHYLKY (N:TYPPOCTU;          { počet hodnot }
                       var ODCHYLKY: TABULKA)  { odchylky }
                       : real;
{ výpočet směrodatné odchylky průměru }
var
   SOUCET : real;   { součet druhých mocnin }
   I: TYPPOCTU;     { řídicí proměnná cyklu }
begin { VYSMODCHYLKY }
   SOUCET := 0.0;
   for I := 1 to N do
      SOUCET := SOUCET + sqr(ODCHYLKY[I]);
   VYSMODCHYLKY := sqrt(SOUCET/(N * (N–1)));
end; { VYSMODCHYKY }

procedure VYSTUP (PRUMER: real; { aritm. průměr }
                  SMODCHYLKA : real); { směr. odchylka }
{ výstup vypočtených hodnot }
begin { VYSTUP }
   writeln (PRUMER:10:5,' +/–',SMODCHYLKA:10:5);
end; { VYSTUP }

begin { MERENI }
   { vstup naměřených hodnot }
   VSTUPHODNOT (N, NAMHODNOTY);
   { výpočet aritmetického průměru }
   PRUMER := VYPOCETPRUMERU(N,NAMHODNOTY);
   { odchylky měření }
   ODCHMERENI (N, PRUMER, NAMHODNOTY, ODCHYLKY);
   { výpočet směrodatné odchylky }
   SMODCHYLKA := VYSMODCHYLKY (N, ODCHYLKY);
   { výstup průměru a směrodatné odchylky }
   VYSTUP (PRUMER, SMODCHYLKA);
end. { MERENI }
```

Program 7.1

Při deklaraci procedury ODCHMERENI jsme použili vstupního parametru NAMHODNOTY typu TABULKA. Přestože jde o vstupní parametr, deklarovali jsme jej jako parametr *volaný odkazem*. Jak víme, pro parametry volané hodnotou se při aktivaci procedury vytváří lokální proměnná a hodnota skutečného parametru (u pole všechny složky) se se do ní kopíruje. To může být u strukturovaných datových typů časově i paměťově náročné. Proto pokud uvnitř procedury není volání hodnotou nezbytné, je výhodnější (přesto, že nejde o parametr výstupní) parametr deklarovat jako volaný odkazem.

Strukturované datové typy zabírají v paměti obvykle značný prostor. Mimo jiné to bývá i proto, že jejich reprezentace v paměti nemusí být souvislá a může obsahovat i některá nevyužitá místa.

Chceme-li, aby překladač přeložil náš program tak, že je dávána přednost úsporné reprezentaci datové struktury v paměti před rychlým přístupem k jejím složkám, využijeme vyhrazeného slova **packed** (packed = zhuštěný) v deklaraci typu datové struktury. Můžeme např. napsat

packed array [1..MAXN] **of** 0..255

nebo

packed array [1..15] **of** *char*

7.2 Řetězce znaků

Hodnoty strukturovaného typu nemůžeme obvykle zapsat do programu přímo. Neexistují totiž literály strukturovaných typů. Podobně nelze nad hodnotami strukturovaných typů provádět operace aritmetického výrazu a relační operace.

Výjimkou je strukturovaný datový typ, jehož literály jsme již poznali a využívali v příkazech *write* a *writeln*. Byly to literály nazvané *řetězce znaků*. Nyní již můžeme (se znalostí datové struktury zhuštěné pole) popsat nejen literály, ale i celou deklaraci tohoto datového typu.

Datový typ řetězců znaků délky N je deklarován jako zhuštěné pole

packed array [1..N] **of** *char*

přičemž interval určující délku řetězce musí být celočíselný a musí mít dolní hranici 1. Horní hranice N určuje délku řetězce znaků ve znacích.

Jak již bylo řečeno, můžeme kromě literálů vytvářet i pojmenované konstanty datového typu řetězců. Například

```
const
  ZAHLAVI   =
      'PROGRAM PRO VYPOCET DRUHE ODMOCNINY';
  CISLICE   = '0123456789';
```

Počet znaků v konstantě určuje délku, a tím i konkrétní typ. Konstanta ZAHLAVI je tedy typu (do celkového počtu znaků zahrnujeme i mezery)

packed array [1..35] **of** *char*

a konstanta CISLICE je datového typu

packed array [1..10] **of** *char*

Deklarujeme-li proměnné

```
var
  RET35 : packed array [1..35] of char;
  RET10 : packed array [1..10] of char;
```

je možné provést přiřazení

```
RET35 := ZAHLAVI;
RET10 := CISLICE;
```

Přiřazovat hodnoty řetězců jiné délky, než je uvedena v deklaraci proměnné, Pascal nedovoluje. Nelze proto např. zapsat

```
RET10 := ZAHLAVI;
RET35 := CISLICE;
```

Tato vlastnost pascalských řetězců je nevýhodná. Proto mnoho překladačů umožňuje přiřazovat proměnným typu řetězec znaků hodnoty řetězců libovolné délky. Pokud je přiřazovaná hodnota kratší, doplní se *zprava* mezerami. Je-li delší, přebytečné znaky se zanedbají (odříznou).

Například po přiřazení

```
RET10 := ZAHLAVI;
RET35 := CISLICE;
```

by proměnná RET35 měla hodnotu

'0123456789

a proměnná RET10 hodnotu

'PROGRAM PR'

Z operací výrazu je pro řetězce znaků povoleno užívat relačních operací =, < >, <, >, < =, > = s výsledkem typu *boolean*. Uspořádání řetězců je

dáno znakovým kódem. Nejvyšší váhu má znak nejvíce vlevo, potom druhý zleva atd. (podobně jako u jmen uspořádaných v telefonním seznamu). Ve standardním Pascalu lze porovnávat pouze řetězce stejných délek. Často užívané praktické rozšíření dovoluje porovnávat i různě dlouhé řetězce. Kratší z porovnávaných řetězců se pro potřeby porovnání doplní na délku delšího mezerami zprava.

Hodnoty řetězců znaků není dovoleno standardním Pascalem přímo číst ze vstupního zařízení. Při čtení těchto hodnot je třeba užít cyklu a řetězec číst znak po znaku, např.:

```
for I := 1 to 10 do
  read (RET10 [I])
```

Často je však rozšířením definice jazyka dovoleno tento zápis zkrátit na

```
read (RET10)
```

a znakové řetězce číst přímo jediným příkazem *read* nebo *readln*.

Výstup řetězců znaků vcelku je povolen standardně, a to nejen ve formě výstupu konstant (jak jsme byli zvyklí)

```
write (ZAHLAVI);
writeln ('Zadej hodnotu n');
```

ale i ve formě výstupu proměnných

```
write (RET10);
writeln (RET35);
```

Uvedené příkazy vypíšou okamžitou hodnotu řetězců RET10 a RET35.

Podobně jako u výstupu znaků, lze i u výstupu řetězců užít formátu

řetězec : celočíselný výraz určující délku výstupu

Je-li zadaná délka výstupu menší než skutečná délka řetězce, vytiskne se jen určená část hodnoty zleva. Je-li délka ve formátu větší, doplní se vystupující řetězec *zleva* mezerami na potřebnou délku. Například příkazy

```
RET10 := CISLICE;
writeln (RET10);
writeln (RET10:5);
writeln (RET10:13);
```

způsobí výpis

```
0123456789
01234
   0123456789
```

Zdůrazněme ještě jednou, že uvedené vlastnosti platí pouze pro *řetězce znaků*

packed array [1..N] **of** *char*

Pro jiné zhuštěné strukturované datové typy, a to ani jiná zhuštěná pole, neplatí.

Přístup ke složkám zhuštěných a nezhuštěných datových typů je z hlediska tvůrce programu týž. Jsou-li deklarována pole

```
var
  POLE1 : array [1..5] of char;
  POLE2 : array [1..10] of char;
  POLE3 : packed array [1..5] of char;
  POLE4 : packed array [1..10] of char;
```

je např. přístup ke třetí složce všech polí zapisován stejně

POLE1 [3] POLE2 [3] POLE3 [3] POLE4 [3]

Vzájemné přiřazení zhuštěných a nezhuštěných datových struktur stejného základního typu struktury přiřazovacím příkazem není možné. Následující zápis *není* z těchto důvodů v Pascalu povolen

```
POLE1 := POLE3;
POLE4 := POLE2
```

Existují však dvě standardní procedury pro vzájemné přiřazení těchto dvou forem pole. Nechť N, Z jsou proměnné typu pole. N je v nezhuštěné a Z ve zhuštěné formě. Příkaz

pack(N,I,Z)

naplní celé zhuštěné pole Z hodnotami složek nezhuštěného pole N počínajíc I-tou složkou pole N. Příkaz

unpack(Z,N,I)

rozvine hodnoty celého zhuštěného pole Z do složek nezhuštěného pole N počínajíc I – tou složkou pole N. Počet složek přenesených z pole N do pole Z, resp. nazpět, je dán počtem složek zhuštěného pole Z. Pokud počet složek pole N není dostatečný pro přenos všech složek celého pole Z, dojde k chybě při spuštění programu.

7.3 Jiné typy indexu

O datovém typu indexu bylo řečeno, že jde o libovolný ordinální typ. Až dosud jsme užívali pouze celočíselného intervalu, což je skutečně nejčastější

datový typ pro indexování. Ukažme si nyní praktické využití i jiného typu indexu (nejen celočíselného).

Jde o program pro výpočet kalorické hodnoty vybraného pokrmu. Program nabídne uživateli jídelní lístek a poskytne mu možnost vybrat si z nabídky libovolný počet jednotlivých druhů jídel. Po provedení objednávky se vypočte a vypíše kalorická hodnota objednaných druhů jídel i celková kalorická hodnota celého jídla. Kalorické hodnoty potravin byly převzaty z [10]. Základní struktura programu je tato:

```
program JIDLO (input, output);
var
  JOULY          : SEZJOULY;  { kalorické hodnoty }
  NABIDKA        : SEZNABIDKA; { popisy jídel }
  OBJEDNAVKA     : SEZOBJED;  { objednané počty }

begin { JIDLO }
  { přiřazení počátečních hodnot }
  POCATHODNOTY (JOULY, NABIDKA, OBJEDNAVKA);
  { výpis jídelního lístku }
  VYPISLISTKU (NABIDKA);
  { objednávka jídel }
  OBJJIDEL (NABIDKA, OBJEDNAVKA);
  { výpis kalorické hodnoty }
  VYPISHODNOTY (NABIDKA, OBJEDNAVKA);
end. { JIDLO }
```

Základním datovým typem programu bude výčtový typ JIDLA. V něm budou identifikátorem pojmenována všechna jídla, která jsou předmětem našeho zájmu.

```
type
  JIDLA =
    (VLSALAT, SARDINKY, ZAVINAC, SLANINA, HERMELIN,
    EIDAM, TVARUZKY, MASLO, KLOBASY, PARKY,
    UHERSAM, UZENE, JAHDZEM, MERDZEM, SACHR,
    BABOVKA, ROHLIK, CAJ, MOST, DZUS, PLZENSKE);
```

Tři proměnné typu pole JOULY, NABIDKA a OBJEDNAVKA budou indexovány typem JIDLA.

```
type
  TYPJOULY       = 0 .. maxint;   { typ kalorické hodnoty }
```

```
  TYPNABIDKA   = packed array [1..30] of char; { popis nabídky }
  TYPOBJED     = 0 .. maxint;   { typ objednaného počtu kusů }
  SEZJOULY     = array [JIDLA] of TYPJOULY;
                                { seznam kalorických hodnot }
  SEZNABIDKA = array [JIDLA] of TYPNABIDKA;
                                { seznam popisů jídel }
  SEZOBJED   = array [JIDLA] of TYPOBJED;
                                { seznam objednaných počtů }
var
  JOULY        : SEZJOULY;  { kalorické hodnoty }
  NABIDKA      : SEZNABIDKA; { popisy jídel }
  OBJEDNAVKA   : SEZOBJED;  { objednané počty }
```

Dvě z nich, JOULY a NABIDKA, mají konstantní hodnotu. Vzhledem k tomu, že v Pascalu neexistují konstanty strukturovaného datového typu, musíme i pro vyjádření konstantních hodnot užít proměnných. Ty je nutno v úvodu programu složku po složce naplnit.

Tento rys jazyka je nevýhodný jak z hlediska plýtvání pamětí (konstantní hodnoty existují v paměti dvakrát, jednak jako konstanty pro vyjádření hodnot složek, jednak jako skutečné složky pole), tak z hlediska časového (je zbytečně spotřebován čas výpočtu k naplnění pole). S danými jazykovými prostředky jej však není dost dobře možné obejít. Proto častým rozšířením jazyka je možnost přiřazení počátečních hodnot proměnným v deklaraci i možnost zápisu strukturovaných konstant.

Třetí proměnná typu pole OBJEDNAVKA je již proměnnou s měnící se hodnotou a obsahuje objednané počty potravin.

Kompletní program je označen 7.2.

```
program JIDLO (input, output);
{ Program pro výpočet kalorické hodnoty stravy. Na základě zvoleného
jídla z jídelníčku vypočte celkovou kalorickou hodnotu v kJ.

Popis vstupů a výstupů :
    Nejprve se vypíše nabídka všech jídel. Potom se nabízí jednotlivá jídla
    a uživatel odpovídá celým číslem. Hodnoty 0 .. n určují, kolik udaných
    jednotek si vybral. Výběr jídel je ukončen zadáním záporného čísla.
    Nakonec je vypsán seznam vybraných jídel s příslušnou kalorickou
    hodnotou a celková kalorická hodnota
}
```

```
type
  JIDLA = (VLSALAT, SARDINKY, ZAVINAC, SLANINA,
           HERMELIN, EIDAM, TVARUZKY, MASLO,
           KLOBASY, PARKY, UHERSAM, UZENE, JAHDZEM,
           MERDZEM, SACHR, BABOVKA, ROHLIK, CAJ,
           MOST, DZUS, PLZENSKE);
           { výčet jednotlivých jídel }
  TYPJOULY      = 0 .. maxint;   { typ kalorické hodnoty }
  TYPNABIDKA    = packed array [1..30] of char;
                                 { řetězec pro popis nabídky }
  TYPOBJED      = 0 .. maxint; { typ objednaného počtu kusů }
  SEZJOULY      = array [JIDLA] of TYPJOULY;
                                 { seznam kalorických hodnot }
  SEZNABIDKA    = array [JIDLA] of TYPNABIDKA;
                                 { seznam popisů jídel }
  SEZOBJED      = array [JIDLA] of TYPOBJED;
                                 { seznam objednaných počtů }

var
  JOULY      : SEZJOULY;     { kalorické hodnoty }
  NABIDKA    : SEZNABIDKA; { popisy jídel }
  OBJEDNAVKA : SEZOBJED; { objednané počty jídel }

procedure POCATHODNOTY (var JOULY: SEZJOULY;
                        var NABIDKA: SEZNABIDKA;
                        var OBJEDNAVKA : SEZOBJED);
{ nastavení počátečních hodnot }
var
  I       : JIDLA;     { řídicí proměnná }
begin { POCATHODNOTY }
  NABIDKA[VLSALAT]    := '100g Vlassky salat          ';
  NABIDKA[SARDINKY]   := '100g Sardinky v oleji       ';
  NABIDKA[ZAVINAC]    := '100g Zavinace               ';
  NABIDKA[SLANINA]    := '100g Anglicka slanina       ';
  NABIDKA[HERMELIN]   := '100g HERMELIN – syr         ';
  NABIDKA[EIDAM ]     := '100g Syr EIDAM              ';
  NABIDKA[TVARUZKY]   := '100g Olomoucke tvaruzky     ';
  NABIDKA[MASLO ]     := '50g Maslo vyberove          ';
  NABIDKA[KLOBASY]    := '100g Moravske klobasy       ';
```

```
    NABIDKA[PARKY ]      :=  '100g Jemne parky          ';
    NABIDKA[UHERSAM]     :=  '100g Uhersky salam        ';
    NABIDKA[UZENE ]      :=  '100g Moravske uzene       ';
    NABIDKA[JAHDZEM]     :=  '50g Jahodovy dzem         ';
    NABIDKA[MERDZEM]     :=  '50g Merunkovy dzem        ';
    NABIDKA[SACHR ]      :=  '1 ks Sachruv dort         ';
    NABIDKA[BABOVKA]     :=  '1 ks Trena babovka        ';
    NABIDKA[ROHLIK]      :=  '1 ks Rohlik               ';
    NABIDKA[CAJ   ]      :=  'Salek caje                ';
    NABIDKA[MOST]        :=  '0,2l Most jablecny        ';
    NABIDKA[DZUS]        :=  '0,2l Pomerancovy dzus     ';
    NABIDKA[PLZENSKE]    :=  '0,5l Plzenske pivo 12     ';
    JOULY[VLSALAT]       :=  1046;
    JOULY[SARDINKY]      :=  1407;
    JOULY[ZAVINAC]       :=  736;
    JOULY[SLANINA]       :=  2654;
    JOULY[HERMELIN]      :=  1134;
    JOULY[EIDAM]         :=  1088;
    JOULY[TVARUZKY]      :=  575;
    JOULY[MASLO]         :=  1506;
    JOULY[KLOBASY]       :=  878;
    JOULY[PARKY]         :=  1319;
    JOULY[UHERSAM]       :=  2100;
    JOULY[UZENE]         :=  1470;
    JOULY[JAHDZEM]       :=  527;
    JOULY[MERDZEM]       :=  523;
    JOULY[SACHR]         :=  815;
    JOULY[BABOVKA]       :=  609;
    JOULY[ROHLIK]        :=  576;
    JOULY[CAJ]           :=  0;
    JOULY[MOST]          :=  378;
    JOULY[DZUS ]         :=  411;
    JOULY[PLZENSKE]      :=  966;
  for I := VLSALAT to PLZENSKE do
    OBJEDNAVKA[I] := 0; { vynulování objednávek }
end; { POCATHODNOTY }

procedure VYPISLISTKU (var NABIDKA: SEZNABIDKA);
{ výpis jídelního lístku }
```

```
var
  I        : JIDLA;      { řídicí proměnná }
begin { VYPISLISTKU }
  writeln ('VYPOCET KALORICKE HODNOTY');
  writeln;
  writeln ('Stiskni klavesu ENTER nebo RETURN');
  readln;
  writeln ('Jidelni listek');
  writeln ('= = = = = = =  = = = = = = =');
  for I := VLSALAT to PLZENSKE do
    writeln (NABIDKA[I]);
  writeln ('Stiskni klavesu ENTER nebo RETURN');
  readln;
end; { VYPISLISTKU }

procedure OBJJIDEL (var NABIDKA: SEZNABIDKA;
                    var OBJEDNAVKA: SEZOBJED);
{ objednávka jídel }
var
  POCET: integer;      { uživatelem zadaný počet }
  I        : JIDLA;    { řídicí proměnná }
begin { OBJJIDEL }
  writeln ('Objednavej cislem udavajicim pocet jednotek');
  writeln ('Zaporne cislo objednavku ukonci');
  I := VLSALAT; { první objednávané jídlo }
  repeat
    writeln (NABIDKA[I]); { výpis nabídky }
    readln (POCET);     { čtení zadaného počtu }
    if POCET > 0 then  { něco bylo objednáno }
      OBJEDNAVKA[I] := OBJEDNAVKA[I] + POCET;
      { zaznamenání }
    if I = PLZENSKE then { poslední v seznamu }
      I := VLSALAT       { opět na začátek seznamu }
    else
      I := succ(I);     { na následníka }
  until POCET < 0;   { záporné číslo ukončí cyklus }
end; { OBJJIDEL }
procedure VYPISHODNOTY (var NABIDKA: SEZNABIDKA;
                        var OBJEDNAVKA: SEZOBJED);
```

```
{ výpis kalorické hodnoty }
var
   CELKEM    : TYPOBJED; { součet kJ }
   I         : JIDLA;  { řídicí proměnná }
   KALORICHOD : TYPJOULY; { kalorická hodnota jednoho druhu }
begin { VYPISHODNOTY }
   writeln;
   writeln;
   writeln ('OBJEDNANO                    kJ');
   writeln;
   CELKEM := 0; { součet kJ }
   for I := VLSALAT to PLZENSKE do
      if OBJEDNAVKA[I] > 0 then
         begin { přes všechny objednávky }
            { počet kJ }
            KALORICHOD := OBJEDNAVKA[I] * JOULY[I];
            writeln (OBJEDNAVKA[I]:2,'x ',NABIDKA[I],
                     KALORICHOD:6);
            { do celkového součtu }
            CELKEM := CELKEM + KALORICHOD;
         end; { přes všechny objednávky }
   writeln ('-------------------------------');
   writeln ('Celkem                ',CELKEM:7);
end; { VYPISHODNOTY }

begin { JIDLO }
   { přiřazení počátečních hodnot }
   POCATHODNOTY (JOULY, NABIDKA, OBJEDNAVKA);
   { výpis jídelního lístku }
   VYPISLISTKU (NABIDKA);
   { objednávka jídel }
   OBJJIDEL (NABIDKA, OBJEDNAVKA);
   { výpis kalorické hodnoty }
   VYPISHODNOTY (NABIDKA, OBJEDNAVKA);
end. { JIDLO }
```

Program 7.2

Procedura POCATHODNOTY naplní hodnoty konstantních polí JOULY a NABIDKA. Dále vynuluje pole objednávek OBJEDNAVKA. Výpis

nabídky ve formě jídelního lístku provede procedura VYPISLISTKU. Potom jsou znovu nabízena (procedurou OBJJIDEL) jednotlivá jídla a uživatel zadává objednané počty v uvedených množstvích. Objednávka je realizována cyklem repeat, přičemž konec objednávky je ze strany uživatele vyjádřen zadáním záporného čísla.

Uživatel může nabízená jídla procházet při objednávce i několikrát, neboť po nabídce posledního jídla PLZENSKE následuje opět nabídka prvního jídla VLSALAT. Je to zajištěno příkazem:

```
if I = PLZENSKE then    { poslední v seznamu }
  I:= VLSALAT
else                    { není poslední }
  I := succ (I)
```

Po objednávce je proveden procedurou VYPISHODNOTY výpis objednaných jídel (je-li objednaný počet OBJEDNAVKA[I] > 0) s jejich kalorickou hodnotou cyklem for. Současně je prováděn součet celkové kalorické hodnoty do proměnné CELKEM.

V proceduře VYPISLISTKU chceme po ukončení výpisu záhlaví a jídelního lístku zastavit běh programu a nechat uživatele lístek v klidu přečíst. To je realizováno dvojicí příkazů

```
writeln ('Stiskni klavesu ENTER nebo RETURN');
readln;
```

Příkaz *readln* bez parametrů bude čekat tak dlouho, dokud nebude proveden vstup řádku (i prázdného) stlačením klávesy ENTER, resp. RETURN.

7.4 Vícerozměrná pole

V programu 7.2 jsme viděli, že složka pole může být nejen jednoduchého typu (např. typu NEZAP), ale i strukturovaného (např. TYPNABIDKA je zhuštěné pole).

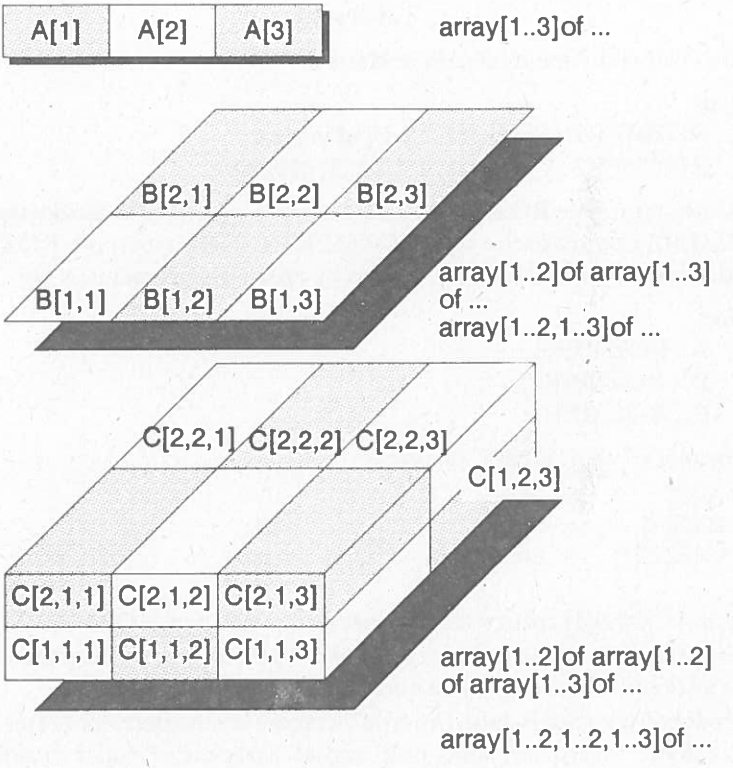
Je-li složkou pole opět pole se složkami jednoduchého typu, jako v případě datového typu

```
array [JIDLA] of TYPNABIDKA
```

obdržíme tzv. *dvourozměrné pole* (*matici*). Je-li matice opět složkou pole, obdržíme třírozměrné pole (krychli). Tak můžeme pokračovat až do libovolného rozměru pole. Grafické zobrazení jedno-, dvou- a třírozměrného pole je na obr. 7.3. Je obvyklé nazývat první index matice *řádkovým* indexem a druhý *sloupcovým* indexem, neboť označují řádky, resp. sloupce matice (obr. 7.3).

Deklarace datových typů vícerozměrného pole ROZMER2 a ROZMER3 z obr. 7.3 má tvar (předpokládáme, že jednoduchý typ složky na poslední úrovni je *integer*):

```
type
  ROZMER1 = array [1..3] of integer;
  ROZMER2 = array [1..2] of array [1..3] of integer;
  ROZMER3 = array [1..2] of array [1..2] of array [1..3] of integer;
```



Obr. 7.3 Vícerozměrná pole

Je možné užít i předchozích deklarací typů

```
type
  ROZMER1 = array [1..3] of integer;
```

```
ROZMER2 = array [1..2] of ROZMER1;
ROZMER3 = array [1..2] of ROZMER2;
```

nebo i jiné kombinace, např.

```
type
  ROZMER3 = array [1..2] of array [1..2] of ROZMER1;
```

Pro zkrácení zápisu je možné v deklaraci datového typu vícerozměrného pole nahradit posloupnost znaků

```
] of array [
```

čárkou. Tím obdržíme stručnější deklarace

```
type
  ROZMER2 = array [1..2,1..3] of integer;
  ROZMER3 = array [1..2,1..2,1..3] of integer;
```

Složky typu pole ROZMER1 jsou datového typu *integer*. Složky typu pole ROZMER2 jsou datového typu ROZMER1 a složky typu pole ROZMER3 jsou datového typu ROZMER2. Pokud by existovaly proměnné z obr. 7.3

```
var
  A : ROZMER1;
  B : ROZMER2;
  C : ROZMER3;
```

bylo by možné provést např. přiřazení

```
A [2]       := 2;
B [2][1]    := 8;       B [1]    := A;
C [2][2][1] := 16;      C [2][1] := A;       C [1] := B;
```

Zápis C[2][2][1] zpřístupňuje 2.složku datového typu ROZMER3, která je typu ROZMER2. Její 2.složka je typu ROZMER1 a její 1.složka je typu *integer*. Proto jí lze přiřadit celočíselnou hodnotu 16.

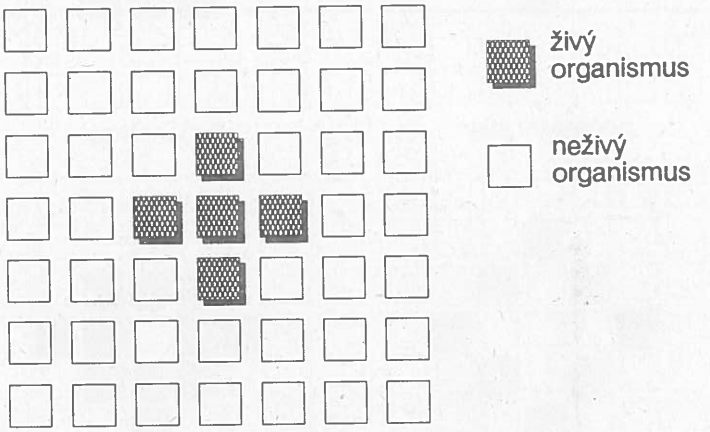
Podobně jako popis datového typu vícerozměrného pole, lze i zápis přístupu ke složce vícerozměrného pole zapsat zkráceně. Dvojici znaků] [nahradíme čárkou a můžeme psát např.

```
B [2,1]     := 8;
C [2,2,1]   := 16;      C [2,1]  := A;
```

Tak jako u jednorozměrných polí, ani zde není datový typ indexu omezen na celá čísla a jejich intervaly. Lze užívat všech ordinálních typů a jejich kombinací.

Zahrajme si nyní tzv. *hru života* (game of life), kterou vymyslel anglický matematik John Conway.

Je to hra pro jednoho hráče. Hraje se obvykle na čtvercové hrací desce, ale efekt má pouze tehdy, můžeme-li sledovat tvar vytvořený živými organismy měnících se generací na obrazovce počítače. Na desce jsou políčka podobně jako na šachovnici. (obr. 7.4).

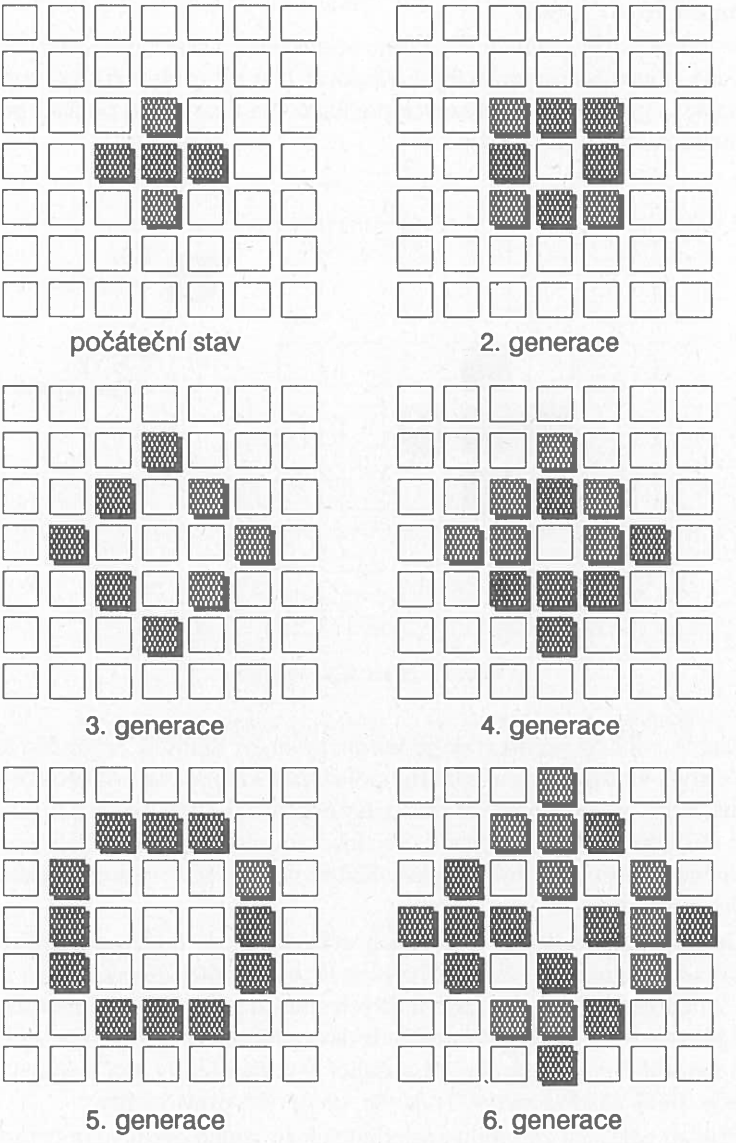


Obr. 7.4 Hrací deska hry života

Každé políčko představuje jednoduchý organismus, který může být živý nebo mrtvý. Ve hře je to znázorněno položením kamene na políčko (živý organismus) nebo prázdným políčkem (mrtvý organismus). Podle umístění na hrací desce může mít každý organismus nejvíce 8 sousedů (mimo okraj hrací desky) a nejméně 3 sousedy (v rohu desky). Každé políčko tedy sousedí s maximálně 8 jinými políčky.

Celé společenství žije v životních cyklech, které budeme nazývat generacemi. Je-li organismus živý, pak přežije do následující generace pouze tehdy, má-li 2 nebo 3 živé sousedy. Má-li živých sousedů méně, umírá na *osamělost*, má-li jich více, umírá na *přelidnění*. Je-li organismus mrtvý, může se na jeho místě narodit živý organismus následující generaci tehdy, má-li právě tři živé sousedy. Jinak zůstává mrtvý. To je vše, co se týká pravidel hry.

Příklad několika za sebou následujících životních cyklů společenství ukazuje obr 7.5.



Obr. 7.5 Příklad životních cyklů hry života

Je výhodné mít možnost, kromě zadání základního tvaru živých organismů, zasáhnout do průběhu generování a modifikovat právě vytvořený obrazec společenství.

Základní schéma programu realizujícího hru života je:

```
program ZIVOT (input, output);
var
  DESKA ,                  { hrací deska }
  B      : ORGANISMY;      { pomocná hrací deska }
  ZNAK   : char;           { pomocná proměnná }
begin { ZIVOT }
  INICIALIZACE; { inicializace }
  repeat
    POCNASTAVENI; { počáteční nastavení }
    repeat
      NOVAGENERACE; { výpočet nové generace }
      writeln ('Chces dalsi generaci ?');
      readln (ZNAK); { čtení odpovědi }
    until (ZNAK = 'n') or (ZNAK = 'N');
    writeln ('Chces doplnit zadani ?');
    readln (ZNAK);   { čtení odpovědi }
  until (ZNAK = 'n') or (ZNAK = 'N');
  writeln ('Konec programu');
end. { ZIVOT }
```

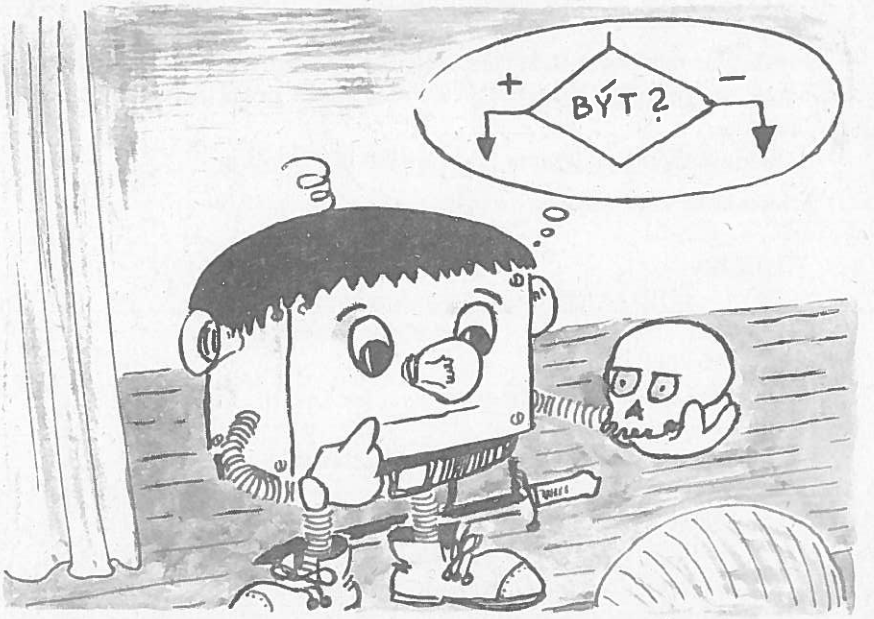
Hrací desku nejlépe v programu popíšeme jako proměnnou typu dvourozměrného pole ORGANISMY. Složky této matice nabývají hodnot výčtového typu

```
type
  ORGANISMUS = (NEZIVY, ZIVY);
```

Skutečné rozměry pole vyjádříme pojmenovanou konstantou pro pozdější snadnější modifikaci rozměrů.

```
const
  SKVELIKOST = 25;   { velikost hrací desky }
type
  ORGANISMY = array [0..SKVELIKOST,0..SKVELIKOST]
              of ORGANISMUS; { datový typ hrací desky }
```

Deklarujeme-li proměnnou




```
var
   DESKA : ORGANISMY;
```

můžeme pro každou složku pole DESKA[I, J], *která neleží na okraji pole*, zjistit počet živých sousedů výrazem

```
ord ( DESKA [I–1,J–1] ) +
ord ( DESKA [I–1,J ]   ) +
ord ( DESKA [I–1,J+1] ) +
ord ( DESKA [I ,J–1]   ) +
ord ( DESKA [I ,J+1])   +
ord ( DESKA [I+1,J–1] ) +
ord ( DESKA [I+1,J ])   +
ord ( DESKA [I+1,J+1])
```

(7.1)

Ordinální číslo hodnoty ZIVY (reprezentující živý organismus) je totiž 1. Ordinální číslo hodnoty NEZIVY je 0. Proto součtem ordinálních čísel sousedů obdržíme skutečný počet těch, jejichž hodnota je ZIVY.

Problémy jsou s těmi organismy, které žijí na okraji společenství (s hodnotami indexů [I,0], [I,SKVELIKOST], [0,I] a [SKVELIKOST,I] pro I od 0 do SKVELIKOST). Tyto složky matice mají menší počet sousedů, a proto výraz (7.1) by pro ně musel mít jiný (omezený) tvar. To by program zbytečně komplikovalo.

Jednodušší bude, když tyto krajní sloupce a řádky naplníme neživými organismy a jako hrací desku budeme využívat pouze ty sloupce a řádky desky, které neleží na okraji, tj. s hodnotami indexů od 1 do (SKVELIKOST–1). Potom výraz pro výpočet počtu živých sousedů bude pro všechny zkoumané prvky jednotný. Krajní organismy nebudou v průběhu generování svůj stav měnit.

Nový tvar společenství je třeba vytvářet na jiné hrací desce. Proto máme k dispozici pomocnou desku B, na kterou vytváříme nový tvar společenství. Přímo na původní hrací desce nelze stav organismů průběžně měnit, neboť bychom pak prováděli chybný součet živých sousedů pro složky s vyššími indexy (výpočet by probíhal částečně s hodnotami další generace). Použijeme tedy desku B a po vytvoření další generace zkopírujeme novou generaci zpět na původní desku.

Průchod přes všechny prvky dvourozměrného pole provedeme cyklem for, v němž je vnořen další příkaz cyklu for. Každý z cyklů mění hodnoty jednoho z indexů. Vnitřní cyklus mění hodnoty indexu rychleji, proto je obvyklé, aby pracoval se sloupcovým indexem. Vnější cyklus mění hodnoty řádkového indexu.

```
for I:=1 to VELIKOST do
  for J:=1 to VELIKOST do
    begin { výpočet podmínky života }

      POCET := výraz (7.1);

      if (POCET=3) or ((POCET=2) and (DESKA[I,J]=ZIVY))
      then { splněna podmínka pro živý organismus }
        B[I,J] := ZIVY
      else
        B[I,J] := NEZIVY;
    end; { výpočet podmínky života }
```

Celý program se všemi deklaracemi je označen 7.3.

```
program ZIVOT (input, output);
{ Programová realizace "hry života" podle Johna Conwaye. Společenství
tvoří živé a neživé organismy. Do další generace přežije organismus, má-li
2 nebo 3 sousedy, jinak zemře. Do další generace vznikne živý organismus
na místě mrtvého, má-li právě 3 sousedy, jinak zůstane mrtvý.

Vstupy:
    1. Po výzvě
  Zadej souradnice x,y ziveho organismu

  dvě celá čísla udávající souřadnice na jednom řádku oddělené mezerou.

    2. Po výzvě
  Chces dalsi generaci

  písmeno N udává přerušení generací a přechod k zadávání.

    3. Po výzvě
  Chces doplnit zadani

  písmeno N udává zápornou odpověď a způsobí ukončení programu.

Výstup: Po vytvoření každé nové generace se zobrazí hrací deska. Živé
  organismy jsou vyjádřeny znakem * , mrtvé mezerou.
}
const
  SKVELIKOST  = 25;     { skutečná velikost hrací desky }
  VELIKOST    = 24;     { využitelná velikost hrací desky }

type
  ORGANISMUS = (NEZIVY, ZIVY);                    { typ organismu }
  TYPSOURADNICE = 0 .. SKVELIKOST;                { typ souřadnice }
  ORGANISMY = array [TYPSOURADNICE,TYPSOURADNICE]
        of ORGANISMUS;                            { typ hrací desky }

var
  DESKA  ,                  { hrací deska }
  B      : ORGANISMY;       { pomocná hrací deska }
  ZNAK   : char;            { pomocná proměnná }
```

```
procedure INICIALIZACE;
{ inicializace hrací desky }
var
   I,J      : TYPSOURADNICE; { řídicí proměnné cyklů }
begin { INICIALIZACE }
   writeln ('HRA ZIVOTA');
   writeln;
   for I:=0 to SKVELIKOST do { neživé organismy }
      for J:=0 to SKVELIKOST do
         begin
            DESKA [I,J]:= NEZIVY;
            B [I,J] := NEZIVY;
         end;
end; { INICIALIZACE }

procedure VYSTUP;
{ výstup stavu společenství na obrazovku }
var
   I,J      : TYPSOURADNICE; { řídicí proměnné cyklů }
begin { VYSTUP }
   for I:=1 to VELIKOST do
      begin { po řádcích }
         for J:=1 to VELIKOST do { po sloupcích }
            if DESKA [I,J] = ZIVY then
               write ('*')    { živý je hvězdička }
            else
               write (' ');    { neživý je mezera }
         writeln;
      end; { po řádcích }
end; { VYSTUP }
procedure POCNASTAVENI;
{ nastavení počátečního stavu }
var
   I,J      : integer; { pomocné proměnné }
begin { POCNASTAVENI }
   repeat
      writeln ('Zadej souradnice x,y ziveho organismu');
      read (I);       { vstup souřadnic }
      if I <= 0 then { není třeba číst další souřadnici }
```

```
        readln
      else
        begin { není konec zadávání }
          readln(J);
            if (I<SKVELIKOST) and
              (J >0) and (J<SKVELIKOST) then
                DESKA [I,J] := ZIVY
            else { chybné zadání }
              writeln ('Chybne souradnice');
            VYSTUP; { výstup stavu po zadání }
        end; { není konec zadávání }
    until I=0;
end; { POCNASTAVENI }

procedure NOVAGENERACE;
{ vytvoření nové generace }
var
  I,J      : TYPSOURADNICE; { řídicí proměnné cyklů }
  POCET    : 0 .. 8;        { počet živých sousedů }
begin { NOVAGENERACE }
  for I:=1 to VELIKOST do
    for J:=1 to VELIKOST do
      begin { výpočet podmínky života }
        POCET :=
          ord (DESKA [I-1,J-1])
        + ord (DESKA [I-1,J ])
        + ord (DESKA [I-1,J+1])
        + ord (DESKA [I ,J-1])
        + ord (DESKA [I ,J+1])
        + ord (DESKA [I+1,J-1])
        + ord (DESKA [I+1,J ])
        + ord (DESKA [I+1,J+1]);
        if (POCET=3) or ((POCET=2) and (DESKA[I,J]=ZIVY))
        then
          B[I,J] := ZIVY
        else
          B[I,J] := NEZIVY;
      end; { výpočet podmínky života }
  DESKA := B; { zpětná kopie }
```

```
    VYSTUP; { výstup stavu společenství }
end; { NOVAGENERACE }
begin { ZIVOT }
    INICIALIZACE;
    repeat
        POCNASTAVENI; { počáteční nastavení }
        repeat
            NOVAGENERACE; { výpočet nové generace }
            writeln ('Chces dalsi generaci ?');
            readln (ZNAK); { čtení odpovědi }
        until (ZNAK = 'n') or (ZNAK = 'N');
        writeln ('Chces doplnit zadani ?');
        readln (ZNAK);    { čtení odpovědi }
    until (ZNAK = 'n') or (ZNAK = 'N');
    writeln ('Konec programu');
end. { ZIVOT }
```

Program 7.3

Při inicializaci (procedurou INICIALIACE) naplníme celou hrací i pomocnou desku (včetně okrajových sloupců a řádků) neživými organismy (hodnotami NEZIVY). Okrajové řádky a sloupce již nebudeme dále měnit.

Počáteční nastavení (procedurou POCNASTAVENI) spočívá v doplnění hodnoty ZIVY na ta místa hrací desky, která určí uživatel souřadnicemi x a y. Zadávání živých organismů končí, zadáme-li první souřadnici nulovou. Zadané hodnoty souřadnic se kontrolují na správnost. V případě zadání souřadnic mimo hrací desku dojde k výpisu

Chybne souradnice

a hrací deska zůstává v původním tvaru. Pokud nejde o ukončení úseku počátečního nastavení (hodnota první souřadnice I je kladná), je tvar hrací desky pro orientaci zobrazen na displeji. Je-li zadána hodnota souřadnice $x = 0$, cyklus končí (**until** I = 0).

Zobrazení hrací desky (procedura VYSTUP) je realizováno dvěma vnořenými cykly for. Vnitřní cyklus provede vždy výpis jednoho řádku matice DESKA, a to tak, že živé organismy vyjádří znakem * a neživé mezerou. Po výpisu řádku matice se provede příkaz *writeln*. Tímto způsobem obdržíme dvourozměrný obraz matice v přirozeném tvaru (prvek [1,1] v levém horním rohu a prvek [VELIKOST,VELIKOST] vpravo dole). Nefunkční okrajové řádky a sloupce s indexy 0 a SKVELIKOST nezobrazujeme.

Postupný výpočet nových generací (procedurou NOVAGENERACE) probíhá prvek po prvku. Pro každý organismus je vypočten počet živých sousedů a podle zadání hry určen stav prvku v nové generaci na hrací desce B. Nakonec je provedena kopie pomocné hrací desky B na původní desku a zobrazena nová generace živých organismů (procedurou VYSTUP). Výpočet nových generací je prováděn v cyklu repeat. Po každém průchodu tělem cyklu je uživatel dotázán, zda chce vytvářet další generaci. Odpověď znakem N nebo n ukončí cyklus.

```
until (ZNAK='n') or (ZNAK = 'N');
```

Funkční bloky zadávání počátečních podmínek a výpočtu nových generací tvoří tělo cyklu repeat. Tělo cyklu je provedeno znovu zadáním znaku jiného než N nebo n na dotaz

Chces doplnit zadani ?

Odpověď znakem N nebo n program ukončí. Zároveň se vypíše na obrazovku text

Konec programu

Při zadání vhodných počátečních obrazců lze dosáhnout estetických a souměrných tvarů společenství.

Tyto tvary jsou stabilní

```
* *
* *
```

nebo pulsující

```
*
*        <->      * * *
*
```

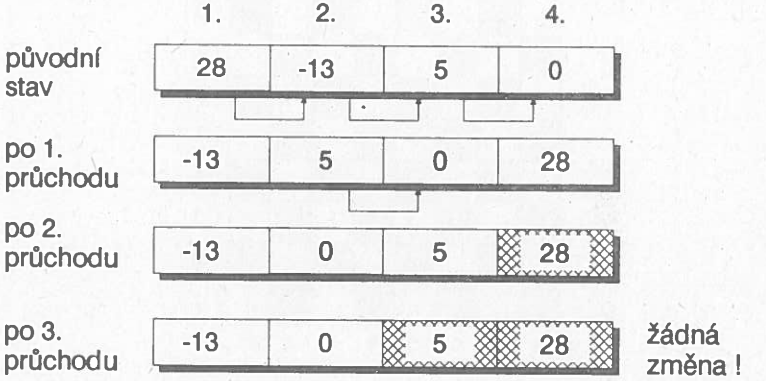
popř. se v následujících generacích neopakují.

Čtenář si sám může experimentálně nalézt další zajímavé tvary.

7.5 Cvičení

1. Vytvořte program, který ze vstupního zařízení přečte N hodnot složek celočíselného pole A a pak je setřídí vzestupně podle rostoucích hodnot indexu. Setříděné hodnoty pole pak vypíše na výstupní zařízení.
   Návod: Využijeme metody *bublinového třídění* (obr. 7.6). Základní myšlenka metody spočívá v tom, že při průchodu všemi složkami pole směrem od 1. k n-té porovnáváme vždy dvě hodnoty sousedních složek v poli (první s druhou, druhou s třetí atd.). Je-li hodnota složky s menším indexem větší

(při vzestupném třídění) než hodnota složky s větším indexem (tj. nevyhovují pozdějšímu uspořádání), obě hodnoty zaměníme. Tímto postupem se nám po průchodu celým polem objeví maximální hodnota ve složce označené indexem *n*. (Tvůrcům metody připadal postupný přesun maximální hodnoty do složky s nejvyšším indexem obdobný pohybu bublinek v sodovce směrem k hladině a odtud vznikl i název metody.) Tento postup opakujeme pro hodnoty indexu 1 až *n*–1 a tím se přesune druhá největší



Obr. 7.6 Bublinové třídění

hodnota do složky s indexem *n*–1. Postupným snižováním rozsahu na 1 až *n*–2, 1 až *n*–3 atd. pole setřídíme vzestupně.

Pokud v průběhu porovnávání pro jistý index (během jednoho průchodu) nedojde k žádné výměně, je pole již setříděno a další průchody není třeba provádět. Pozor na výměnu dvou složek pole. Zápis

```
A[I+1]:= A[I];
A[I]  := A[I+1]
```

je chybný. Je nutné využít pomocné proměnné POM

```
POM   := A[I+1];
A[I+1]:= A[I];
A[I]  := POM
```

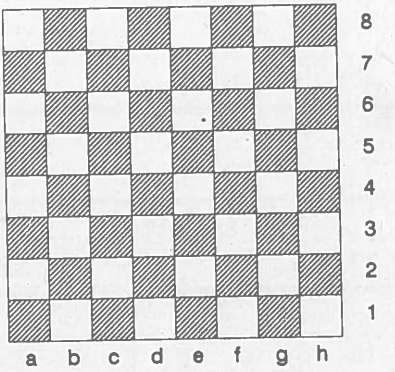
Bublinové třídění lze nalézt i v programu SPORT (kap. 8).

Varianta programu: Vytvořte program pro sestupné setřídění pole.

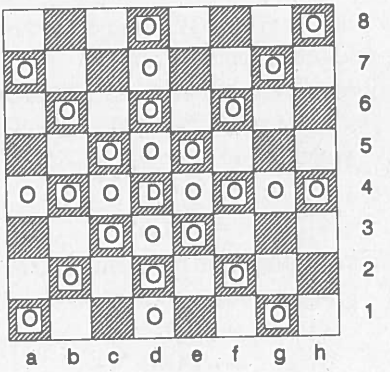
2. Vytvořte program, který ve vstupním textu ukončeném znakem '%' nalezne četnost výskytu všech písmen, číslic a mezer. Výsledek vypíše ve formě řádkového histogramu

```
A * * * * * * * * *
B * * * *
C * * * * * * * * * * * * *
```

3. Mějme šachovnici (hrací desku 8x8 polí).



Vytvořte program, který na základě zadané pozice dámy (např. d4) vykreslí šachovnici s poli ohrožených dámou, např.:



Návod: Pro reprezentaci šachovnice využijte dvourozměrné pole znaků. Políčko s dámou naplňte znakem D, ohrožená políčka znakem O, černá políčka znakem x a bílá políčka mezerou. Pole reprezentující šachovnici zaplňte příslušnými znaky a zobrazte na výstupním zařízení.

4. Vytvořte program, který nejprve řádek po řádku přečte celočíselnou matici A a pak tuto matici vytiskne jako matici transponovanou.

   Návod: Při vytváření transponované matice není nutné složky matice v paměti přemísťovat. Lze využít vhodné volby indexů při tisku. Je-li B transponovaná matice A, platí $B[j, i] = A[i, j]$, jsou tedy zaměněny řádky za sloupce. Zbytečné přesuny hodnot v paměti prodlužují výpočet, což může být podstatné pro větší matice.

5. Vytvořte program, který přečte dva řetězce znaků a zjistí, zda druhý řetězec je obsažen v prvním (tj. zda druhý řetězec je podřetězcem prvního). Pokud toto tvrzení platí, vypíše, od které hodnoty indexu začíná druhý řetězec v prvním. Například pro řetězec 'ABECEDA' a 'CED' vypíše program ANO a hodnoty 4–6, neboť řetězec 'CED' je obsažen v řetězci 'ABECEDA' od hodnoty indexu 4 do hodnoty indexu 6.
6. Deklarujte tělo procedury PODRET s hlavičkou

```
type
  RET32 = packed array [ 1..32 ] of char;
procedure PODRET (var S:RET32; C: char);
```

   která nalezne a vypíše všechny podřetězce řetězce S, které jsou odděleny znakem C. Je-li např. hodnota S

   '123=5678=0=234567= =01234=6789=12=654=321'

   a znak C = '=', bude výstupem procedury

```
123
5678
0
234567

01234
6789
12
654
321
```

7. Vytvořte procedury realizující operace součtu a součinu matic s prvky typu *real*.

   Návod: Součet matic A, B je dán vztahem $C[i,j] = A[i,j] + B[i,j]$, tj. sčítá se prvek po prvku. Součin C (velikost m řádků x n sloupců) matic A (velikosti m x p), B (velikosti p x n) je dán vztahem

   $$C[i,j] = \text{součet } (A[i,k] * B[k,j]) \text{ pro } k = 1, 2, ..., p.$$

8. Vytvořte proceduru pro zřetězení dvou řetězců znaků s hlavičkou

```
const
  DELKA = 20;
type
  RETEZEC = packed array [1..DELKA] of char;

procedure ZRETEZ (var S: RETEZEC; A,B:RETEZEC);
```

   Vstupními parametry procedury jsou řetězce A a B, které mohou ve složkách s nejvyššími indexy obsahovat mezery. Procedura odstraní z řetězce A pravostranné mezery a za tento zkrácený řetězec připojí řetězec B. Pokud je výsledný řetězec delší než DELKA, bude zkrácen na maximální délku řetězce. Výsledek bude uložen do výstupního parametru S.

8 ZÁZNAMY

8.1 Popis reálných objektů

Strukturovaný datový typ pole se vyznačuje tím, že všechny jeho složky jsou stejného datového typu, takže se jím často obtížně a nepřirozeně popisují objekty reálného světa. Každý ze skutečných objektů má totiž velký počet vlastností a ty jsou obvykle různých typů. Takový objekt v programu je pak popsán množinou hodnot obecně různých datových typů. Pro popis objektů v programu vybíráme vždy konečnou množinu pouze těch vlastností objektu reálného světa, které pro vytvářený program mají význam.

Například osobní automobily by mohly mít tyto význačné vlastnosti:

- **značka** – hodnota z množiny { Trabant, Škoda, Lada, Mercedes, Rolls-Royce },
- **barva** – hodnota z množiny { bílá, žlutá, modrá, zelená, červená, hnědá, šedá, černá },
- **obsah** – hodnota z celočíselného intervalu 500..10000,
- **tvar karosérie** – hodnota z množiny { kabriolet, sedan, kombi }.

Sportovce závodícího v běhu by bylo možno charakterizovat těmito údaji:

- **jméno a příjmení** – řetězec znaků,
- **státní příslušnost** – řetězec znaků,
- **čas startu** – celé číslo,
- **výsledný čas** – celé číslo.

Konkrétní hodnota v programu popisující konkrétní automobil, sportovce nebo jiný objekt je složena z několika hodnot obecně různého datového typu. Automobil může být popsán např. hodnotami

(Škoda, žlutá, 1199, sedan).

Závodník je popsán např. hodnotami

('Jan ','Novak ', 'TCH', 1234, 4321).

Pro přirozený popis strukturovaných hodnot, jejichž složky jsou obecně různého datového typu, zavádí Pascal datový typ *záznam* (záznam = record). Popis typu záznam má tento základní tvar:

```
record
  jméno složky1 : typ1;
  jméno složky2 : typ2;
  :
  jméno složkyn : typn;
end;
```

Každá ze složek záznamu je pojmenována identifikátorem a má obor hodnot určený datovým typem zapsaným za dvojtečkou. Počet složek záznamu je předem neomezený, ale konečný.

Pro předchozí příklady reálných objektů automobil a závodník bychom mohli definovat tyto datové typy záznam:

```
type
  AUTA      = (TRABANT, SKODA, LADA, MERCEDES,
              ROLLSROYCE);
  BARVY     = (BILA, ZLUTA, MODRA, ZELENA, CERVENA,
              HNEDA, SEDA, CERNA);
  TYPKAROSERIE = ( KABRIOLET, SEDAN, KOMBI );
  AUTOMOBIL =
    record
      ZNACKA : AUTA;
      BARVA  : BARVY;
      OBSAH  : 500 .. 10000;
      KAROSERIE : TYPKAROSERIE;
    end; { AUTOMOBIL }
  NEZAP     = 0 .. maxint;
  { pomocné definice řetězců délky 3 a 10 }
  RET3      = packed array [1.. 3] of char;
  RET10     = packed array [1..10] of char;
  { vlastní definice typu ZAVODNIK }
  ZAVODNIK =
    record
      JMENO,
```

```
    PRIJMENI        : RET10;
    STAT            : RET3;
    CASSTARTU,
    VYSLEDNYCAS : NEZAP;
end; { ZAVODNIK }
```

Existuje možnost zkrácení zápisu u více po sobě jdoucích složek téhož datového typu (je použito v předchozí definici typu ZAVODNIK). Místo zápisu

jméno složky$_1$: *typ*;
jméno složky$_2$: *typ*;
:
jméno složky$_m$: *typ*;

několika složek záznamu se stejným datovým typem lze uvést zápis

jméno složky$_1$,
jméno složky$_2$,
:
jméno složky$_m$: *typ*;

podobně jako u deklarace proměnných.

Hodnota datového typu záznam je tedy dána uspořádanou posloupností hodnot všech jeho složek.

Z matematického hlediska představuje množina hodnot typu záznam kartézský součin n obecně různých množin hodnot (datových typů) jeho složek. Hodnoty složek tvoří uspořádanou n-tici, která je z matematického hlediska prvkem kartézského součinu.

Deklaraci proměnných typu záznam provádíme obvyklým způsobem

```
var
  BEZEC  : ZAVODNIK;
  VUZ    : AUTOMOBIL;
  LYZAR  : ZAVODNIK;
```

S hodnotou typu záznam jako celkem můžeme provádět pouze *přiřazení*. Existuje však prostředek pro práci s jednou konkrétní složkou záznamu. Přístup ke složce má tento tvar

proměnná typu záznam . jméno složky

tedy proměnná oddělená tečkou od jména složky.

Například můžeme napsat

BEZEC.JMENO
VUZ.BARVA
LYZAR.PRIJMENI[8]

Zápis přístupu ke složce záznamu je stejného datového typu jako složka:

- BEZEC.JMENO datového typu RET10,
- VUZ.BARVA datového typu BARVY,
- LYZAR.PRIJMENI datového typu RET10, což je proměnná typu zhuštěné pole (podobně jako BEZEC.JMENO); je proto možné odkazovat se na osmou složku LYZAR.PRIJMENI[8].

Složkou záznamu může být opět záznam, stejně jako pole nebo jiný typ. V Pascalu platí, že (až na výjimky týkající se vnořování souborů) strukturované datové typy lze do sebe vnořovat libovolně.

Je tedy možné zapsat např.

```
type
  IDENTIFIKACE =
    record
      JMENO,
      PRIJMENI : RET10;
    end; { IDENTIFIKACE }
  OSOBA =
    record
      POJMENOVANI : IDENTIFIKACE;
      PLAT        : NEZAP;
    end; { OSOBA }
var
  VYPLATNIPASKA : array [1..100] of OSOBA;
```

Přístupy ke složkám strukturovaných datových typů se při vnoření typů do sebe řetězí, např.

```
VYPLATNIPASKA[2].POJMENOVANI.JMENO[3]:='A';
```

neboť

- VYPLATNIPASKA je pole, a proto se lze odkázat na jeho složku,
- VYPLATNIPASKA[2] je záznam typu OSOBA a má složku POJMENOVANI,
- VYPLATNIPASKA[2].POJMENOVANI je záznam typu IDENTIFIKACE a má složku JMENO,

- VYPLATNIPASKA[2] .POJMENOVANI.JMENO je zhuštěné pole znaků délky 10 a je možné je indexovat,
- VYPLATNIPASKA[2] .POJMENOVANI.JMENO[3] je typu znak, a proto je možné přiřadit znakový literál 'A'.

Identifikátory složek záznamu jsou platné uvnitř záznamu. Mimo něj se dá užít stejných identifikátorů i pro označení jiných objektů.

8.2 Zpracování výsledků

Problém, který začneme v této kapitole řešit, se týká zpracování výsledků sportovního utkání. Každý z nás viděl v televizi při přenosu z lehké atletiky nebo lyžování, jak se ve dvojexpozici promítají na obrazovku startovní listiny se jmény závodníků i zpracované výsledky. Zobrazované informace mohou být snímány přímo z obrazovky počítače.

Pokusme se nyní postupně vytvořit program, který by uměl pracovat podobně. Jde o dosti složitý a rozsáhlý problém, proto jej budeme postupně rozvíjet v rámci několika kapitol.

Budeme zpracovávat výsledky běhů v lehké atletice, tj. předmětem hodnocení závodníka bude dosažený čas (což můžeme použít i v jiném sportu).

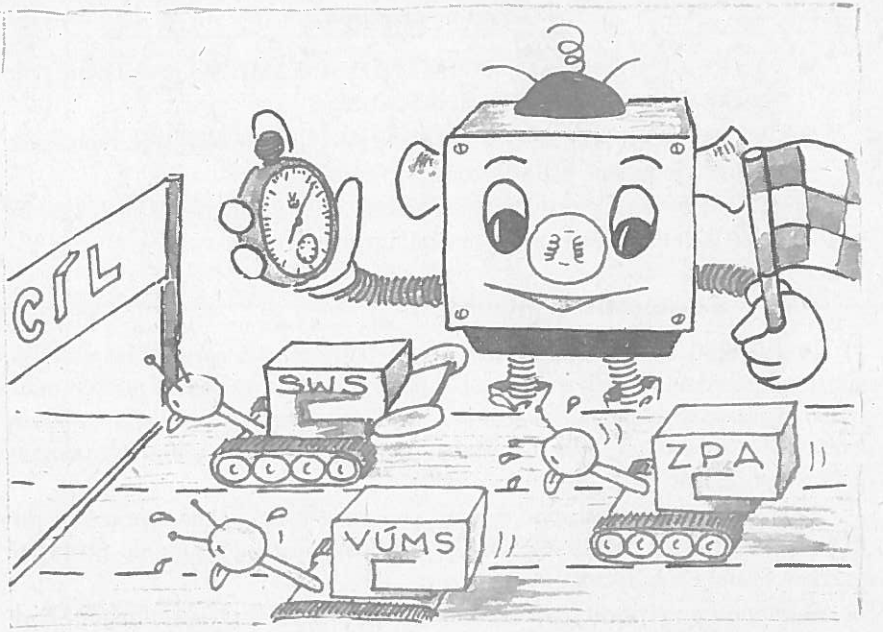
V paměti počítače budeme uchovávat tyto údaje pro každého startujícího závodníka:

- jméno, příjmení a státní příslušnost,
- startovní číslo,
- startovní čas,
- výsledný čas.

Při práci s programem budeme chtít prozatím provádět tyto činnosti:

- zadávání jména, příjmení a státní příslušnosti (tj. vytváření startovní listiny),
- výpis startovní listiny,
- zadávání času, kdy závodník odstartoval,
- zadávání času, kdy závodník dosáhl cíle,
- výpis výsledkové listiny.

Každá z těchto činností bude v programu řešena ne zcela jednoduchým algoritmem a mohla by být zpracovávána i samostatným programem. Všechny části však pracují nad společnými údaji o závodnících, proto je budeme realizovat v rámci programu jediného. Údaje, nad kterými bude program pracovat, budou tvořit údajovou bázi (databázi) informací o závodnících a jejich výkonech.



Využijeme-li známé metody návrhu programu shora dolů, mohl by náš program v prvním přiblížení mít tento tvar:

```
program SPORT (input, output);
type
  TCINNOSTI = (CHYBNYKOD,ZSTARTU,ZCILE,
    VYSLEDKY,ZJMENA,ZLISTINA,KONEC);
                          { typ kódů činností programu }
var
  DB        : DATABAZE; { báze údajů }
  CYKLUJ    : boolean; { proměnná určující dobu činnosti programu }

begin { SPORT }
  writeln('VYPOCET VYSLEDKU');
  writeln;
  POCATECNINASTAVENI(DB); { počáteční nastavení báze údajů }
  CYKLUJ:=true;  { příznak nastavit na režim práce }
```

```
repeat
  case URCENITYPU { určení typu činnosti } of
    ZSTARTU    : ZADSCASU (DB); { zadání start. času }
    ZCILE      : ZADCCILE (DB); { zadání času v cíli }
    VYSLEDKY   : VYPVYSLEDKU (DB); { výsledková listina }
    ZJMENA     : ZADJMENA (DB); { zadání jména a státu }
    ZLISTINA   : LISTINA (DB); { startovní listina }
    KONEC      : CYKLUJ := false; { konec programu }
  end; { case }
until not CYKLUJ;
writeln('Konec programu SPORT');
end. { SPORT }
```

Hlavní řídicí strukturou programu je cyklus repeat. Doba činnosti je řízena hodnotou logické proměnné CYKLUJ, jejíž počáteční hodnotou je *true*. Dokud má proměnná tuto hodnotu, je opakovaně prováděno určení typu činnosti programu funkcí URCENITYPU (s výsledkem typu TCINNOSTI) s následným příkazem case vybírajícím a provádějícím určenou činnost.

Většina konverzačních programů pracujících s více příkazy uživatele má tuto nebo velmi podobnou základní strukturu. Jde vždy o cyklus ukončovaný speciálním povelem (KONEC). Tělo cyklu tvoří nejčastěji příkaz case, který volí jednu z možných činností programu.

V našem programu funkce URCENITYPU pro určení typu činnosti získá konverzací s uživatelem informaci o tom, která činnost se bude v dalším kroku provádět. Její výsledek je výčtového typu, který se pro kódování omezeného počtu činností hodí.

```
type
  TCINNOSTI = (CHYBNYKOD, ZSTARTU, ZCILE,
    VYSLEDKY,ZJMENA,ZLISTINA,KONEC);
  { typ kódů činností programu }
```

Jednotlivé hodnoty popisují postupně chybně zadaný kód činnosti, zadávání času startu, zadávání času cíle, výpis výsledků, zadávání jména, příjmení a státní příslušnosti, výpis startovní listiny a ukončení programu.

Je-li zadanou činností KONEC, je proměnné CINNOST přiřazena hodnota *false* a program se ukončí. Ostatní dílčí činnosti budou zřejmě složitější a jejich realizace bude popsána dále v průběhu této kapitoly. Rovněž konkrétní podobu báze údajů informací o závodnících dosud neurčujeme.

V bázi údajů o závodnících (datový typ DATABAZE) chceme ukládat startovní čísla, jména, příjmení, státní příslušnost, časy startu a výsledné časy.

Údaje budou zadávány podle startovních čísel závodníků.

Bez popisu vnitřní struktury typu DATABAZE můžeme definovat operace pro práci s jejími údaji. Budou to operace pro zápis údajů do databáze a získání údajů z databáze:

- zápis jména, příjmení a státní příslušnosti,
- získání jména, příjmení a státní příslušnosti,
- zápis času startu,
- získání času startu,
- zápis výsledného času,
- získání výsledného času,
- získání maximálního startovního čísla uloženého v databázi.

Kromě poslední operace se budou zápisy a získání údajů provádět vždy pro závodníka s jistým startovním číslem, tj. půjde o procedury a funkce s hlavičkami

```
procedure ZAPISJMEN ( var BAZE:DATABAZE;
  STCISLO:TSTCISLA; JM, PR:RET10; ST: RET3);
procedure ZISKANIJMEN(var  BAZE:DATABAZE;
  STCISLO:TSTCISLA; var JM, PR:RET10; var ST: RET3);
procedure ZAPISCS ( var BAZE:DATABAZE;
  STCISLO:TSTCISLA; CAS:NEZAP);
function ZISKANICS ( var BAZE:DATABAZE;
  STCISLO:TSTCISLA):NEZAP;
procedure ZAPISVC ( var BAZE:DATABAZE;
  STCISLO:TSTCISLA; CAS:NEZAP);
function ZISKANIVC ( var BAZE:DATABAZE;
  STCISLO:TSTCISLA):NEZAP;
function ZISKANIMAXCISLA ( var BAZE: DATABAZE )
  : TSTCISLA;
```

Typ NEZAP vyjadřující čas v sekundách bude celočíselným typem celých nezáporných čísel. Pro 16bitové zobrazení celého čísla je třeba si uvědomit omezení maximální hodnoty na 32767 sekund, což je přibližně 9 hodin a 6 minut.

Pro praktické použití při závodech je přesnost v sekundách malá (obvykle se čas měří s přesností na setiny nebo tisíciny sekundy). Program je proto třeba chápat spíše jako ukázku postupu při řešení zadané problematiky.

```
type
  NEZAP = 0 .. maxint;
```

Datový typ startovního čísla TSTCISLA bude celočíselným intervalem od 1 do maximálního startovního čísla MAXSTCISLO.

```
const
  MAXSTCISLO = 100;
type
  TSTCISLA = 1 .. MAXSTCISLO;
```

Podle principu shora dolů odložíme vlastní realizaci těl těchto procedur a funkcí (a tím i definitivní popis datového typu DATABAZE) na pozdější dobu a pokusíme se nejprve realizovat těla procedur ZADSCASU, ZADCCILE, VYPVYSLEDKU, ZADJMENA a LISTINA.

Pro zadávání startovní listiny slouží procedura ZADJMENA. Příkazem procedury se uloží do databáze jméno, příjmení a státní příslušnost jednoho závodníka:

```
procedure ZADJMENA ( var BAZE:DATABAZE );
{ zadání obsahu startovní listiny }
var
  CISLO : TSTCISLA; { startovní číslo }
  JM,PR : RET10;    { jméno a příjmení }
  ST    : RET3;     { stát }
  I     : NEZAP;    { pro indexování }
begin
  writeln ('Zadej startovni cislo');
  readln  (CISLO);
  writeln ('Zadej jmeno, prijmeni a stat');
  writeln ('na zvlastni radky');
  I:=0;   { čtení jména po znacích }
  JM:='          ';
  while (not eoln) and (I<10) do
    begin
      I:=I+1;
      read(JM[I]);
    end;
  readln;
  I:=0;   { čtení příjmení po znacích }
  PR:='          ';
  while (not eoln) and (I<10) do
    begin
      I:=I+1;
```

```
        read(PR[I]);
      end;
    readln;
    I:=0;   { čtení názvu státu po znacích }
    ST:='    ';
    while (not eoln) and (I<3) do
      begin
        I:=I+1;
        read(ST[I]);
      end;
    readln;
    ZAPISJMEN(BAZE,CISLO,JM,PR,ST);
end; { ZADJMENA }
```

Tělo procedury je tvořeno třemi obdobnými částmi pro vstup jména, příjmení a státu. Každá z částí je tvořena cyklem while, který provádí čtení řetězce znaků (znak po znaku), dokud není dosaženo konce čteného řádku, což je indikováno hodnotou *true* standardní funkce *eoln* (end of line = konec řádku, kap. 9), nebo dokud není délka řetězce zcela zaplněna. Takto jsou postupně naplněny proměnné JM (jméno), PR (příjmení) a ST (stát). Nakonec jsou všechny tři hodnoty proměnných uloženy do báze k zadanému startovnímu číslu CISLO procedurou ZAPISJMEN.

Procedura ZADSCASU má vložit do databáze čas startu jednoho závodníka. Musí tedy konverzací s uživatelem získat startovní číslo a čas startu (pokud možno v hodinách, minutách a sekundách) a tyto údaje uložit do báze údajů. Může být realizována např. takto:

```
procedure ZADSCASU (var BAZE:DATABAZE);
{ zadávání startovního času }
var
  CISLO: TSTCISLA; { startovní číslo }

begin { ZADSCASU }
  writeln  ('Zadej startovni cislo');
  readln   (CISLO); { čtení startovního čísla }
  { zápis času startu do báze }
  ZAPISCS(BAZE,CISLO,CTENICASU);
end; { ZADSCASU }
```

Procedura ZADSCASU má lokální proměnnou CISLO a formální parametr BAZE. Obsahuje tři příkazy. První dva formou konverzace získají

startovní číslo závodníka zapsané jako celé číslo na jednom řádku. Třetí příkaz zapíše získané hodnoty do databáze.

Funkcí CTENICASU získáme údaje o čase ve formě

HH MM SS

kde HH jsou hodiny, MM minuty a SS sekundy, a provedeme pak převod tohoto vstupního údaje na sekundy. Převod se provádí podle vztahu

(hodiny * 60 + minuty) * 60 + sekundy.

Cílový čas bude zřejmě nutné získávat stejným způsobem. Funkce má tvar:

```
function CTENICASU :NEZAP;
{ čtení a převod času na sekundy }
var
  HH  : NEZAP;        { hodiny }
  MM,SS: 0 .. 59;     { minuty a sekundy }
begin { CTENICASU }
  writeln  ('Zadej cas ve forme HH MM SS');
  readln   (HH,MM,SS);
  CTENICASU := (HH * 60 + MM) * 60 + SS;
end; { CTENICASU }
```

Před čtením každé z celočíselných hodnot HH, MM a SS jsou vždy přeskočeny úvodní mezery a konce řádků. Hodnoty lze tedy oddělovat i více mezerami, resp. je psát na zvláštní řádky.

Procedura ZADCCILE pro zadávání cílového času se bude od procedury ZADSCASU lišit v tom, že do báze údajů se ukládá nikoliv čas průchodu cílem, který napíše uživatel na terminálu, nýbrž čas výsledný (tj. čas průchodu cílem – čas startu). Proto je v těle procedury provedeno odečtení času startu od času v cíli a rozdíl uložen jako výsledný čas do databáze.

```
procedure ZADCCILE (var BAZE:DATABAZE);
{ zadávání cílového času }
var
  CISLO: TSTCISLA; { startovní číslo }
begin { ZADCCILE }
  writeln('Zadej startovni cislo');
  readln(CISLO);   { čtení startovního čísla }
  { zápis času cíle do báze }
  ZAPISVC(BAZE,CISLO,
    CTENICASU-ZISKANICS(BAZE,CISLO));
end; { ZADCCILE }
```

První dva příkazy jsou obdobné proceduře ZADSCASU. Čas dosažení cíle se získá od uživatele funkcí CTENICASU. Následuje získání času startu daného startovního čísla z databáze funkcí ZISKANICS(BAZE,CISLO) a nakonec zápis výsledného času, tj. rozdílu času v cíli a času startu.

Můžeme si povšimnout užití celočíselného výrazu

CTENICASU–ZISKANICS(BAZE,CISLO)

který je využit jako skutečný parametr volaný hodnotou v proceduře ZAPISVC.

Zřejmě nejsložitějším úkolem bude výpis výsledkové listiny. Budeme chtít, aby se do výsledkové listiny zahrnul jen *určitý počet* nejlepších závodníků (např. 50).

Musíme nejprve vybrat ty závodníky, kteří startovali, a z nich vyloučit ty, kteří nedosáhli cíle. Vybrané závodníky je pak nutné vzestupně seřadit podle časů dosažených v cíli. Setříděné údaje o závodnících budeme vypisovat na obrazovku ve tvaru

pořadí. startovní číslo výsledný čas

přičemž výsledný čas chceme zobrazit ve tvaru

HH:MM:SS

a nikoliv ve vnitřním tvaru udávajícím čas jako počet sekund.

Výpis na obrazovku budeme chtít zastavit vždy po zaplnění obrazovky (stránky) textem a dát tak uživateli možnost si stránku přečíst.

Proceduru VYPVYSLEDKU lze realizovat např. takto:

```
procedure VYPVYSLEDKU (var BAZE: DATABAZE);
{ výpis výsledkové listiny }
const
  MAXDELKAVYSLEDKU = 50;
  { maximální počet startujících ve výsledcích }
type
  TYPDELKY = 0 .. MAXDELKAVYSLEDKU;
  { typ počtu vyhodnocovaných výsledků }
var
  TRIDICIPOLE : array [TYPDELKY] of
    record
      SCISLO  : TSTCISLA;   { startovní číslo }
      VCAS    : NEZAP ;     { výsledný čas }
    end;
  POCET : TYPDELKY;         { skutečný počet startujících }
  STCISLO : TSTCISLA;       { právě čtené startovní číslo }
```

```
  RADKU,                { počet vypsaných řádků }
  VYSLCAS,              { výsledný čas }
  I,J : NEZAP;          { pracovní proměnné }
  JM,PR : RET10;        { jméno a příjmení }
  ST : RET3;            { státní příslušnost }
  VYMENA : boolean;     { příznak výměny při třídění }

begin { VYPVYSLEDKU }

  { začátek první části procedury }
  POCET := 0; { skutečný počet závodníků vynulovat }
  STCISLO := 1; { začínáme číst výsledky od čísla 1 }
  while (STCISLO <= ZISKANIMAXCISLA(BAZE)) and
    (POCET < MAXDELKAVYSLEDKU) do
    begin { přesun z báze údajů do třídicího pole }
      VYSLCAS:= ZISKANIVC(BAZE,STCISLO);
      { výsledný čas z báze }
      if VYSLCAS > 0 then
        begin { startoval }
          POCET := POCET + 1; { skutečný počet zvýšit }
          { startovní číslo }
          TRIDICIPOLE[POCET].SCISLO:=STCISLO;
          TRIDICIPOLE[POCET].VCAS:=VYSLCAS;
          { výsledný čas }
        end; { startoval }
      STCISLO := STCISLO + 1;
    end; { přesun z báze údajů do třídicího pole }

  { začátek druhé části procedury }
  { třídění }
  I := POCET;
  repeat
    VYMENA := false; { zatím se neuskutečnila výměna }
    for J:=1 to I-1 do
      if TRIDICIPOLE[J+1].VCAS < TRIDICIPOLE[J].VCAS
        then
        begin { výměna dvou složek pole }
          TRIDICIPOLE[0]   := TRIDICIPOLE[J+1];
          TRIDICIPOLE[J+1] := TRIDICIPOLE[J];
```

```
            TRIDICIPOLE[J]      := TRIDICIPOLE[0];
            VYMENA := true; { uskutečnila se výměna }
          end; { výměna dvou složek pole }
    I := I - 1;
  until (I < 2) or (not VYMENA);

  { začátek třetí části procedury }
  { dočtení zbývajících startujících }
  while STCISLO < = ZISKANIMAXCISLA (BAZE) do
    begin { přes všechny zbývající startující }
      VYSLCAS := ZISKANIVC (BAZE, STCISLO);
      if (VYSLCAS > 0) and
        (VYSLCAS < TRIDICIPOLE[MAXDELKAVYSLEDKU]
         VCAS) then
          begin { bude se zařazovat }
            J := MAXDELKAVYSLEDKU - 1;
            while (VYSLCAS < TRIDICIPOLE[J].VCAS)
              and(J > 0) do
              begin { posun složky pole }
                TRIDICIPOLE[J + 1] := TRIDICIPOLE[J];
                J := J-1;
              end; { posun složky pole }
            TRIDICIPOLE[J + 1].SCISLO := STCISLO;
            TRIDICIPOLE[J + 1].VCAS := VYSLCAS;
          end; { bude se zařazovat }
      STCISLO := STCISLO + 1;
    end; { přes všechny zbývající startující }

  { začátek čtvrté části procedury }
  { výpis na obrazovku }
  RADKU := VELIKOSTOBRAZOVKY + 2; { počáteční nastavení }
  for I: = 1 to POCET do
    begin { výpis výsledků }
      if RADKU > = VELIKOSTOBRAZOVKY then
        begin { zastavit - obrazovka zaplněna }
          if RADKU < VELIKOSTOBRAZOVKY + 2 then
            begin { 2. a další průchody }
              readln; { čekání na odpověď operátora }
              RADKU := 0;
```

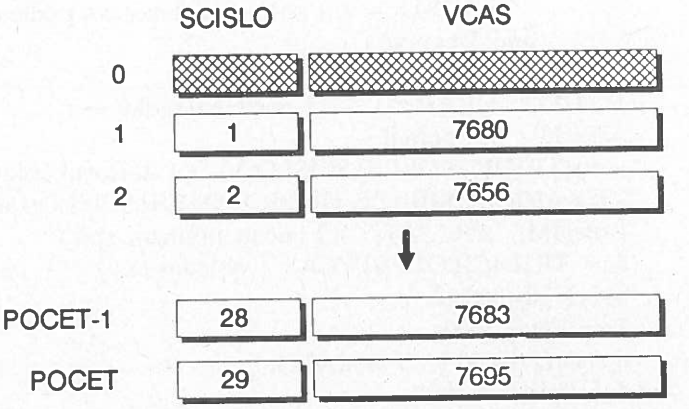
```
            end { 2. a další průchody }
          else
            begin { poprvé }
              writeln('Vysledky');
              writeln;
              RADKU := 2; { počet řádků nastavit podle záhlaví }
            end; { poprvé }
        end; { zastavit – obrazovka zaplněna }
      RADKU := RADKU +1; { počítadlo řádků + 1 }
      write(I:2,'. '); { pořadí }
      write(TRIDICIPOLE[I].SCISLO:3,' '); { startovní číslo }
      ZISKANIJMEN(BAZE,TRIDICIPOLE[I].SCISLO,JM,PR,ST);
      write(JM,' ',PR,' ',ST,' '); { jméno, příjmení, stát }
      J := TRIDICIPOLE[I].VCAS; { výsledný čas }
      write(J div 3600:2,':');
      J := J mod 3600;
      writeln(J div 60:2,':',J mod 60:2);
    end; { výpis výsledků }
  readln; { čekání na povel operátora }
  writeln('Konec vysledkove listiny');
end; { VYPVYSLEDKU }
```

Při realizaci procedury budeme předpokládat, že před započetím práce s databází budou výsledné časy všech závodníků vynulovány. Při výběru výsledných časů z databáze budeme brát proto v úvahu pouze ty, jejichž hodnota výsledného času je větší než nula. Abychom mohli provést setřídění výsledků podle dosažených časů vzestupně, máme deklarováno pole TRIDICIPOLE (obr. 8.1). Každá složka třídicího pole je záznamem

```
var
  TRIDICIPOLE : array [TYPDELKY] of
                record
                  SCISLO   : TSTCISLA; { startovní číslo }
                  VCAS     : NEZAP ; { výsledný čas }
                end;
```

a obsahuje startovní číslo SCISLO a výsledný čas jednoho závodníka VCAS. Počet složek pole je o jedničku větší, než je maximální možný počet vyhodnocovaných závodníků MAXDELKAVYSLEDKU. Položka TRIDICIPOLE[0] bude pracovní proměnnou pro výměnu položek pole při řazení.

První část těla procedury (v těle jsou začátky jednotlivých částí procedury označeny) provádí naplnění třídicího pole TRIDICIPOLE. Proměnná POCET obsahuje počet platných položek třídicího pole (jeho zaplněnost), proto je na začátku nulová. Proměnná STCISLO udává startovní číslo zpracová-



Obr. 8.1 Třídicí pole

vaného závodníka (postupně od 1). Naplnění třídicího pole se provádí cyklem while. Cyklus je ukončen, jsou-li údaje o všech startujících již přeneseny (funkce ZISKANIMAXCISLA udává maximální použité startovní číslo v databázi)

STCISLO > ZISKANIMAXCISLA(BAZE)

nebo je-li třídicí pole již plné

POCET > =MAXDELKAVYSLEDKU

Jak jsme již řekli, z databáze jsou přenášeny pouze výsledné časy větší než nula. K výběru výsledného času z databáze slouží volání procedury ZISKANIVC(BAZE,STCISLO).

Druhá část těla procedury provádí třídění položek v třídicím poli vzestupně podle výsledných časů. Je užito bublinového třídění (cvičení kap. 7). Znovu si připomeňme, že záměna hodnot dvou proměnných je možná jen s využitím třetí pomocné proměnné. Jako pomocná proměnná pro výměny slouží položka třídicího pole s indexem 0. Proměnná VYMENA určuje, zda při průběhu těla cyklu došlo alespoň k jedné výměně. Pokud k ní nedošlo, je pole již setříděno a není nutné pokračovat. Po ukončení druhé části jsou v třídicím poli na indexech odpovídajících pořadí startovní čísla a výsledné časy závodníků.

Třetí část procedury se uplatní pouze tehdy, pokud dosud všichni závodníci nebyli přeneseni z databáze do třídicího pole (STCISLO < = ZISKANIMAXCISLA(BAZE)). V takovém případě jsou cyklem while přenášeny údaje o dalších závodnících a zařazovány do třídicího pole. Pokud závodník startoval

VYSLCAS > 0

a jeho čas je lepší než čas dosud posledního ve výsledkové listině

VYSLCAS < TRIDICIPOLE[MAXDELKAVYSLEDKU].VCAS

jsou všechny položky třídicího pole, které mají horší výsledný čas posunuty o jednu položku ke konci pole a na uvolněné místo jsou uloženy údaje o novém závodníkovi. Tím dosáhneme, že i po přečtení zbývajících údajů zůstane pole setříděno.

Čtvrtá část procedury realizuje vlastní výpis výsledků na obrazovku. Chceme-li, aby se výpis vždy po zaplnění obrazovky zastavil, musíme vypsaný počet řádků počítat. K tomu slouží celočíselná proměnná RADKU, která je vždy před výpisem jedné položky testována. Je-li její hodnota větší, než konstanta VELIKOSTOBRAZOVKY (udávající možný počet řádků na obrazovce), je příkazem *readln* výpis zastaven a očekává se povel operátora (zadáním prázdného řádku).

Jelikož jsou časy uchovávány v sekundách, je nutné je před výstupem opět převést na požadovaný tvar. Jde vlastně o převod hodnoty do dané číselné soustavy (tentokrát do soustavy se základem 60). Hodiny získáme celočíselným dělením hodnotou 60^2 a ze zbytku dělením 60^1 získáme minuty. Zbytek jsou sekundy.

Procedura pro výpis startovní listiny LISTINA bude z hlediska tvaru výpisu podobná proceduře pro výpis výsledků.

```
procedure LISTINA (var BAZE: DATABAZE);
{ výpis startovní listiny }
var
  STCISLO : TSTCISLA;   { právě čtené startovní číslo }
  RADKU   : NEZAP;      { počet vypsaných řádků }
  JM,PR   : RET10;      { jméno a příjmení }
  ST      : RET3;       { státní příslušnost }
begin { LISTINA }
  RADKU := VELIKOSTOBRAZOVKY +2; { počáteční nastavení }
  for STCISLO:=1 to ZISKANIMAXCISLA(BAZE) do
    begin { výpis }
      ZISKANIJMEN(BAZE,STCISLO,JM,PR,ST);
```

```
            if ST < > '???' then
              begin { toto startovní číslo je v databázi }
                if RADKU > = VELIKOSTOBRAZOVKY then
                  begin { zastavit – obrazovka zaplněna }
                    if RADKU < VELIKOSTOBRAZOVKY + 2 then
                      begin { 2. a další průchody }
                        { čekání na odpověď operátora }
                        readln;
                        RADKU : = 0;
                      end  { 2. a další průchody }
                    else
                      begin { poprvé }
                        writeln('Startovni listina');
                        writeln;
                        { počet řádků nastavit podle záhlaví }
                        RADKU : = 2;
                      end; { poprvé }
                  end; { zastavit – obrazovka zaplněna }
                RADKU : = RADKU + 1; { počítadlo řádků + 1 }
                writeln(STCISLO:3,'. ',JM,' ',PR,' ',ST);
                    { startovní číslo, jméno, příjmení, státní příslušnost }
              end; { toto startovní číslo je v databázi }
          end; { výpis }
        writeln;
        writeln('Konec startovni listiny');
      end; { LISTINA }
```

Z báze údajů o závodnících se postupně vybírají údaje podle startovních čísel. Nechceme však vypisovat neobsazená startovní čísla, proto je třeba rozpoznat, kteří závodníci jsou již k závodu přihlášeni. Procedura ZISKANI-JMEN bude při zadání startovního čísla závodníka, který nebyl dosud do báze údajů vložen, vracet jako státní příslušnost závodníka pomocnou hodnotu '???'. Podle této hodnoty pak vyloučíme neobsazená startovní čísla.

Proto, pokud po získání údajů pro dané startovní číslo nemá státní příslušnost (proměnná ST) hodnotu '???', jde o závodníka, který je v databázi uložen, a provede se výpis jednoho řádku listiny. Určování počtu vypsaných řádků a zastavování výpisu po zaplnění stránky je stejné jako u procedury VYPIS-VYSLEDKU.

Funkce URCENITYPU čte znaky určující typ žádané činnosti programu.

```
function URCENITYPU: TCINNOSTI;
{ určení typu činnosti, která se bude provádět }
var
  ZNAK : char;   { pracovní proměnné }
  DALSICINNOST : TCINNOSTI;
begin { URCENITYPU }
  repeat
    writeln('Zadej kod cinnosti');
    readln(ZNAK);
    case ZNAK of
      's','S'   : DALSICINNOST := ZSTARTU;
      'c','C'   : DALSICINNOST := ZCILE;
      'v','V'   : DALSICINNOST := VYSLEDKY;
      'j','J'   : DALSICINNOST := ZJMENA;
      'l','L'   : DALSICINNOST := ZLISTINA;
      'k','K'   : DALSICINNOST := KONEC;
      else
        begin { chybný kód }
          writeln ('Chybny kod cinnosti');
          writeln;
          writeln ('S ... cas startu');
          writeln ('C ... cas v cili');
          writeln ('V ... vysledkova listina');
          writeln ('J ... zadani jmena, prijmeni a statu');
          writeln ('L ... vypis startovni listiny');
          writeln ('K ... konec programu');
          writeln;
          DALSICINNOST := CHYBNYKOD;
        end; { chybný kód }
    end; { case }
  until DALSICINNOST <> CHYBNYKOD;
  URCENITYPU := DALSICINNOST;
end; { URCENITYPU }
```

Příkazem case jsou znaky převedeny na hodnoty výčtového typu TCINNOSTI. Kódy činností se mohou zadávat jak malými, tak velkými písmeny (varianta příkazu case je označena vždy dvěma znakovými konstantami). Není-li zadáno správné písmeno (varianta else), je vypsána nápověda obsahující popis všech povolených kódů a uživatel může zadat správný znak.

Program máme tedy téměř hotov, až na *malou* výjimku. Z datového typu DATABAZE známe dosud pouze jméno a hlavičky procedur a funkcí, které s ním bezprostředně pracují. To nám umožnilo napsat zbytek programu.

Dosud není určeno, jak vlastně bude datový typ DATABAZE popsán (není známa jeho vnitřní stavba). Nemohou být tudíž deklarována ani těla těch procedur a funkcí, které přímo pracují s jeho hodnotami. Dále již deklaraci typu DATABAZE odkládat nemůžeme.

Vzhledem k tomu, že údaje jsou ukládány podle startovních čísel, bude zřejmě vhodné organizovat bázi jako jako pole indexované startovními čísly

```
type
  DATABAZE = array [TSTCISLA] of ZAVODNIK;
```

Ve složce pole typu ZAVODNIK je uloženo jméno, příjmení, státní příslušnost, čas startu a čas cíle. Jelikož jde o hodnoty různých datových typů, musíme použít záznamu. Výhodná je datová struktura ZAVODNIK definovaná na počátku této kapitoly.

Typ báze údajů bude pak deklarován takto:

```
type
  NEZAP     = 0 .. maxint;          { nezáporné celé číslo }
  TSTCISLA  = 0..MAXSTCISLO; { typ startovního čísla }
  RET3      = packed array [1.. 3] of char; { řetězec délky 3 }
  RET10     = packed array [1..10] of char; { řetězec délky 10 }

  ZAVODNIK  =
    record     { typ záznamu o závodníkovi }
      JMENO,                          { jméno }
      PRIJMENI: RET10;                { příjmení }
      STAT  : RET3;                   { státní příslušnost }
      CASSTARTU,                      { čas startu }
      VYSLEDNYCAS : NEZAP; { výsledný čas }
    end; { ZAVODNIK }
  DATABAZE  =
    record     { typ báze údajů }
      MAXSTCIS  : TSTCISLA; { maximální uvedené startovní číslo }
      ZAVODNICI : array [TSTCISLA] of ZAVODNIK;
                        { údaje o závodnících }
    end; { DATABAZE }
```

V popisu typu DATABAZE je kromě pole ZAVODNICI obsahujícího údaje

o závodnících obsažena i složka MAXSTCIS obsahující maximální použité startovní číslo.

Nyní je nutné deklarovat ovládací procedury a funkce pro styk s databází ZAPISJMEN, ZISKANIJMEN, ZAPISCS, ZISKANICS, ZAPISVC, ZISKANIVC a ZISKANIMAXCISLA.

Procedury pro zápis a získání jména, příjmení a státní příslušnosti budou:

```
procedure ZAPISJMEN ( var BAZE:DATABAZE;
  STCISLO:TSTCISLA; JM, PR:RET10; ST: RET3);
{ zápis jména JM, příjmení PR a státu ST startovnímu číslu STCISLO }
begin { ZAPISJMEN }
  if BAZE.MAXSTCIS < STCISLO then { další startovní číslo }
    BAZE.MAXSTCIS := STCISLO;
  BAZE.ZAVODNICI[STCISLO].JMENO      := JM;
  BAZE.ZAVODNICI[STCISLO].PRIJMENI   := PR;
  BAZE.ZAVODNICI[STCISLO].STAT       := ST;
end; { ZAPISJMEN }
procedure ZISKANIJMEN (var BAZE:DATABAZE;
  STCISLO:TSTCISLA; var JM, PR:RET10; var ST: RET3);
{ získání jména JM, příjmení PR a státu ST startovního čísla STCISLO }
begin { ZISKANIJMEN }
  JM  := BAZE.ZAVODNICI[STCISLO].JMENO;
  PR  := BAZE.ZAVODNICI[STCISLO].PRIJMENI;
  ST  := BAZE.ZAVODNICI[STCISLO].STAT;
end; { ZISKANIJMEN }
```

Při vyvářeni startovní listiny procedurou ZAPISJMEN se upravuje i složka MAXSTCIS, která obsahuje maximální startovní číslo uložené v bázi údajů.

K efektivnosti procedur ZAPISJMEN a ZISKANIJMEN můžeme mít oprávněné námitky. Opakovaný zápis

BAZE . ZAVODNICI[STCISLO]

je nepohodlný a navíc vícenásobné vyhodnocování indexu může zpomalovat výpočet. Protože obdobné bloky příkazů pracující s jedinou proměnnou typu záznam a různými jejími složkami jsou časté, umožňuje Pascal společnou proměnnou typu záznam *vytknout před závorku* užitím *příkazu with* (with = předložka s).

Příkaz with má tvar

with *proměnná typu záznam* **do**
  příkaz

Uvnitř *příkazu* následujícího za **do** je možné se na složky *proměnné typu záznam* uvedené za **with** odkazovat nikoli zápisem

proměnná typu záznam . jméno složky

ale pouze zápisem identifikátoru

jméno složky

Tělo procedury ZISKANIJMEN lze pak zapsat takto

```
begin { ZISKANIJMEN }
  with BAZE.ZAVODNICI[STCISLO] do
    begin { získání jména, příjmení a státní příslušnosti }
      JM  :=  JMENO;
      PR  :=  PRIJMENI;
      ST  :=  STAT;
    end; { with ZAVODNICI }
end; { ZISKANIJMEN }
```

V příkazu with vystupují jména složek samostatně. Mohlo by proto dojít ke kolizi jmen složek a ostatních identifikátorů v programu. V tom případě platí pravidlo, že jméno samostatné složky má vždy přednost (kolidující jméno je vždy chápáno jako jméno složky vztažené k nejblíže nadřazenému příkazu with).

Je-li možné vnořovat do sebe datové typy záznam, musí být možné vnořovat do sebe i příkazy with. I zde existuje možnost zkrácení zápisu. Místo

with *proměnná*$_1$ **do**
  with *proměnná*$_2$ **do**
    :
      with *proměnná*$_n$ **do** *příkaz*

je možné napsat zkráceně

with *proměnná*$_1$, *proměnná*$_2$,..., *proměnná*$_n$ **do** *příkaz*

Tělo procedury ZAPISJMEN je tedy možné zapsat i takto:

```
begin { ZAPISJMEN }
  with BAZE,ZAVODNICI[STCISLO] do
    begin { zápis jména, příjmení a státní příslušnosti }
    if MAXSTCIS < STCISLO then { další startovní číslo }
      MAXSTCIS :=  STCISLO;
      JMENO    :=  JM;
      PRIJMENI :=  PR;
```

```
        STAT        := ST;
      end; { with ZAVODNICI }
    end; { ZAPISJMEN }
```

Procedura a funkce pro zápis a získání času startu pro zadané startovní číslo mají pak tvar:

```
  procedure ZAPISCS ( var BAZE:DATABAZE;
    STCISLO:TSTCISLA; CAS:NEZAP);
  { zápis času startu CAS startovnímu číslu STCISLO }
  begin { ZAPISCS }
    BAZE.ZAVODNICI[STCISLO].CASSTARTU:=CAS;
  end; { ZAPISCS }

  function ZISKANICS ( var BAZE:DATABAZE;
    STCISLO:TSTCISLA) :NEZAP;
  { získání času startu startovního čísla STCISLO }
  begin { ZISKANICS }
    ZISKANICS:=BAZE.ZAVODNICI[STCISLO].CASSTARTU;
  end; { ZISKANICS }
```

Procedura a funkce pro zápis a získání výsledného času ZAPISVC a ZISKANIVC se liší od ZAPISCS a ZISKANICS pouze v identifikátoru složky záznamu (VYSLEDNYCAS namísto CASSTARTU).

Funkce ZISKANIMAXCISLA je velmi jednoduchá.

```
  function ZISKANIMAXCISLA ( var BAZE: DATABAZE )
    : TSTCISLA;
  { získání maximálního startovního čísla }
  begin { ZISKANIMAXCISLA }
    ZISKANIMAXCISLA := BAZE.MAXSTCIS;
  end; { ZISKANIMAXCISLA }
```

Provede pouze přenos složky BAZE.MAXSTCIS, který předá jako výsledek funkce.

Nakonec se zmíníme ještě o proceduře POCATECNINASTAVENI. Tato procedura dosadí do proměnné typu DATABAZE hodnoty, které jsme na začátku činnosti programu v bázi předpokládali.

```
  procedure POCATECNINASTAVENI (var BAZE:DATABAZE);
  { počáteční nulování výsledných časů }
  var
    I:TSTCISLA;
```

```
begin { POCATECNINASTAVENI }
  for I: = 1 to MAXSTCISLO do
    begin
      ZAPISVC(BAZE,I,0);  { vynulování startovního čísla I }
      ZAPISJMEN(BAZE,I,'??????????','??????????','???');
    end; { for }
  BAZE.MAXSTCIS : = 0;
end; { POCATECNINASTAVENI }
```

Jde o dosazení nuly do všech výsledných časů. Toho se využilo v proceduře VYPISVYSLEDKU pro odlišení závodníků, kteří nestartovali. Dále jde o dosazení otazníků do všech jmen, příjmení a státních příslušností (tuto hodnotu jsme využívali pouze u státní příslušnosti v proceduře LISTINA pro vyloučení nepoužitých startovních čísel). Nakonec se dosadí i první hodnota maximálního použitého čísla (na začátku bude 0).

Kompletní program SPORT je označen 8.1.

```
program SPORT (input, output);
{ Program slouží pro vyhodnocování výsledků sportovních soutěží tam,
kde měřítkem výkonu je čas (zaokrouhlený na sekundy). Pro 16bitové
zobrazení celého čísla je maximální možný zadávaný čas 32767 sekund, tj.
asi 546 minut, tj. 9 hodin a 6 minut.

Vstupy:
  Na výzvu
    Zadej kod cinnosti

  je možno odpovědět těmito znaky
    S ... zadávání startovního času
    C ... zadávání cílového času
    V ... výpis výsledků
    J ... zadávání jména, příjmení a státu
    L ... výpis startovní listiny
    K ... konec programu

    Všechny časy se zadávají ve formě HH MM SS, kde HH jsou hodiny,
    MM jsou minuty a SS jsou sekundy.
Výstup: Při zadání kódu činnosti V se vypíše výsledková listina setříděná
    podle dosažených časů ve tvaru
    XX.  YY   jméno příjmení stát HH:MM:SS
  kde XX je pořadí, YY startovní číslo a HH:MM:SS je dosažený čas
  (hodiny:minuty:sekundy).
```

```
   Při zadání kódu činnosti L se vypíše startovní listina setříděná
   podle startovních čísel ve tvaru
      YY. jméno příjmení stát
   kde YY je startovní číslo
}
const
   MAXSTCISLO       = 100;        { maximální startovní číslo }
   VELIKOSTOBRAZOVKY = 24; { počet řádků na obrazovce }

type
   NEZAP       = 0 .. maxint;          { nezáporné celé číslo }
   TSTCISLA    = 0..MAXSTCISLO;{ typ startovního čísla }
   RET3        = packed array [1.. 3] of char; { řetězec délky 3 }
   RET10       = packed array [1..10] of char; { řetězec délky 10 }
   ZAVODNIK    =
      record      { typ záznamu o závodníkovi }
         JMENO,                        { jméno }
         PRIJMENI: RET10;              { příjmení }
         STAT  : RET3;                 { státní příslušnost }
         CASSTARTU,                    { čas startu }
         VYSLEDNYCAS : NEZAP; { výsledný čas }
      end; { ZAVODNIK }
   DATABAZE    =
      record      { typ báze údajů }
         MAXSTCIS  : TSTCISLA; { maximální uvedené startovní číslo }
         ZAVODNICI : array [TSTCISLA] of ZAVODNIK;
                                { údaje o závodnících }
      end; { DATABAZE }
   TCINNOSTI = (CHYBNYKOD,ZSTARTU,ZCILE,VYSLEDKY,
      ZJMENA,ZLISTINA,KONEC); { typ kódů činností programu }

var
   DB        : DATABAZE;     { báze údajů }
   CYKLUJ : boolean;     { proměnná určující dobu činnosti programu }

{ procedury pro práci s bází údajů }

procedure ZAPISCS ( var BAZE:DATABAZE;STCISLO:TSTCISLA;
                    CAS:NEZAP);
{ zápis času startu CAS startovnímu číslu STCISLO }
begin { ZAPISCS }
   BAZE.ZAVODNICI[STCISLO].CASSTARTU:=CAS;
end; { ZAPISCS }
```

```
function ZISKANICS ( var BAZE:DATABAZE;STCISLO:TSTCISLA)
                    :NEZAP;
{ získání času startu startovního čísla STCISLO }
begin { ZISKANICS }
  ZISKANICS: = BAZE.ZAVODNICI[STCISLO].CASSTARTU;
end; { ZISKANICS }

procedure ZAPISVC ( var BAZE:DATABAZE;STCISLO:TSTCISLA;
                    CAS:NEZAP);
{ zápis výsledného času CAS startovnímu číslu STCISLO }
begin { ZAPISVC }
  if BAZE.MAXSTCIS < STCISLO then { další startovní číslo }
    BAZE.MAXSTCIS : = STCISLO;
  BAZE.ZAVODNICI[STCISLO].VYSLEDNYCAS: = CAS;
end; { ZAPISVC }

function ZISKANIVC ( var BAZE:DATABAZE;STCISLO:TSTCISLA)
                    :NEZAP;
{ získání výsledného času startovního čísla STCISLO }
begin { ZISKANIVC }
  ZISKANIVC: = BAZE.ZAVODNICI[STCISLO].VYSLEDNYCAS;
end; { ZISKANIVC }

procedure ZAPISJMEN(var BAZE:DATABAZE;STCISLO:TSTCISLA;
                    JM, PR:RET10; ST: RET3);
{ zápis jména JM, příjmení PR a státu ST startovnímu číslu STCISLO }
begin { ZAPISJMEN }
  with BAZE,ZAVODNICI[STCISLO] do
    begin { zápis jména, příjmení a státní příslušnosti }
    if MAXSTCIS < STCISLO then { další startovní číslo }
      MAXSTCIS : = STCISLO;
      JMENO    : = JM;
      PRIJMENI : = PR;
      STAT     : = ST;
    end; { with ZAVODNICI }
end; { ZAPISJMEN }

procedure ZISKANIJMEN ( var BAZE:DATABAZE;
                        STCISLO:TSTCISLA;
                        var JM, PR:RET10; var ST: RET3);
{ získání jména JM, příjmení PR a státu ST startovního čísla STCISLO }
begin { ZISKANIJMEN }
  with BAZE.ZAVODNICI[STCISLO] do
```

```
      begin { získání jména příjmení a státní příslušnosti }
        JM  := JMENO ;
        PR  := PRIJMENI;
        ST  := STAT ;
      end; { with ZAVODNICI }
end; { ZISKANIJMEN }

function ZISKANIMAXCISLA(var BAZE: DATABAZE ): TSTCISLA;
{ získání maximálního startovního čísla }
begin { ZISKANIMAXCISLA }
  ZISKANIMAXCISLA := BAZE.MAXSTCIS;
end; { ZISKANIMAXCISLA }

procedure POCATECNINASTAVENI (var BAZE:DATABAZE);
{ počáteční nulování výsledných časů }
var
  I:TSTCISLA;
begin { POCATECNINASTAVENI }
  for I: = 1 to MAXSTCISLO do
    begin
      ZAPISVC(BAZE,I,0);  { vynulování startovního čísla I }
      ZAPISJMEN(BAZE,I,'??????????','??????????','???');
    end; { for }
  BAZE.MAXSTCIS : = 0;
end; { POCATECNINASTAVENI }

{ ostatní procedury }

function CTENICASU :NEZAP;
{ čtení a převod času na sekundy }
var
  HH        : NEZAP; { hodiny }
  MM,SS     : 0 .. 59;   { minuty a sekundy }

begin { CTENICASU }
  writeln('Zadej cas ve forme HH MM SS');
  readln(HH,MM,SS);
  CTENICASU : = (HH * 60 + MM) * 60 + SS;
end; { CTENICASU }

procedure ZADSCASU (var BAZE:DATABAZE);
{ zadávání startovního času }
```

```
var
   CISLO: TSTCISLA; { startovní číslo }
begin { ZADSCASU }
   writeln('Zadej startovni cislo');
   readln(CISLO);    { čtení startovního čísla }
   { zápis času startu do báze }
   ZAPISCS(BAZE,CISLO,CTENICASU);
end; { ZADSCASU }

procedure ZADCCILE (var BAZE:DATABAZE);
{ zadávání cílového času }
var
   CISLO: TSTCISLA; { startovní číslo }
begin { ZADCCILE }
   writeln('Zadej startovni cislo');
   readln(CISLO);    { čtení startovního čísla }
   { zápis času cíle do báze }
   ZAPISVC(BAZE,CISLO,
      CTENICASU-ZISKANICS(BAZE,CISLO));
end; { ZADCCILE }

procedure VYPVYSLEDKU (var BAZE: DATABAZE);
{ výpis výsledkové listiny }
const
   MAXDELKAVYSLEDKU = 50;
        { maximální počet startujících ve výsledcích }
type
   TYPDELKY = 0 .. MAXDELKAVYSLEDKU;
        { typ počtu vyhodnocovaných výsledků }
var
   TRIDICIPOLE : array [TYPDELKY] of
                 record
                    SCISLO : TSTCISLA; { startovní číslo }
                    VCAS   : NEZAP  ; { výsledný čas }
                 end;
   POCET      : TYPDELKY; { skutečný počet startujících }
   STCISLO : TSTCISLA;    { právě čtené startovní číslo }
   RADKU,                 { počet vypsaných řádků }
   VYSLCAS,               { výsledný čas }
   I,J        : NEZAP;    { pracovní proměnné }
   JM,PR      : RET10;    { jméno a příjmení }
   ST         : RET3;     { státní příslušnost }
   VYMENA     : boolean;  { příznak výměny při třídění }
```

```
begin { VYPVYSLEDKU }
  POCET := 0;  { skutečný počet závodníků vynulovat }
  STCISLO := 1; { začínáme číst výsledky od čísla 1 }
  while (STCISLO <= ZISKANIMAXCISLA(BAZE)) and
        (POCET < MAXDELKAVYSLEDKU) do
    begin { přesun z báze údajů do třídicího pole }
      VYSLCAS:=ZISKANIVC(BAZE,STCISLO);
      { výsledný čas z báze }
      if VYSLCAS > 0 then
        begin { startoval }
          POCET := POCET +1; { skutečný počet zvýšit }
          { startovní číslo }
          TRIDICIPOLE[POCET].SCISLO:=STCISLO;
          TRIDICIPOLE[POCET].VCAS:=VYSLCAS;
          { výsledný čas }
        end; { startoval }
      STCISLO := STCISLO + 1;
    end; { přesun z báze údajů do třídicího pole }
  { třídění }
  I := POCET;
  repeat
    VYMENA := false; { zatím se neuskutečnila výměna }
    for J:=1 to I-1 do
      if TRIDICIPOLE[J+1].VCAS < TRIDICIPOLE[J].VCAS then
        begin { výměna dvou složek pole }
          TRIDICIPOLE[0]   := TRIDICIPOLE[J+1];
          TRIDICIPOLE[J+1] := TRIDICIPOLE[J];
          TRIDICIPOLE[J]   := TRIDICIPOLE[0];
          VYMENA := true; { uskutečnila se výměna }
        end; { výměna dvou složek pole }
  I := I-1;
  until (I<2) or (not VYMENA);
  { dočtení zbývajících startujících }
  while STCISLO <= ZISKANIMAXCISLA (BAZE) do
    begin { přes všechny zbývající startující }
      VYSLCAS := ZISKANIVC (BAZE, STCISLO);
      if (VYSLCAS>0) and
        (VYSLCAS<TRIDICIPOLE[MAXDELKAVYSLEDKU].
          VCAS)
      then
        begin { bude se zařazovat }
          J := MAXDELKAVYSLEDKU - 1;
          while (VYSLCAS<TRIDICIPOLE[J].VCAS)and(J>0) do
```

```
                begin { posun složky pole }
                  TRIDICIPOLE[J+1] := TRIDICIPOLE[J];
                  J := J-1;
                end; { posun složky pole }
              TRIDICIPOLE[J+1].SCISLO := STCISLO;
              TRIDICIPOLE[J+1].VCAS := VYSLCAS;
            end; { bude se zařazovat }
          STCISLO := STCISLO + 1;
        end; { přes všechny zbývající startující }
    { výpis na obrazovku }
    RADKU := VELIKOSTOBRAZOVKY +2; { počáteční nastavení }
    for I:=1 to POCET do
      begin { výpis výsledků }
        if RADKU >= VELIKOSTOBRAZOVKY then
          begin { zastavit – obrazovka zaplněna }
            if RADKU < VELIKOSTOBRAZOVKY + 2 then
              begin { 2. a další průchody }
                readln;     { čekání na odpověď operátora }
                RADKU := 0;
              end  { 2. a další průchody }
            else
              begin { poprvé }
                writeln('Vysledky');
                writeln;
                RADKU := 2; { počet řádků nastavit podle záhlaví }
              end; { poprvé }
          end; { zastavit – obrazovka zaplněna }
        RADKU := RADKU +1;   { počítadlo řádků + 1 }
        write(I:2,'. ');     { pořadí }
        write(TRIDICIPOLE[I].SCISLO:3,' '); { startovní číslo }
        ZISKANIJMEN(BAZE,TRIDICIPOLE[I].SCISLO,JM,PR,ST);
        write(JM,' ',PR,' ',ST,' '); { jméno, příjmení, státní příslušnost }
        J := TRIDICIPOLE[I].VCAS; { výsledný čas }
        write(J div 3600:2,':');
        J := J mod 3600;
        writeln(J div 60:2,':',J mod 60:2);
      end; { výpis výsledků }
    readln; { čekání na povel operátora }
    writeln('Konec vysledkove listiny');
  end; { VYPVYSLEDKU }

  procedure ZADJMENA ( var BAZE:DATABAZE );
  { zadání obsahu startovní listiny }
```

```
var
   CISLO : TSTCISLA;    { startovní číslo }
   JM,PR : RET10;       { jméno a příjmení }
   ST    : RET3;        { státní příslušnost }
   I     : NEZAP;       { pro indexování }
begin
   writeln ('Zadej startovni cislo');
   readln(CISLO);
   writeln ('Zadej jmeno, prijmeni a stat');
   writeln ('na zvlastni radky');
   I:=0;   { čtení jména po znacích }
   JM:='          ';
   while (not eoln) and (I<10) do
      begin
         I:=I+1;
         read(JM[I]);
      end;
   readln;
   I:=0;   { čtení příjmení po znacích }
   PR:='          ';
   while (not eoln) and (I<10) do
      begin
         I:=I+1;
         read(PR[I]);
      end;
   readln;
   I:=0;   { čtení názvu státu po znacích }
   ST:='  ';
   while (not eoln) and (I<3) do
      begin
         I:=I+1;
         read(ST[I]);
      end;
   readln;
   ZAPISJMEN(BAZE,CISLO,JM,PR,ST);
end; { ZADJMENA }

procedure LISTINA (var BAZE: DATABAZE);
{ výpis startovní listiny }
var
   STCISLO :TSTCISLA;    { právě čtené startovní číslo }
   RADKU   :NEZAP;       { počet vypsaných řádků }
   JM,PR   :RET10;       { jméno a příjmení }
```

```
   ST          :RET3;          { státní příslušnost }
begin { LISTINA }
   RADKU := VELIKOSTOBRAZOVKY +2; { počáteční nastavení }
   for STCISLO:=1 to ZISKANIMAXCISLA(BAZE) do
      begin { výpis }
         ZISKANIJMEN(BAZE,STCISLO,JM,PR,ST);
         if ST <> '???' then
            begin { toto startovní číslo je v databázi }
               if RADKU >= VELIKOSTOBRAZOVKY then
                  begin { zastavit – obrazovka zaplněna }
                     if RADKU < VELIKOSTOBRAZOVKY + 2 then
                        begin { 2. a další průchody }
                           { čekaní na odpověď operátora }
                           readln;
                           RADKU := 0;
                        end  { 2. a další průchody }
                     else
                        begin { poprvé }
                           writeln('Startovni listina');
                           writeln;
                           { počet řádků nastavit podle záhlaví }
                           RADKU := 2;
                        end; { poprvé }
                  end; { zastavit – obrazovka zaplněna }
               RADKU := RADKU +1; { počítadlo řádků + 1 }
               writeln(STCISLO:3,'. ',JM,' ',PR,' ',ST,' ');
                     { startovní číslo,jméno, příjmení, státní příslušnost }
            end; { toto startovní číslo je v databázi }
      end; { výpis }
   writeln;
   writeln('Konec startovni listiny');
end; { LISTINA }

function URCENITYPU: TCINNOSTI;
{ určení typu činnosti, která se bude provádět }
var
   ZNAK : char;   { pracovní proměnné }
   DALSICINNOST : TCINNOSTI;
begin { URCENITYPU }
   repeat
      writeln('Zadej kod cinnosti');
      readln(ZNAK);
      case ZNAK of
```

```
        's','S'   : DALSICINNOST  := ZSTARTU;
        'c','C'   : DALSICINNOST  := ZCILE;
        'v','V'   : DALSICINNOST  := VYSLEDKY;
        'j','J'   : DALSICINNOST  := ZJMENA;
        'l','L'   : DALSICINNOST  := ZLISTINA;
        'k','K'   : DALSICINNOST  := KONEC;
        else
          begin { chybný kód }
            writeln('Chybny kod cinnosti');
            writeln;
            writeln('S ... cas startu');
            writeln('C ... cas v cili');
            writeln('V ... vysledkova listina');
            writeln('J ... zadani jmena, prijmeni a statu');
            writeln('L ... vypis startovni listiny');
            writeln('K ... konec programu');
            writeln;
            DALSICINNOST := CHYBNYKOD;
          end; { chybný kód }
      end; { case }
    until DALSICINNOST <> CHYBNYKOD;
    URCENITYPU := DALSICINNOST;
  end; { URCENITYPU }

  begin { SPORT }
    writeln('VYPOCET VYSLEDKU');
    writeln;
    POCATECNINASTAVENI(DB);
    CYKLUJ:=true;  { příznak nastavit na režim práce }
    repeat
      case URCENITYPU { určení typu činnosti } of
        ZSTARTU  : ZADSCASU (DB); { zadání startovního času }
        ZCILE    : ZADCCILE (DB); { zadání času v cíli }
        VYSLEDKY: VYPVYSLEDKU (DB); { výsledková listina }
        ZJMENA   : ZADJMENA (DB); { zadání jména a státu }
        ZLISTINA : LISTINA (DB); { startovní listina }
        KONEC    : CYKLUJ := false;{ konec programu }
      end; { case }
    until not CYKLUJ;
    writeln('Konec programu SPORT');
  end. { SPORT }
```

Program 8.1

8.3 Cvičení

1. Vytvořte datové typy záznam pro vyjádření těchto objektů reálného světa:
   - student (jméno, příjmení, bydliště, datum narození, studijní průměr),
   - zaměstnanec (jméno, příjmení, bydliště, datum narození, zařazení, plat),
   - skladová položka (název, evidenční číslo, počet kusů, výrobce),
   - kniha v knihovně (název, autor, vydavatelství, počet stran, datum půjčky).
2. Vytvořte pole záznamů o občanech (jméno, příjmení, datum narození). Pole setřiďte podle stáří občanů a v tomto pořadí vytiskněte.
3. Ze zadaných výsledků turnaje v hokeji (kopané) vytiskněte tabulku, ve které budou účastníci seřazeni podle umístění. Do tabulky zahrňte počet zápasů, vítězství, remíz a porážek, celkové skóre a počet bodů.

   Návod: Za výhru se udělují 2 body, za remízu 1 bod, za prohru žádný bod. Účastníci se seřadí podle počtu bodů. Mají-li dva účastníci stejný počet bodů, pak rozhoduje:
   - počet bodů ze vzájemných zápasů,
   - rozdíl počtu dosažených a obdržených branek ze vzájemných utkání,
   - jsou-li oba předchozí údaje shodné, pořadí nelze rozlišit.
4. Pro datový typ komplexních čísel

```
type
  KOMPLEX =
    record
      RE,          { reálná část }
      IM: real;    { imaginární část }
    end;
```

   navrhněte a realizujte procedury pro součet a rozdíl dvou komplexních čísel. Reálná část součtu je součtem reálných částí. Imaginární část součtu je součtem imaginárních částí. Obdobně je tomu u rozdílu.

9 VSTUPY A VÝSTUPY

Ve všech našich programech jsme využívali vstupní, resp. výstupní zařízení (klávesnici, resp. displej) a do záhlaví programu jsme zapisovali identifikátory *input* a *output*, o nichž se blíže nemluvilo.

V této kapitole popíšeme podrobně, jak program zapsaný v Pascalu spolupracuje se vstupními a výstupními zařízeními počítače prostřednictvím strukturovaného datového typu *soubor*.

Všechny dosud diskutované datové typy mají jednu společnou vlastnost. Proměnné těchto typů jsou v průběhu výpočtu uloženy v paměti počítače. K jednotlivým elementům paměti máme rychlý a přímý přístup realizovaný v programu zápisem identifikátoru proměnné. Aby však bylo možno s hodnotami v operační paměti pracovat, musí existovat prostředky, které umožňují údaje získané na vstupu ve vhodném okamžiku přenést do paměti a naopak. Kromě toho je třeba si uvědomit, že s počítačem nelze komunikovat jinak než přes vstupní a výstupní zařízení. Proto je práce se vstupními a výstupními zařízeními důležitou částí programů.

Organizace údajů na vstupních a výstupních zařízeních bývá různá. Většinou údaje nejsou získávány ze vstupu všechny najednou a nejsou rovněž předávány vcelku na výstup. Přesun probíhá postupně po úsecích jisté velikosti, kterým říkáme obvykle *bloky* (nezaměňovat s bloky určujícími blokovou strukturu programu).

Z klávesnice vstupují do počítače postupně jednotlivé řádky (vždy po stisknutí klávesy ENTER, resp. RETURN). Také na obrazovku vystupuje jeden řádek za druhým a tiskárna tiskne po řádcích. Podobně i na pružných discích jsou údaje obvykle uloženy v blocích pevné délky.

Jinou charakteristickou vlastností přenosu ze vstupních a do výstupních zařízení je programem neomezená velikost přenášených údajů, tj. z hlediska programu neexistuje omezení počtu přenášených bloků. Tiskárna tiskne tak dlouho, dokud jsou jí z počítače zasílány znaky, na displeji jsou zobrazeny také všechny údaje, které program chce vypsat.

Přenosy ze vstupních a do výstupních zařízení jsou nedílnou součástí programů, jak jsme je měli možnost poznat. Zatím jsme využívali příkazů *write*, resp. *writeln* pro výpis a příkazů *read*, resp. *readln* pro čtení údajů. Ve vhod-

nou chvíli prováděly programy těmito příkazy přenos údajů ze vstupního zařízení do operační paměti (čtení), popř. přenos údajů z operační paměti na výstupní zařízení (zápis).

V následujících odstavcích popíšeme datový typ *soubor*. Tento typ umožňuje abstrahovat od konkrétních vlastností vstupních i výstupních zařízení a psát program bez znalosti jejich specifických vlastností.

9.1 Obecné soubory

Soubor (obr. 9.1) je strukturovaný datový typ. Skládá se ze složek, které jsou (podobně jako u pole) *stejného* datového typu. Na rozdíl od pole, jehož počet složek je *konečný* a předem daný v popisu typu, počet složek souboru je předem *neomezený* a z hlediska Pascalu neexistuje omezení počtu jeho složek (délky). Důležité je, že vždy lze rozeznat *konec souboru* nacházející se za poslední složkou souboru. Zatímco ke složkám pole lze přistupovat přímo udáním indexu a každá ze složek pole je v libovolném okamžiku činnosti programu ihned k dispozici, u souboru tomu tak není.

K dispozici je právě jediná složka souboru, která je *aktivní*. Chceme-li získat nebo zapsat hodnotu jiné složky, musíme tuto složku zaktivovat. U pascalského souboru se změny aktivní složky provádějí standardními procedurami.

Na obr. 9.1 vidíme, že soubor je tvořen posloupností složek jistého datového typu. Tento typ je libovolný s jedinou výjimkou. Nemůže to být opět soubor nebo strukturovaný datový typ, jenž má složku typu soubor. Jinak může jít o libovolné jednoduché nebo i strukturované datové typy (velmi často mívá soubor složku datového typu záznam nebo *char*).

Soubor nesmí být nikdy použit jako parametr procedury nebo funkce volaný hodnotou. Soubory nesmí také být navzájem přiřazovány přiřazovacím příkazem.

Datový typ soubor popisujeme takto (file = soubor):

file of *libovolný typ mimo soubor*

Můžeme pak deklarovat proměnné typu, např.

```
type
  RADEK = packed array [1..80] of char;

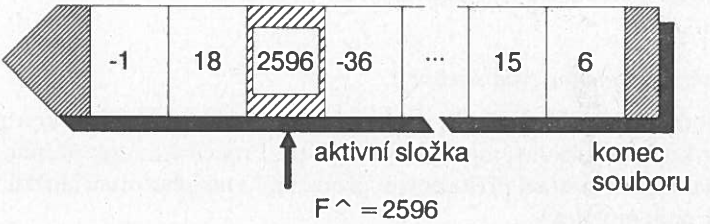
var
  PASKA : file of RADEK;{ soubor PASKA se složkami RADEK }
```

Chceme-li v programu pracovat s aktivní složkou souboru, zpřístupníme ji pomocí *přístupové proměnné souboru*

proměnná typu soubor ^

Přístupovou proměnnou si můžeme představit jako pomocnou proměnnou, do níž při vstupu údajů ze souboru přenášíme hodnotu aktivní složky

```
var
  F: file of integer;
```



Obr. 9.1 Typ soubor

souboru, abychom s ní mohli v programu pracovat. Stejně při výstupu údajů si nejprve připravíme hodnotu nové složky souboru v přístupové proměnné a potom ji zapíšeme do souboru.

V případě souboru PASKA označujeme přístupovou proměnnou PASKA^. Znak ^ nebývá u některých překladačů Pascalu k dispozici a nahrazuje se znakem @. Často lze proto vidět v programech označení přístupové proměnné PASKA@.

Přístupová proměnná je stejného datového typu jako složky souboru a pracuje se s ní stejně jako s proměnnou tohoto typu. Například

```
PASKA^ [25] := 'A';
writeln (PASKA^);
```

Zbývá ještě popsat, jak se mění aktivní složka souboru a s ní i hodnota přístupové proměnné. K tomu, abychom mohli pohyb aktivní složky v souboru popsat, musíme se odděleně zabývat soubory pro vstup a soubory pro výstup, neboť oba mají operace přesunu různé.

Pro vstupní i výstupní soubor je charakteristické, že pohyb aktivní složky je *jednosměrný*, a to od začátku souboru k jeho konci. Operace pohybu nastavují aktivní složku *na začátek souboru* a *posunují* ji směrem ke konci souboru. Pohyb zpět, k začátku souboru, není možný. Je-li přesto třeba vrátit se ke složce ležící před aktivní složkou, je nutné začít znovu od první složky a provést odpovídající počet posunů vpřed.

Soubor může být ve dvou stavech. Ve stavu *zápisu*, kdy vyjadřuje strukturu údajů na výstupním zařízení, a ve stavu *čtení*, kdy vyjadřuje strukturu údajů na vstupním zařízení.

Stav souboru se mění standardními procedurami (reset = nastavit na počáteční stav, rewrite = přepsat)

rewrite (*proměnná typu soubor*)

a

reset (*proměnná typu soubor*).

Procedura *rewrite* vymaže soubor (zruší všechny jeho složky) a připraví soubor k zápisu (nastaví jej do stavu zápisu). Procedura *reset* přepne soubor do stavu čtení a nastaví přístupovou proměnnou na jeho první složku (pokud soubor není prázdný).

Při práci s konkrétními vstupními a výstupními zařízeními není možné přepínat soubory ze stavu čtení do stavu zápisu a zpět libovolně, např. není možné přepnout do stavu čtení soubory na typicky výstupních zařízeních, jako je tiskárna nebo obrazovka displeje, nebo do stavu zápisu typicky vstupní soubor na klávesnici.

Každý soubor musí mít před začátkem práce nastaven stav čtení nebo zápisu procedurou *reset* nebo *rewrite*.

Jde-li o soubor výstupní, je vždy nutné nejprve naplnit přístupovou proměnnou odpovídající hodnotou. Po naplnění se využije standardní procedura (put = dát)

put (*proměnná typu soubor*)

která připojí hodnotu přístupové proměnné jako další složku na konec souboru. Ve vytvářeném souboru je tedy poslední ta složka, jež byla zapsána poslední provedenou procedurou *put*. Za ní se vždy nachází konec souboru. Po provedení operace *put* zůstává hodnota přístupové proměnné nezměněna.

Jde-li o soubor vstupní, je standardní procedurou *reset* nastavena hodnota přístupové proměnné na hodnotu první složky (pokud v souboru nějaká složka existuje). S přístupovou proměnnou, obsahující hodnotu první složky, je možno po provedení operace *reset* v programu pracovat. V okamžiku, kdy je třeba

do přístupové proměnné získat hodnotu další složky souboru (změnit aktivní složku souboru), využijeme procedury (get = vzít)

get (*proměnná typu soubor*)

Tato procedura naplní přístupovou proměnnou hodnotou následující složky souboru (pokud tato existuje). Po provedení operace *get* se hodnota přístupové proměnné změní. Aktivní složkou se stane složka nacházející se za právě aktivní složkou, pokud právě aktivní složka není v souboru poslední.

Pokud by se prováděla procedura *get* a právě aktivní složka souboru by neexistovala (soubor by byl prázdný nebo by již byl proveden posun za poslední složku), dojde k chybě při spuštění programu.

Důležitou operací pro práci se vstupním souborem je dotaz, zda po provedení procedury *get* (popř. i *reset*) je k dispozici další složka souboru (zda jsme nedošli při čtení na konec souboru). K tomu slouží standardní funkce

eof (*proměnná typu soubor*)

s výsledkem typu *boolean*. Funkce *eof* (end of file = konec souboru) nabývá hodnoty *false*, je-li k dispozici aktivní složka, a hodnoty *true* v opačném případě.

Při práci se soubory jsou pak nejčastěji používanými dvojicemi příkazů:

- při zápisu do souboru

proměnná typu soubor ^ := *výraz určující hodnotu složky*;
put (*proměnná typu soubor*);

- při čtení souboru

proměnná pro uložení hodnoty := *proměnná typu soubor* ^;
get (*proměnná typu soubor*);

Pascal umožňuje zkrátit uvedené dvojice příkazů jedním příkazem. První dvojici lze zapsat jako

write(*proměnná typu soubor*, *výraz určující hodnotu složky*);

a druhou dvojici jako

read(*proměnná typu soubor*, *proměnná pro uložení hodnoty*);

Některé implementace Pascalu neznají pojem přístupové proměnné souboru, a nemají tedy ani standardní procedury *get* a *put*. Uživatel musí vystačit s procedurami *read* a *write*. Například zde využívaný Turbo Pascal má toto omezení.

Posledním problémem, který nás bude zajímat před uvedením příkladu programu pracujícího se soubory, je doba existence proměnných, konkrétně souborů.

Víme již, že proměnné a parametry volané hodnotou procedury (funkce) zaniknou v okamžiku, kdy je ukončena její aktivace. Obdobně je tomu u programu. Globální proměnné programu přestanou existovat v okamžiku ukončení činnosti programu. My bychom však někdy potřebovali, aby hodnoty proměnných přežily dobu činnosti programu, popř. bychom chtěli užít hodnoty proměnné vzniklé před začátkem činnosti programu. Nejnaléhavější je tato potřeba právě u souborů obsahujících vstupní údaje programu připravené předem, popř. u výsledků práce programu s delší platností.

Kromě toho bychom také chtěli mít možnost sdělit programu, že některá proměnná není jeho součástí a existuje mimo něj. U souborů např. chceme určit konkrétní vstupní nebo výstupní zařízení, na němž bude soubor umístěn.

Podobně jako u procedur, slouží k předávání údajů mimo program parametry nazvané *parametry programu*, které se zapisují do záhlaví programu

program *jméno programu* (*parametry programu*);

a jsou to jména všech globálních proměnných programu oddělená čárkou, o nichž víme, že existují mimo program. Pro nás půjde vždy jen o soubory existující na konkrétních vstupních a výstupních zařízeních, i když datový typ parametru programu není Pascalem omezen.

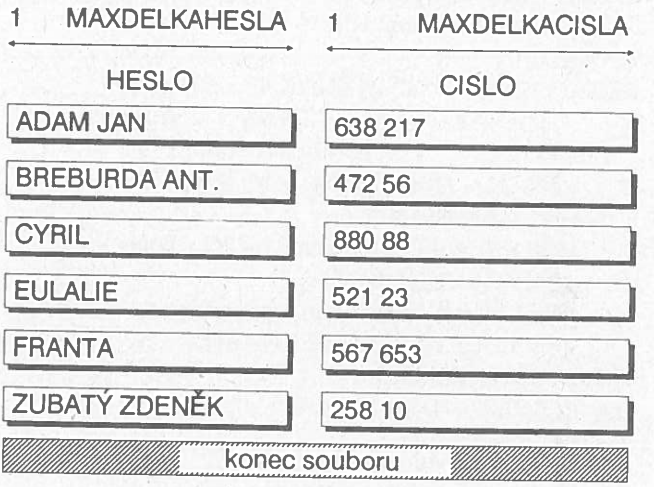
Víme-li např., že budeme užívat soubor PASKA, jehož doba existence není omezena dobou života programu XY, napíšeme

```
program XY (PASKA);
var
  PASKA: file of packed array [1..80] of char;
```

Souborům uvedeným v parametrech programu říkáme v pascalské terminologii *externí* (vnější). Není-li soubor uveden v záhlaví programu, nazývá se *interní* (pracovní,vnitřní) a jeho doba života je stejná jako u proměnných jiných typů deklarovaných v témže bloku. Ovládání vnějších i pracovních souborů je totožné. S vnějšími soubory je tedy program svázán *pouze* jménem v parametrech programu. Konkrétní vazbu jména ke skutečnému souboru na disketě či jinde standardní Pascal neřeší a musí být provedena jinými prostředky. Je možné svázat s programem vnější soubory přežívající dobu existence programu (na magnetických médiích, tiskárně ap.) nebo je možné pracovat s externími soubory vznikajícími průběžně (na klávesnici nebo displeji).

Často také bývají zavedeny nestandardní způsoby vazby programu na konkrétní soubor nebo zařízení. Proto je dobré se vždy přesvědčit, jak tento problém řeší náš konkrétní překladač.

Při psaní programu není ve většině případů důležité, jde-li o soubor na disketě nebo na jiném zařízení. Pro všechna tato zařízení je program z hlediska ovládání souboru totožný. Z hlediska algoritmu ovládání externích souborů je podstatná jen skutečnost, jde-li o soubor přežívající dobu činnosti programu, nebo vznikající průběžně. Podrobněji se touto problematikou zabývá čl. 9.4.



Obr. 9.2 Telefonní seznam

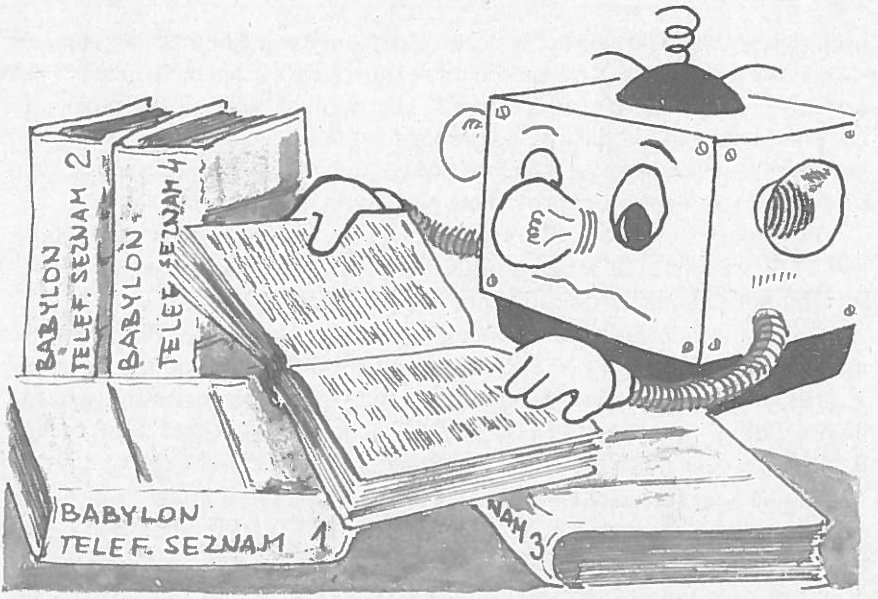
Představme si např., že chceme udržovat ve vnějším souboru telefonní seznam. Neříkáme již, bude-li soubor umístěn na pružném disku či jinde. Složky souboru budou (obr. 9.2) datového typu záznam a budou obsahovat jméno nebo jiné označení (heslo) majitele telefonní stanice a jeho telefonní číslo.

Složky budou v souboru setříděny podle hesel vzestupně. Chceme, aby náš program uměl vyhledávat v telefonním seznamu podle hesel (popř. pouze podle prvních několika písmen hesla) a uměl přidávat i rušit složky seznamu.

Základní tvar programu je

```
program TELEFON (TELSEZNAM, input, output);
var
  TELSEZNAM,                { telefonní seznam }
  KOPIE      : TYPSEZNAMU;  { pracovní kopie }
  KOPIEAKTIVNI : boolean;   { indikace, že kopie má aktuální obsah }
  ZNAK       : char;        { znak určující typ činnosti }
```

```
begin { TELEFON }
  writeln('TELEFONNI SEZNAM');
  writeln;
  KOPIEAKTIVNI := false;        { na začátku není aktivní kopie }
  repeat
    writeln('Chces–li hledat, zadej H, doplnovat D,');
    writeln('rusit R a koncit K');
    ZNAK:=' ';
    while((ZNAK<='a')or(ZNAK>='z')) and
          ((ZNAK<='A')or(ZNAK>='Z')) do
      read(ZNAK);   { čtení řídicího znaku }
    if    (ZNAK='H')or(ZNAK='h') then
      begin { vyhledávání }
        if KOPIEAKTIVNI then { hledej v kopii }
          VYHLEDEJ(KOPIE)
        else { hledej v původním seznamu }
          VYHLEDEJ(TELSEZNAM);
      end  { vyhledávání }
    else if (ZNAK='D')or(ZNAK='d') then
      begin { doplňování }
        if KOPIEAKTIVNI then { doplňuj v kopii }
          DOPLN (KOPIE, TELSEZNAM)
        else { doplňuj v původním seznamu }
          DOPLN (TELSEZNAM, KOPIE);
        KOPIEAKTIVNI := not KOPIEAKTIVNI;
          { změna aktivity }
      end  { doplňování }
    else if (ZNAK='R')or(ZNAK='r') then
      begin { rušení }
        if KOPIEAKTIVNI then { zruš v kopii }
          ZRUS (KOPIE, TELSEZNAM)
        else { zruš v původním seznamu }
          ZRUS (TELSEZNAM, KOPIE);
        KOPIEAKTIVNI := not KOPIEAKTIVNI;
          { změna aktivity }
      end  { rušení }
    else if (ZNAK<>'K')and(ZNAK<>'k') then
      writeln('Chybna odpoved');
  until (ZNAK='K') or (ZNAK='k');
```




```
    if KOPIEAKTIVNI then { uklidit aktivní kopii }
      KOPIRUJ (KOPIE, TELSEZNAM);
    writeln('Konec');
  end. { TELEFON }
```

Datový typ TYPSEZNAMU popisuje pro náš program strukturu souboru z obr. 9.2. Na tento vnější soubor se budeme odkazovat jménem proměnné TELSEZNAM.

Tři navržené činnosti (vyhledání, doplnění a rušení) jsou realizovány procedurami VYHLEDEJ, DOPLN a ZRUS.

Tělo programu na základě konverzace s uživatelem zjistí znak určující jednu z činností a vyvolá odpovídající proceduru. Pro výběr činnosti podle zadaného znaku je užit vícenásobný podmíněný příkaz.

Výběr a provedení jednotlivých činností se provádí tak dlouho (cyklus repeat), dokud není zadán znak 'K'nebo 'k' ukončující činnost.

Procedury DOPLN a ZRUS mění obsah telefonního seznamu. Z uvede-

ných vlastností souboru plyne, že nelze obsah souboru měnit jinak než zápisem složky na jeho konec, je-li ovšem soubor ve stavu zápisu. Chceme-li měnit obsah souboru, jehož stav není určen, popř. chceme-li vkládat složky jinam než na jeho konec (což je náš případ, neboť telefonní seznam je setříděn podle hesel), musíme zkopírovat všechny složky souboru do jiné proměnné typu soubor. Během kopírování provádíme příslušnou změnu obsahu.

Procedury DOPLN a ZRUS budou mít proto dva parametry typu soubor SVSTUP a SVYSTUP a budou vždy z původní hodnoty souboru SVSTUP vytvářet novou hodnotu do proměnné SVYSTUP.

Existuje pracovní kopie telefonního seznamu nazvaná KOPIE, do které mohou procedury DOPLN a ZRUS uložit modifikovaný soubor. Konečný a správný tvar telefonního seznamu může tedy být během činnosti programu uložen buď v proměnné TELSEZNAM, nebo po modifikaci v proměnné KOPIE (po další modifikaci opět v proměnné TELSEZNAM atd.).

Logická proměnná KOPIEAKTIVNI svou hodnotou určuje, zda platný stav seznamu je v proměnné KOPIE (KOPIEAKTIVNI má hodnotu *true*), nebo v proměnné TELSEZNAM (KOPIEAKTIVNI má hodnotu *false*). Na začátku činnosti je platný stav seznamu ve vnějším souboru TELSEZNAM a proměnná KOPIEAKTIVNI má tedy hodnotu *false*.

Každým provedením procedury DOPLN nebo ZRUS se přesune seznam z proměnné TELSEZNAM do KOPIE nebo naopak a hodnota proměnné KOPIEAKTIVNI se změní na opačnou příkazem

KOPIEAKTIVNI := **not** KOPIEAKTIVNI

Procedura VYHLEDEJ neprovádí změnu obsahu seznamu.

Podle hodnoty proměnné KOPIEAKTIVNI jsou určovány vstupní a výstupní parametry procedur DOPLN a ZRUS a vstupní parametr procedury VYHLEDEJ. Má-li proměnná KOPIEAKTIVNI hodnotu *true*, je vstupním parametrem proměnná KOPIE a výstupním proměnná TELSEZNAM, při hodnotě *false* je tomu naopak, např.

```
if KOPIEAKTIVNI then { doplňuj v kopii }
   DOPLN (KOPIE, TELSEZNAM)
else { doplňuj v původním seznamu }
   DOPLN (TELSEZNAM, KOPIE);
```

Podobně je tomu i u procedury ZRUS a při určování vstupního parametru procedury VYHLEDEJ.

Je-li při ukončení programu platný telefonní seznam v proměnné KOPIE, je nutné jej přenést do souboru TELSEZNAM (pracovní soubor KOPIE

po ukončení programu zanikne, vnější soubor TELSEZNAM nikoliv).

```
if KOPIEAKTIVNI then { uklidit aktivní kopii }
  KOPIRUJ (KOPIE, TELSEZNAM);
```

Kopírování jednoho souboru do druhého budeme v našem programu provádět procedurou KOPIRUJ.

Deklarujme nyní datový typ TELSEZNAM. Bude to (jak už bylo několikrát řečeno) soubor

```
const
  MAXDELKACISLA   = 7;   { maximální délka tel. čísla }
  MAXDELKAHESLA   = 19;  { maximální délka hesla }

type
  TYPDELKYCISLA  = 0 .. MAXDELKACISLA; { typ délky čísla }
  TYPDELKYHESLA = 0 .. MAXDELKAHESLA; { typ délky hesla }
  TYPCISLA       = packed array [1..MAXDELKACISLA] of char;
                                  { typ čísla }
  TYPHESLA       = packed array [1..MAXDELKAHESLA] of char;
                                  { typ hesla }
  SLOZKASEZNAMU =  record { typ složky telefonního seznamu }
                     HESLO : TYPHESLA; { heslo }
                     CISLO : TYPCISLA; { číslo }
                   end; { SLOZKASEZNAMU }
  TYPSEZNAMU   = file of SLOZKASEZNAMU; { typ seznamu }
```

Všechna pole znaků jsou v programu realizována jako zhuštěná (**packed**). Umožňuje to snazší porovnávání (nad zhuštěným polem znaků jsou povoleny relační operátory) i snadnější výstup na obrazovku (příkazy *write* i *writeln* dovolují vypisovat zhuštěná pole znaků vcelku jako proměnnou, nikoliv znak po znaku). Ukazuje se, že nemáme-li speciální požadavek na rychlost přístupu, je vždy lepší organizovat pole znaků z uvedených důvodů zhuštěně.

Procedura VYHLEDEJ musí konverzací s uživatelem nejprve zjistit, které heslo se bude vyhledávat.

```
procedure VYHLEDEJ(var SEZNAM: TYPSEZNAMU);
{ vyhledání čísla v seznamu podle hesla }
var
  DELKAHESLA  : TYPDELKYHESLA;{ délka hesla }
  HLEDHESLO : TYPPOLE;        { hledané heslo }
  VYHLED      : boolean;      { příznak vyhledávání }
```

```
  SHODA  : boolean;                { příznak nalezení }
  I      : TYPDELKYHESLA;          { pracovní proměnná }
  XX     : SLOZKASEZNAMU;          { pracovní proměnná }
begin { VYHLEDEJ }
  writeln('Vyhledavani');
  DELKAHESLA := CTIPOLE(HLEDHESLO,
                        MAXDELKAHESLA);
  reset(SEZNAM);  { soubor pro čtení }
  VYHLED := true;   { vyhledávat }
  while not eof(SEZNAM) and VYHLED do
    begin { přes celý seznam }
      read(SEZNAM,XX);
      SHODA:=true;   { předpokládá se shoda hesel }
      for I:=1 to DELKAHESLA do { porovnání }
        SHODA:=SHODA and (XX.HESLO[I]=HLEDHESLO[I]);
      VYHLED:=not SHODA; { není-li shoda, vyhledávej dál }
      if SHODA then
        begin { heslo bylo nalezeno }
          writeln;
          writeln('Nalezen:');
          writeln;
          writeln(XX.HESLO,' tel. cislo: ',XX.CISLO);
          writeln;
        end; { heslo bylo nalezeno }
    end; { přes celý seznam }
  if VYHLED then { není dosud nalezeno }
    writeln('Heslo neni v seznamu');
end; { VYHLEDEJ }
```

Heslo je při vstupu čteno funkcí CTIPOLE. Funkční hodnotou této funkce je skutečná délka přečteného hesla ve znacích. Prvním (výstupním) parametrem je proměnná, do které je přečtené heslo uloženo. Druhým (vstupním) parametrem je maximální délka hesla, kterou může uživatel zadat. Po přečtení hesla je nutné nastavit soubor do stavu čtení a znak po znaku porovnávat zadané heslo s heslem uloženým v aktivní složce souboru. Porovnáváme znak po znaku, protože jsme připustili, že heslo nemusí být shodné jako celek. Stačí zadat jen několik prvních znaků. Proto se porovnání provádí pouze v zadané délce hesla. Výsledkem hledání je buď výpis telefonního čísla a jména při nalezení hesla, nebo hlášení o tom, že zadané heslo v souboru není.

Nalezení hesla je označeno hodnotou *true* logické proměnné SHODA. Příkaz SHODA:=SHODA **and** (XX[I]=HESLO[I]) zajistí nastavení logické proměnné SHODA na hodnotu *false* při prvním neshodném znaku a ponechání hodnoty *false* i při další případné shodě.

```
SHODA:=true;    { předpokládá se shoda hesel }
for I:=1 to DELKAHESLA do { porovnání }
  SHODA:=SHODA and (XX.HESLO[I]=HLEDHESLO[I]);
```

Pracovní proměnná XX slouží k dočasnému uložení složky souboru pro porovnání.

Proměnná VYHLED určuje, zda dané heslo bylo v seznamu nalezeno. Pokud je nalezeno (VYHLED=*false*), je vyhledávání ukončeno a je vypsáno uživatelovo telefonní číslo. Pokud cyklus vyhledávání narazí na konec souboru a po ukončení cyklu má VYHLED hodnotu *true*, je vypsáno hlášení

Heslo neni v seznamu

Abychom mohli funkci CTIPOLE užít nejen pro čtení hesel, ale i pro čtení telefonních čísel, není její hlavička deklarována s parametry a výsledkem typu TYPHESLA a TYPDELKYHESLA

```
function CTIPOLE (var POLE:TYPHESLA; { výstupní pole znaků }
                  KOLIK:TYPDELKYHESLA){ maximální délka }
                  :TYPDELKYHESLA; { čtená délka }
```

nýbrž jsou deklarovány pomocné datové typy

```
const
  MAXDELKAPOLE = MAXDELKAHESLA;
                 { délka pomocného pole }
type
  TYPDELKYPOLE = 0 .. MAXDELKAPOLE;
                 { typ délky pom. pole }
  TYPPOLE      = packed array [1..MAXDELKAPOLE] of char;
```

a funkce CTIPOLE má hlavičku

```
function CTIPOLE (var POLE:TYPPOLE; { výstupní pole znaků }
                  KOLIK:TYPDELKYPOLE) { maximální délka }
                  :TYPDELKYPOLE; { čtená délka }
```

Výsledek takto deklarované funkce lze dosadit jak do proměnné typu TYPDELKYHESLA, tak do proměnné TYPDELKYCISLA (jsou to intervaly téhož datového typu *integer*). Podobně je tomu při dosazování hodnoty skutečného parametru KOLIK. Je-li potřeba přenést hodnotu výstupního parametru

POLE do proměnných typu TYPHESLA nebo TYPCISLA, použije se cyklus a přenáší se znak po znaku (uvidíme to v proceduře DOPLN). Zadání skutečného parametru volaného odkazem typu TYPHESLA není pro formální parametr typu TYPPOLE možný, neboť u parametrů volaných odkazem musí být typ formálních a skutečných parametrů stejný. V proceduře VYHLEDEJ není přenos do proměnné typu TYPHESLA nutný, neboť se stejně provádí porovnání obou polí znaků XX.HESLO a HLEDHESLO znak po znaku.

Procedura DOPLN pro přidání nové složky telefonního seznamu nejprve do proměnné NOVASLOZKA uloží (po konverzaci s uživatelem) heslo a k němu odpovídající telefonní číslo. Čtení hesla i telefonního čísla se provádí funkcí CTIPOLE, přenos do složek záznamu NOVASLOZKA cyklem znak po znaku (jde o pole znaků různých délek, tedy i různých datových typů).

```
procedure DOPLN (var SVSTUP, SVYSTUP:TYPSEZNAMU);
{ vložení nové složky seznamu }
var
  CTENEPOLE     : TYPPOLE;     { přečtené pole znaků }
  DELKAHESLA    : TYPDELKYHESLA; { délka nového hesla }
  DELKACISLA    : TYPDELKYCISLA; { délka nového čísla }
  NOVASLOZKA    : SLOZKASEZNAMU; { nová složka }
  DOPLNENI      : boolean;     { příznak doplnění }
  I             : 0 .. maxint; { pracovní proměnné }
  XX            : SLOZKASEZNAMU;
begin { DOPLN }
  writeln('Doplneni');
  DELKAHESLA := CTIPOLE (CTENEPOLE,
                         MAXDELKAHESLA);
  for I:=1 to MAXDELKAHESLA do { vytvoření nového hesla }
    if I<=DELKAHESLA then
      NOVASLOZKA.HESLO[I]:=CTENEPOLE[I]
    else { zbytek vymezerovat }
      NOVASLOZKA.HESLO[I]:=' ';
  writeln('Telefonni cislo');
  DELKACISLA := CTIPOLE (CTENEPOLE,
                         MAXDELKACISLA);
  for I:=1 to MAXDELKACISLA do { vytvoření nového čísla }
    if I<=DELKACISLA then
      NOVASLOZKA.CISLO[I]:=CTENEPOLE[I]
    else { zbytek vymezerovat }
      NOVASLOZKA.CISLO[I]:=' ';
```

```
reset(SVSTUP);        { příprava pro čtení }
rewrite(SVYSTUP);     { příprava pro zápis }
DOPLNENI:=false;      { příznak doplnění }
while not eof(SVSTUP) do
  begin { přes vstupní seznam }
    read(SVSTUP,XX); { složka vstupního seznamu }
    if (XX.HESLO > NOVASLOZKA.HESLO) and
      (not DOPLNENI) then
      begin { místo pro vložení nové složky }
        write(SVYSTUP,NOVASLOZKA);
        DOPLNENI:=true;  { bylo doplněno }
      end; { místo pro vložení nové složky }
    write(SVYSTUP,XX);
  end; { přes vstupní seznam }
if not DOPLNENI then { doplnit na konec }
  write(SVYSTUP,NOVASLOZKA);
end; { DOPLN }
```

Víme již, že do souboru nelze ukládat nové složky jinak než na jeho konec, a to ještě v případě nastavení souboru do stavu zápisu.

Při aktualizaci je však možné soubor SVSTUP telefonního seznamu okopírovat do pomocného souboru SVYSTUP. Při kopírování se provedou změny.

V našem případě při přepisu parametru SVSTUP do parametru SVYSTUP porovnáváme hesla čtených složek XX.HESLO souboru SVSTUP a heslo nové složky NOVASLOZKA.HESLO

```
if XX.HESLO > NOVASLOZKA.HESLO
```

a v okamžiku splnění tohoto předpokladu zapíšeme naši novou složku do souboru SVYSTUP před původní složku. Tím umístíme nové heslo na správné místo v abecedním uspořádání složek podle hesel. Doplnění souboru novou složkou je označeno hodnotou *true* logické proměnné DOPLNENI.

Není-li během celého průchodu původním souborem SVSTUP nalezena složka splňující uvedenou podmínku (lze to poznat podle hodnoty *false* proměnné DOPLNENI po průchodu souborem), je nová složka největší v abecedním uspořádání podle hesel a je nutné ji doplnit na konec souboru.

Třetí procedurou je ZRUS provádějící rušení složky obsahující zadané heslo:

```
procedure ZRUS (var SVSTUP,SVYSTUP:TYPSEZNAMU);
{ nalezení hesla a zrušení nalezené složky }
```

```
var
  DELKAHESLA  : TYPDELKYHESLA; { délka rušeného hesla }
  RUSHESLO    : TYPPOLE; { rušené heslo }
  VYHLED      : boolean;    { příznak vyhledávání}
  SHODA       : boolean;    { příznak nalezení }
  I           : 0 .. maxint; { pracovní proměnné }
  XX          : SLOZKASEZNAMU;

begin { ZRUS }
  writeln('Ruseni');
  DELKAHESLA := CTIPOLE(RUSHESLO,
                        MAXDELKAHESLA);
  reset(SVSTUP);       { příprava pro čtení }
  rewrite(SVYSTUP); { příprava pro zápis }
  VYHLED:=true;     { příznak vyhledávání }
  while not eof(SVSTUP) do
    begin { přes vstupní soubor }
      read(SVSTUP,XX); { složka vstupního souboru }
      SHODA:=true;   { předpokládá se shoda hesel }
      for I:=1 to DELKAHESLA do { porovnání }
        SHODA:=SHODA and (XX.HESLO[I]=RUSHESLO[I]);
      VYHLED:=VYHLED and (not SHODA);
        { není-li shoda, vyhledávej dál }
      if SHODA then
        begin { heslo bylo nalezeno }
          writeln;
          writeln('Zruseno:');
          writeln;
          writeln(XX.HESLO,' tel. cislo: ',XX.CISLO);
          writeln;
        end { heslo bylo nalezeno }
      else
        begin { není nalezeno }
          write(SVYSTUP,XX);
        end; { není nalezeno }
    end; { přes vstupní soubor }
  if VYHLED then { heslo nebylo nalezeno }
    writeln('Toto heslo neexistuje – nelze zrusit');
end; { ZRUS }
```

Stejně jako u předchozích dvou procedur je nutno nejprve konverzací s uživatelem zjistit heslo složky, která se má zrušit (proměnná RUSHESLO). Podobně jako v proceduře DOPLN, je i u rušení nutné soubor kopírovat.

Při kopírování směrem z původního souboru SVSTUP do souboru SVYSTUP je pro každou složku souboru prováděno porovnání, zda se v aktuální složce souboru XX nachází hledané heslo RUSHESLO. Kontrola se provádí porovnáním znak po znaku v zadané délce hesla DELKAHESLA.

V případě, že je nalezena vhodná složka s totožným heslem, není do kopie zahrnuta.

Nalezení rušené složky je oznámeno uživateli výpisem na obrazovku. Například

```
Zruseno:
Martin Novak        880888
```

Procedura KOPIRUJ pro kopii souboru A do souboru B bude deklarována takto:

```
procedure KOPIRUJ (var A,B: TYPSEZNAMU);
{ kopie souboru A do souboru B }
var
  XX : SLOZKASEZNAMU;
begin { KOPIRUJ }
  reset(A);      { příprava souboru A pro čtení }
  rewrite(B);    { příprava souboru B pro zápis }
  while not eof(A) do
    begin { dokud není konec souboru A }
      read(A,XX);    { čtení složky souboru A }
      write(B,XX);   { zápis složky souboru B }
    end; { while }
end; { KOPIRUJ }
```

Jednotlivé složky vstupního souboru A jsou přenášeny do souboru B, dokud se nenarazí na konec vstupního souboru (*eof*(A) = *true*). Přenos se provádí cyklem while, neboť soubor A může být prázdný a tělo cyklu se nemusí provést ani jednou.

Do úplného programu potřebujeme ještě deklaraci funkce CTIPOLE. Procedura CTIPOLE bude předpokládat, že heslo nebo telefonní číslo bude ukončeno zadáním konce řádku. Ukončení řádku lze při čtení zjistit hodnotou *true* standardní funkce *eoln* . Tato funkce má obdobné vlastnosti jako funkce *eof*. Funkci *eoln* bude podrobněji probrána v čl. 9.2 o textových souborech.

```
function CTIPOLE (var POLE:TYPPOLE; { výstupní pole znaků }
                  KOLIK:TYPDELKYPOLE) { maximální délka }
                  :TYPDELKYPOLE; { přečtená délka }
{ získání pole znaků komunikací s uživatelem }
var
  DELKAPOLE : TYPDELKYPOLE; { skutečná délka }
  ZNAK: char;                          { pracovní proměnná }
begin { CTIPOLE }
  writeln('Zadej maximalne ',KOLIK:1,' pismen, cislic a mezer ');
  DELKAPOLE := 0;    { nulovat skutečnou délku pole }
  readln;
  while not eoln do
    begin { přečtení jednoho řádku }
      read(ZNAK);   { čtení jednoho znaku }
        if (DELKAPOLE<KOLIK) then
          begin { není již přečtena plná délka }
            { skutečná délka +1 }
            DELKAPOLE := DELKAPOLE+1;
            POLE[DELKAPOLE]:=ZNAK;
          end; { není již přečtena plná délka }
    end; { přečtení jednoho řádku }
  CTIPOLE := DELKAPOLE;
end; { CTIPOLE }
```

Čtení řetězce znaků se provádí tak dlouho, dokud není přečten celý řádek znaků

```
while not eoln
```

Do výstupního pole se ukládá maximálně tolik znaků, kolik bylo zadáno v parametru KOLIK.

```
if (DELKAPOLE<KOLIK) then
  begin { není již přečtena plná délka }
    { skutečná délka +1 }
    DELKAPOLE := DELKAPOLE+1;
    POLE[DELKAPOLE]:=ZNAK;
  end; { není již přečtena plná délka }
```

Skutečná délka přečteného pole znaků je hodnotou funkce CTIPOLE.

Celý program pro práci s telefonním seznamem je uveden pod označením program 9.1.

K programu dodejme ještě několik poznámek. Vzhledem k tomu, že Turbo Pascal, ve kterém byl program zapsán, nedovoluje přístup k souborům procedurami *get* a *put*, ale pouze procedurami *read* a *write*, jsou v programu přístupy do souboru realizovány těmito alternativními příkazy.

Kromě toho je také provedena nestandardní vazba proměnné typu soubor se skutečným jménem souboru (které je uvedeno např. v adresáři na pružném disku). V Turbo Pascalu se provádí vyvoláním standardní procedury (assign = přidělit) *assign* (*proměnná typu soubor*, *řetězec znaků*), která spojí jméno uvedené *proměnné typu soubor* se skutečným souborem (např. na pružném disku), jehož jméno je uvedeno ve tvaru *řetězce znaků*. Od vyvolání této standardní procedury platí, že kdykoliv se v programu odkážeme na soubor *proměnná typu soubor*, bude se ve skutečnosti pracovat se souborem *řetězec znaků*. Mimo užití procedury *assign* vyžaduje Turbo Pascal použití procedury *close* při ukončení zápisu do souboru. Příkaz procedury má tvar *close* (*proměnná typu soubor*).

Program 9.1 také vyžaduje, aby před prvním spuštěním již existoval prázdný vnější soubor TELEFON. Lze jej vytvořit např. programem

```
program NOVYSEZNAM (T);
{ Konstanty a typy uvést stejné jako v programu TELEFON }
var
  T : TYPSEZNAMU;    { nový telefonní seznam }
begin   { NOVYSEZNAM }
  assign (T,'TELEFON'); rewrite (T); close (T);
end.    { NOVYSEZNAM }
```

Vlastní program 9.1 má následující tvar:

```
program TELEFON (TELSEZNAM, input, output);
{ Program realizuje telefonní seznam uložený ve formě vnějšího souboru.
Vstupy :
  Na výzvu :   Chces–li hledat, zadej H, doplnovat D, rusit R a koncit K
               se zadá kód činnosti znakem H,D,R nebo K.
Výstupy:
  Zprávy o provedené činnosti, popř. nalezeném telefonním čísle.
}
const
  MAXDELKACISLA = 7;        { maximální délka tel. čísla }
  MAXDELKAHESLA = 19;       { maximální délka hesla }
  MAXDELKAPOLE = MAXDELKAHESLA; { délka pom. pole }
```

```
type
  TYPDELKYCISLA = 0 .. MAXDELKACISLA; { typ délky čísla }
  TYPDELKYHESLA = 0 .. MAXDELKAHESLA; { typ délky hesla }
  TYPDELKYPOLE = 0 .. MAXDELKAPOLE; { typ délky pom. pole }
  TYPCISLA      = packed array [1..MAXDELKACISLA] of char;
                                    { typ čísla }
  TYPHESLA      = packed array [1..MAXDELKAHESLA] of char;
                                    { typ hesla }
  SLOZKASEZNAMU = record { typ složky telefonního seznamu }
                    HESLO : TYPHESLA; { heslo }
                    CISLO : TYPCISLA; { číslo }
                  end; { SLOZKASEZNAMU }
  TYPSEZNAMU    = file of SLOZKASEZNAMU; { typ seznamu }

  TYPPOLE       = packed array [1..MAXDELKAPOLE] of char;

var
  TELSEZNAM,                    { telefonní seznam }
  KOPIE           : TYPSEZNAMU;{ pracovní kopie }
  KOPIEAKTIVNI : boolean; { indikace, že kopie má aktuální obsah }
  ZNAK            : char;       { znak indikující typ činnosti }

procedure KOPIRUJ (var A,B: TYPSEZNAMU);
{ kopie souboru A do souboru B }
var
  XX:SLOZKASEZNAMU;
begin { KOPIRUJ }
  reset(A);      { příprava souboru A pro čtení }
  rewrite(B);    { příprava souboru B pro zápis }
  while not eof(A) do
    begin { dokud není konec souboru A }
      { XX:=A^; get(A); }
      read(A,XX);     { čtení další složky souboru A }
      { B^:=XX; put(B); }
      write(B,XX);    { zápis další složky souboru B }
    end; { while }
  close(B); { uzavření výstupního souboru }
end; { KOPIRUJ }

function CTIPOLE (var POLE:TYPPOLE; { výstupní pole znaků }
                  KOLIK:TYPDELKYPOLE) { maximální délka }
                  :TYPDELKYPOLE; { čtená délka }
{ získání pole znaků komunikací s uživatelem }
```

```
var
   DELKAPOLE : TYPDELKYPOLE; { skutečná délka }
   ZNAK: char;    { pracovní proměnná }
begin { CTIPOLE }
   writeln('Zadej maximalne ',KOLIK:1,' pismen, cislic a mezer ');
   DELKAPOLE := 0;    { nulovat skutečnou délku pole }
   readln;
   while not eoln do
      begin { přečtení jednoho řádku }
         read(ZNAK);   { čtení jednoho znaku }
            if (DELKAPOLE < KOLIK) then
               begin { není již přečtena plná délka }
                  { skutečná délka +1 }
                  DELKAPOLE := DELKAPOLE + 1;
                  POLE[DELKAPOLE]: = ZNAK;
               end; { není již přečtena plná délka }
      end; { přečtení jednoho řádku }
CTIPOLE := DELKAPOLE;
end; { CTIPOLE }

procedure VYHLEDEJ(var SEZNAM: TYPSEZNAMU);
{ vyhledání čísla v seznamu podle hesla }
var
   DELKAHESLA  : TYPDELKYHESLA; { délka hesla }
   HLEDHESLO   : TYPPOLE;    { hledané heslo }
   VYHLED      : boolean;    { příznak vyhledávání }
   SHODA       : boolean;    { příznak nalezení }
   I           : TYPDELKYHESLA; { pracovní proměnná }
   XX          : SLOZKASEZNAMU; { pracovní proměnná }
begin { VYHLEDEJ }
   writeln('Vyhledavani');
   DELKAHESLA: = CTIPOLE(HLEDHESLO, MAXDELKAHESLA);
   reset(SEZNAM);    { soubor pro čtení }
   VYHLED := true;   { vyhledávat }
   while not eof(SEZNAM) and VYHLED do
      begin { přes celý seznam }
         { XX: = SEZNAM ^; get(SEZNAM); }
         read(SEZNAM,XX);
         SHODA: = true;   { předpokládá se shoda hesel }
         for I: = 1 to DELKAHESLA do { porovnání }
            SHODA: = SHODA and (XX.HESLO[I] = HLEDHESLO[I]);
         VYHLED: = not SHODA; { není-li shoda, vyhledávej dál }
         if SHODA then
```

```
            begin { heslo bylo nalezeno }
              writeln;
              writeln('Nalezen:');
              writeln;
              writeln(XX.HESLO,' tel. cislo: ',XX.CISLO);
              writeln;
            end; { heslo bylo nalezeno }
        end; { přes celý seznam }
      if VYHLED then { není dosud nalezeno }
        writeln('Heslo neni v seznamu');
    end; { VYHLEDEJ }

    procedure DOPLN (var SVSTUP, SVYSTUP:TYPSEZNAMU);
    { vložení nové složky seznamu }
    var
      CTENEPOLE     : TYPPOLE;    { přečtené pole znaků }
      DELKAHESLA    : TYPDELKYHESLA; { délka nového hesla }
      DELKACISLA    : TYPDELKYCISLA; { délka nového čísla }
      NOVASLOZKA    : SLOZKASEZNAMU; { nová složka }
      DOPLNENI      : boolean;    { příznak doplnění }
      I             : 0 .. maxint; { pracovní proměnné }
      XX            : SLOZKASEZNAMU;
    begin { DOPLN }
      writeln('Doplneni');
      DELKAHESLA:=CTIPOLE(CTENEPOLE,MAXDELKAHESLA);
      for I:=1 to MAXDELKAHESLA do { vytvoření nového hesla }
        if I<=DELKAHESLA then
          NOVASLOZKA.HESLO[I]:=CTENEPOLE[I]
        else    { zbytek vymezerovat }
          NOVASLOZKA.HESLO[I]:=' ';
      writeln('Telefonni cislo');
      DELKACISLA:= CTIPOLE(CTENEPOLE,MAXDELKACISLA);
      for I:=1 to MAXDELKACISLA do { vytvoření nového čísla }
        if I<=DELKACISLA then
          NOVASLOZKA.CISLO[I]:=CTENEPOLE[I]
        else    { zbytek vymezerovat }
          NOVASLOZKA.CISLO[I]:=' ';
      reset(SVSTUP);          { příprava pro čtení }
      rewrite(SVYSTUP);       { příprava pro zápis }
      DOPLNENI:=false;        { příznak doplnění }
      while not eof(SVSTUP) do
        begin { přes vstupní seznam }
        { XX:=SVSTUP^; get(SVSTUP); }
```

```
        read(SVSTUP,XX); { složka vstupního seznamu }
        if (XX.HESLO > NOVASLOZKA.HESLO) and
          (not DOPLNENI) then
          begin { místo pro vložení nové složky }
          { SVYSTUP^:=NOVASLOZKA; put(SVYSTUP); }
            write(SVYSTUP,NOVASLOZKA);
            DOPLNENI:=true; { bylo doplněno }
          end; { místo pro vložení nové složky }
        { SVYSTUP^:=XX; put(SVYSTUP); }
        write(SVYSTUP,XX);
      end; { přes vstupní seznam }
    if not DOPLNENI then { doplnit na konec }
        { SVYSTUP^:=NOVASLOZKA; put(SVYSTUP); }
        write(SVYSTUP,NOVASLOZKA);
    close(SVYSTUP); { uzavření výstupního souboru }
end; { DOPLN }

procedure ZRUS (var SVSTUP,SVYSTUP:TYPSEZNAMU);
{ nalezení hesla a zrušení nalezené složky }
var
  DELKAHESLA  : TYPDELKYHESLA; { délka rušeného hesla }
  RUSHESLO    : TYPPOLE; { rušené heslo }
  VYHLED      : boolean;   { příznak vyhledávání }
  SHODA       : boolean;   { příznak nalezení }
  I           : 0 .. maxint; { pracovní proměnné }
  XX          : SLOZKASEZNAMU;
begin { ZRUS }
  writeln('Ruseni');
  DELKAHESLA := CTIPOLE(RUSHESLO,MAXDELKAHESLA);
  reset(SVSTUP);          { příprava pro čtení }
  rewrite(SVYSTUP);       { příprava pro zápis }
  VYHLED:=true;           { příznak vyhledávání }
  while not eof(SVSTUP) do
    begin { přes vstupní soubor }
      { XX:=SVSTUP^; get(SVSTUP); }
      read(SVSTUP,XX); { složka vstupního souboru }
      SHODA:=true;  { předpokládá se shoda hesel }
      for I:=1 to DELKAHESLA do { porovnání }
        SHODA:=SHODA and (XX.HESLO[I]=RUSHESLO[I]);
      VYHLED:=VYHLED and (not SHODA);
        { není-li shoda, vyhledávej dál }
      if SHODA then
        begin { heslo bylo nalezeno }
```

```
            writeln;
            writeln('Zruseno:');
            writeln;
            writeln(XX.HESLO,' tel. cislo: ',XX.CISLO);
            writeln;
          end { heslo bylo nalezeno }
        else
          begin { není nalezeno }
            { SVYSTUP^:=SVSTUP^; put(SVYSTUP); }
            write(SVYSTUP,XX);
          end; { není nalezeno }
      end; { přes vstupní soubor }
    if VYHLED then { heslo nebylo nalezeno }
      writeln('Toto heslo neexistuje – nelze zrusit');
    close(SVYSTUP); { uzavření výstupního souboru }
  end; { ZRUS }

  begin { TELEFON }
    assign(TELSEZNAM,'TELEFON');
    assign(KOPIE, 'KOPIE');
    writeln('TELEFONNI SEZNAM');
    writeln;
    KOPIEAKTIVNI := false; { na začátku není aktivní kopie }
    repeat
      writeln('Chces–li hledat, zadej H, doplnovat D,');
      writeln('rusit R a koncit K');
      ZNAK:=' ';
      while ((ZNAK<='a')or(ZNAK>='z')) and
            ((ZNAK<='A')or(ZNAK>='Z')) do
        read(ZNAK);   { čtení řídicího znaku }
      if    (ZNAK='H')or(ZNAK='h') then
        begin { vyhledávání }
          if KOPIEAKTIVNI then { hledej v kopii }
            VYHLEDEJ(KOPIE)
          else { hledej v původním seznamu }
            VYHLEDEJ(TELSEZNAM);
        end  { vyhledávání }
      else if (ZNAK='D')or(ZNAK='d') then
        begin { doplňování }
          if KOPIEAKTIVNI then { doplňuj v kopii }
            DOPLN (KOPIE, TELSEZNAM)
          else { doplňuj v původním seznamu }
            DOPLN (TELSEZNAM, KOPIE);
```

```
            KOPIEAKTIVNI := not KOPIEAKTIVNI; { změna aktivity }
          end  { doplňování }
        else if (ZNAK = 'R')or(ZNAK = 'r') then
          begin { rušení }
            if KOPIEAKTIVNI then { zruš v kopii }
              ZRUS (KOPIE, TELSEZNAM)
            else { zruš v původním seznamu }
              ZRUS (TELSEZNAM, KOPIE);
            KOPIEAKTIVNI := not KOPIEAKTIVNI; { změna aktivity }
          end  { rušení }
        else if (ZNAK < > 'K')and(ZNAK < > 'k') then
          writeln('Chybna odpoved');
      until (ZNAK = 'K') or (ZNAK = 'k');
      if KOPIEAKTIVNI then { uklidit aktivní kopii }
        KOPIRUJ (KOPIE, TELSEZNAM);
      writeln('Konec');
    end. { TELEFON }
```

Program 9.1

9.2 Textové soubory

Soubory, tak jak jsme je poznali v čl. 9.1, měly složky libovolného datového typu. Nyní naši pozornost nyní soustředíme na typ souborů se složkami typu *char,* které jsou základním prostředkem pro komunikaci člověka s počítačem.

Všechny údaje zadávané v písemné formě počítači mají charakter souboru znaků a stejně tak údaje získávané v čitelné formě z počítače.

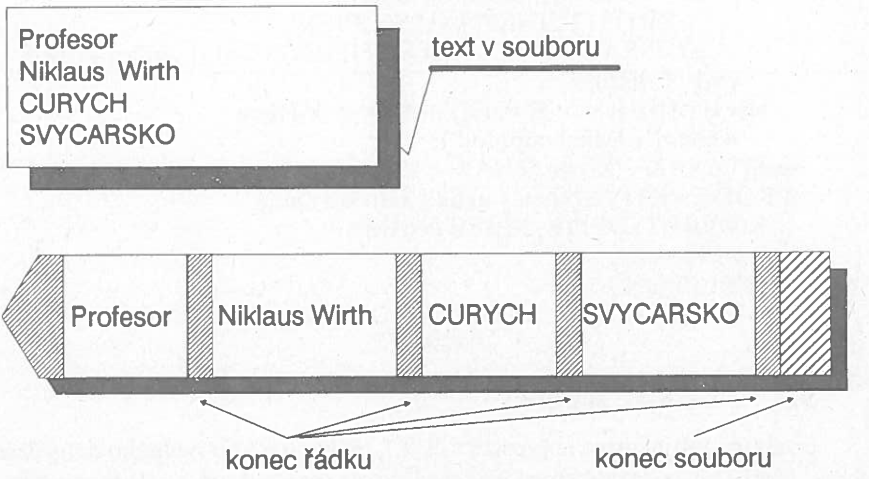
Kromě toho, že složkou souboru je znak, je pro tuto komunikační datovou strukturu typické, že je *členěna na řádky* (obr. 9.3). Souboru, jehož složky jsou znaky a jež je členěn na řádky, říkáme *textový.* Pro textové soubory je v Pascalu k dispozici standardní identifikátor datového typu *text.*

Textovým souborem je např. struktura údajů předávaná z klávesnice, kde řádek je ukončen stisknutím klávesy ENTER, resp. RETURN.

Textové soubory, podobně jako soubory obecné, mohou být rovněž pracovní i vnější, vstupní i výstupní. Lze pro ně používat všechny popsané standardní procedury a funkce *reset, rewrite, get, put, read, write* a *eof.* Je pouze nutné dodat, že narazí-li se při čtení textového souboru operací *get* nebo *read* na konec řádku, je přístupové proměnné přiřazena hodnota ' ' (mezera). Zapsat konec řádku operacemi *put* nebo *write* není možné.

Nutným předpokladem pro komunikaci člověka a počítače je převod (konverze) textového tvaru číselných údajů srozumitelného pro člověka na vnitřní

dvojkový tvar srozumitelný počítači a naopak. Tuto konverzi operace *get* a *put* neumí provést. Aby nebylo nutné konverzi vždy opakovaně programovat, byly pro textové soubory rozšířeny vlastnosti procedur *read* a *write* a zavedeny nové operace nové *readln*, *writeln*, *eoln* a *page*.



Obr. 9.3 Textový soubor

Vlastnosti operací *read*, *readln*, *write* a *writeln*, týkající se konverzí údajů, jsme již probrali v předchozích kapitolách. Zopakujme pouze stručně, že příkazem *read* lze číst z textového souboru celočíselné, reálné a znakové hodnoty, přičemž při čtení celočíselných a reálných hodnot se provádí převod textového tvaru čísel na tvar vnitřní. Příkazem *write* lze vypisovat do textového souboru hodnoty celočíselné, reálné, znakové, logické a řetězce znaků. Všechny tyto výstupy lze provádět formátovaně, přičemž při výstupu celočíselných, reálných a logických hodnot se provádí převod jejich vnitřního tvaru na tvar textový.

Upřesněme nyní vztah operací *read*, *readln*, *write* a *writeln* k textovým souborům.

Procedura *read* má obecně tento tvar volání:

read(*proměnná typu text*, *proměnná*$_1$, *proměnná*$_2$, ..., *proměnná*$_n$);

Prvním skutečným parametrem procedury *read* je textový soubor, který musí být ve stavu čtení (tj. provedení operace *read* musí předcházet provedení

operace *reset*). Ostatní proměnné jsou datových typů *integer*, *real* nebo *char*, které budou postupně naplněny hodnotami odpovídajícího datového typu uloženými v textovém tvaru v daném souboru. Kromě typů *char* a *integer* se připouštějí i jejich intervaly. Pro proměnné datových typů *integer* a *real* se provede převod textového tvaru hodnoty na hodnotu příslušného typu a před čtením vlastní hodnoty se přečtou všechny mezery a prázdné řádky. Alespoň jedna proměnná, do níž se bude ukládat čtená hodnota, musí být ve skutečných parametrech procedury *read* uvedena.

Procedura *readln* má obecně tento tvar volání:

readln(*proměnná typu text*, *proměnná*$_1$, *proměnná*$_2$, ..., *proměnná*$_n$);

Význam parametrů je týž jako u procedury *read*, s tím rozdílem, že procedura *readln* provede po přečtení *proměnné*$_n$ přechod na další řádek souboru. Jinými slovy, přeskočí všechny zbývající znaky řádku a zastaví se na nejbližším začátku nového řádku (pokud další řádek existuje). Na rozdíl od procedury *read* nemusí *readln* číst hodnotu proměnné. Není-li uvedena ani jediná proměnná pro čtení, je proveden pouze přechod na nový řádek.

Obdobné vlastnosti mají procedury *write* a *writeln*.

Procedura *write* má tento obecný tvar volání:

write(*proměnná typu text*, *parametr write*$_1$,
parametr write$_2$,
:
parametr write$_n$);

Parametr write má tvar

výraz nebo
výraz : *výraz*$_1$ nebo
výraz : *výraz*$_1$: *výraz*$_2$

Prvním skutečným parametrem procedury *write* je textový soubor, který musí být předem nastaven procedurou *rewrite* do stavu zápisu. Výrazy v dalších parametrech mohou být datového typu *integer*, *real*, *char*, *boolean*, popř. to mohou být řetězce znaků (typu **packed array** [1..N] **of** *char*). U výrazů typu *integer*, *real* a *boolean* se provádí převod hodnoty příslušného typu na textový tvar. Reálné hodnoty se zaokrouhlí. U všech výstupních položek je možno zadat formát výstupní hodnoty tak, jak bylo popsáno u jednotlivých datových typů v předchozích kapitolách.

Ve skutečných parametrech procedury *write* musí být uveden alespoň jeden *parametr write* .

Procedura *writeln* má obecně tento tvar volání:

writeln(*proměnná typu text*,*parametr write*$_1$,
parametr write$_2$,
:
parametr write$_n$);

Význam parametrů je totožný s významem parametrů procedury *write*. Procedura *writeln* provede navíc *po* výpisu textového tvaru všech hodnot přechod na nový řádek (ukončí zápis řádku). Je to jediný způsob, jak ukončit řádek ve výstupním textovém souboru.

Není-li v parametrech procedury *writeln* uveden *parametr write*, provede se pouze přechod na nový řádek (zápis konce řádku).

Je-li textový soubor ve stavu čtení, je možno testovat, zda nebyl právě přečten poslední znak řádku. K tomu slouží standardní funkce *eoln* s výsledkem typu *boolean* volaná se skutečným parametrem určujícím textový soubor

eoln (*proměnná typu text*)

Je-li právě přečten poslední znak řádku, má funkce *eoln* hodnotu *true*. V opačném případě má funkce *eoln* hodnotu *false*.

Poslední standardní procedurou nad textovým souborem je procedura *page* (page = stránka).

page (*proměnná typu text*)

Vyvoláním této procedury s parametrem udávajícím textový soubor se provede na výstupu přechod na začátek nové stránky, pokud to charakter výstupního zařízení dovoluje (např. tiskárna).

S využitím popsaných procedur a funkce můžeme nyní zapsat např. program pro kopírování vnějšího textového souboru A do vnějšího textového souboru B (program 9.2). Během kopírování jsou všechny skupiny mezer s délkou větší než jedna zkráceny na jedinou mezeru. Prázdné řádky se zachovávají.

```
program KOPTEXT (A,B);
{ Program provádí kopii textového souboru A do B

Vstup programu: textový soubor A

Výstup programu : textový soubor B
   V souboru B jsou všechny posloupnosti mezer s délkou větší než 1
   zkráceny na délku 1
}
```

```
var
   A,B  : text;       { vnější textové soubory }
   ZNAK : char;       { pracovní proměnná }

begin { KOPTEXT }
   assign(A,'A');
   assign(B,'B');
   reset(A);          { příprava pro čtení }
   rewrite(B);        { příprava pro zápis }
   while not eof(A) do
      begin { přes celý soubor A }
         ZNAK := ' * '; { na začátku řádku ZNAK není mezera }
         while not eoln(A) do
            begin { přes celý řádek }
               if ZNAK=' ' then { vynechat další mezery }
                  while (ZNAK=' ') and (not eoln(A)) do
                     read(A,ZNAK)
               else { nemezerový znak }
                  read(A,ZNAK);
               write(B,ZNAK);
            end; { přes celý řádek }
         readln(A); { přechod na nový řádek }
         writeln(B);
   end; { přes celý soubor A }
   close(B);
end. { KOPTEXT }
```

Program 9.2

Spojení identifikátoru proměnné soubor a skutečného jména souboru je opět (stejně jako v programu 9.1) provedeno standardní procedurou Turbo Pascalu *assign* a uzavření souboru operací *close*.

Proměnná ZNAK obsahuje vždy před čtením následujícího znaku znak předchozí. Byla-li předchozím znakem mezera, všechny další mezery se vynechají. Vždy předtím, než se začne číst další řádek, přiřadí se proměnné ZNAK nemezerová hodnota (v našem případě ' * ', ale může to být libovolný znak mimo mezeru). Tím je zajištěno, že první znak na řádku bude vždy přenesen do výstupního souboru, což zajišťuje i přenos řádků obsahujících pouze mezery.

9.3 Soubory *input* a *output*

Nyní se již dostáváme k tomu, abychom objasnili význam identifikátorů *input* a *output*, které jsme zapisovali do záhlaví programů, když jsme měli v úmyslu provádět vstup, resp. výstup z klávesnice, resp. na obrazovku.

Input (input = vstup) a *output* (output = výstup) jsou *standardní textové* soubory. Vzhledem k tomu, že jsou užívány téměř v každém programu, je usnadněno jejich použití některými zjednodušeními zápisu.

Soubory *input* a *output není nutné deklarovat*, stačí uvést jejich identifikátory v parametrech programu. Plná deklarace vnějšího textového souboru by vyžadovala zápis

```
program XY (input, output);
var
  input, output: text;
```

Deklaraci proměnných *input* a *output* uvedenou za vyhrazeným slovem **var** však takto *nesmíme* provést.

Další vlastností souborů *input* a *output* je automatické nastavení stavu obou těchto souborů.

U všech ostatních souborů mimo *input* a *output* je nutné před prvním čtením či zápisem provést nastavení stavu operací *reset* nebo *rewrite*. U souboru *input* (je-li uveden v záhlaví programu) překladač Pascalu zajistí, aby se automaticky provedla operace *reset před provedením prvního příkazu programu*. Obdobně se na stejném místě provede operace *rewrite* pro soubor *output* (je-li uveden v záhlaví programu).

Je určeno, že soubor *input* je *pouze vstupní* a jeho stav nelze změnit operací *rewrite*. Naopak soubor *output* je *pouze výstupní* a jeho stav nelze změnit operací *reset*.

Posledním zjednodušením práce se soubory *input* a *output* je zkrácení zápisu volání procedur *read*, *readln*, *write*, *writeln*, *page* a funkcí *eof* a *eoln*. Jsou-li použity pro soubory *input* nebo *output*, je dovoleno vynechat při volání těchto operací první skutečný parametr určující textový soubor. Není-li tedy tento parametr uveden, pracují procedury *read*, *readln* a funkce *eof* i *eoln* se souborem *input* a procedury *write*, *writeln* a *page* se souborem *output*.

Většina našich programů se soubory *input* a *output* pracuje a uvedených zkrácených zápisů využívá. Program 9.2 upravený pro kopírování souboru *input* do souboru *output* může mít (za použití popsaných zkrácení) tvar 9.3.

```
program KOPTEXT (input, output);
{ Program provádí kopii textového souboru input do output

Vstup programu: textový soubor input
Výstup programu : textový soubor output
   V souboru output jsou všechny posloupnosti mezer s délkou větší než
   1 zkráceny na délku 1
}
var
   ZNAK : char; { pracovní proměnná }
begin { KOPTEXT }
   while not eof do
      begin { přes celý soubor input }
         ZNAK := ' * '; { na začátku řádku ZNAK není mezera }
         while not eoln do
            begin { přes celý řádek }
               if ZNAK = ' ' then { vynechání mezer }
                  while (ZNAK = ' ') and (not eoln) do
                     read(ZNAK)
               else { nemezerový znak }
                  read(ZNAK);
               write(ZNAK);
            end; { přes celý řádek }
         readln; { přechod na další řádek }
         writeln;
      end; { přes celý soubor input }
end. { KOPTEXT }
```

Program 9.3

9.4 Konverzační způsob práce s programem

Vlastnosti souborů *input* a *output* popsané v čl. 9.3 byly určeny pro *dávkový* způsob práce programu. V současnosti je velmi častý *konverzační* způsob práce (v programech této knihy se pracuje převážně konverzačním způsobem).

Při dávkovém způsobu připraví nejprve uživatel programu předem všechny vstupní údaje do vnějšího souboru (např. editorem do souboru na pružný disk). Potom proběhne program, který přečte tento vstupní soubor a vytvoří výstupní soubor (např. na tiskárně). Výstupní údaje obdrží uživatel opět všechny naráz.

Jiná je práce s údaji konverzačním způsobem. Údaje nejsou připraveny předem a také nejsou na výstupu zpracovávány vcelku. Vstup a výstup programu se v průběhu činnosti programu střídají, přičemž program obvykle vyžaduje od uživatele zadání vstupních údajů výzvou (výzva je vlastně výstupem programu). Aby měl program smysl, musí výzva bezprostředně předcházet požadovanému vstupu údajů. Pořadí vstupních a výstupních operací není tedy libovolné, ale naopak pevně určené. Tím se konverzační způsob práce liší od dávkového, kdy vstup údajů není třeba vyžadovat výzvou, neboť všechny údaje jsou na vstupu k dispozici.

Při práci s klávesnicí pascalský program čte/zapisuje údaje z/do souboru po celých řádcích. Řádek se čte v okamžiku stlačení klávesy ENTER, resp. RETURN. Z přečteného řádku jsou postupně předávány znaky podle požadavků jednotlivých volání procedury *read*. Provedení procedury *reset* (při přepínání stavu souboru na čtení) nebo *readln* způsobí čtení dalšího řádku. Stejně tak způsobí čtení řádku situace, kdy řádek je již zcela zpracován a vznikne další požadavek na vstupní údaje příkazem *read*.

Připomeňme nyní, že pro soubory *input*, resp. *output* se automaticky provádí (podle vlastností standardního Pascalu) operace *reset*, resp. *rewrite* před provedením prvního příkazu programu.

Pro dávkové zpracování takové umístění operací *reset* a *rewrite* vyhovuje. Rovněž lze považovat za výhodné umístění operace *rewrite* pro konverzační způsob práce se souborem *output*. V konverzačním způsobu práce však často nevyhovuje automatické provádění operace *reset* pro soubor *input*.

U konverzačního způsobu práce vznikají údaje na vstupu i výstupu průběžně a střídání čtení a zápisů do obou typů souborů musí být smysluplné.

Představme si např. tento úsek programu:

```
program XY (input,output);
var
   I,J : integer;
begin
   { zde se provádí automaticky reset(input) a
   rewrite(output) }
   writeln('ZAHLAVI PROGRAMU');
   writeln('Zadej cele cislo I');
   readln(I);
   writeln('Zadej dalsi cislo J');
   readln(J);
   :
```

Pracuje-li program konverzačně, pak *reset*(*input*) , provedený automaticky na začátku programu, bude požadovat vstup prvního řádku z klávesnice. To se projeví tak, že program bude bez jakéhokoliv výpisu na obrazovku očekávat zadání prvního řádku. Zadáme-li řádek textu, vypíše se

```
ZAHLAVI PROGRAMU
Zadej cele cislo I
```

a program začne zpracovávat obsah prvního uloženého řádku a číst z něj hodnotu proměnné I. Ta, jak vidíme, byla zadána již před požadavkem na ni. Po zpracování hodnoty I (příkazem *readln*) se opět bude čekat na vstup, tj. na čtení dalšího řádku. Půjde však již o požadavek na čtení řádku obsahujícího hodnotu J.

Vidíme, že takto napsaný program vždy nejprve vyžaduje údaje a teprve potom vyzývá uživatele k jejich zápisu na klávesnici.

Vhodným rozmístěním příkazů *readln* by bylo možno této chybné následnosti výpisů a čtení zabránit. Nelze však potlačit automatický *reset*(*input*), který na začátku činnosti programu požaduje vstup prvního řádku.

Proto se soubor *input* (pracuje-li standardně) obvykle nehodí pro konverzační způsob práce s programem. Problém standardního konverzačního textového vstupu se řeší různými způsoby.

Má-li překladač pouze soubory pracující standardně, je nutné užít pro vstup z klávesnice jiného textového souboru než souboru *input* (např. můžeme použít souboru se jménem CON; console = operátorský displej). Tento soubor již není standardní, proto se musí deklarovat a jeho stav se musí nastavit příkazem *reset*(CON). To se může provést na vhodném místě programu (nikoliv na úplném začátku). Zde se také provede čtení prvního řádku vstupního souboru.

Konverzační program by pak mohl mít např. tvar

```
program XY (CON, output);
var
  CON : text;
  I,J : integer;
begin
  { zde se provádí automaticky rewrite(output) }
  writeln('ZAHLAVI PROGRAMU');
  writeln('Zadej cele cislo I');
  reset(CON);
  read(CON,I);
  writeln('Zadej dalsi cislo J');
```

```
readln(CON);
read(CON,J);
:
```

Program provádí čtení dalšího řádku příkazem *reset*(CON) nebo *readln*(CON). Čtení řádku se provádí vždy až po výzvě. Následné příkazy *read* zpracovávají pak obsah přečteného řádku.

Další způsoby řešení jsou nestandardní, nicméně často používané, např. čtení řádku až v okamžiku prvního požadavku na vstup znaků v příkazu *read* nebo *readln*. Funkce operace *reset* a *readln* je tedy pozměněna a skutečné čtení řádku je přesunuto na místo vlastního požadavku na vstup hodnoty. To nám umožňuje zapisovat programy přirozeným způsobem jako pro dávkové zpracování:

```
program XY (input, output);
var
  I,J : integer;
begin
  writeln('ZAHLAVI PROGRAMU');
  writeln('Zadej cele cislo I');
  readln(I);
  writeln('Zadej dalsi cislo J');
  readln(J);
  :
```

Jiné řešení upravuje funkci automatické operace *reset*(*input*), která nečte znaky z klávesnice. Do přečteného řádku však uloží jednu nebo více mezer, které uživatel ve skutečnosti nenapsal. Operace *reset* tedy nepožaduje vstup z klávesnice a nečeká na něj. Jinak se se souborem pracuje standardně. Při čtení číselných hodnot jsou mezery a konce řádků před čtením vlastní hodnoty přeskočeny, takže se význam programu pro čtení čísel nezmění. Pozor je však třeba dávat při vstupu znakových hodnot, kde mezera jako znak může hrát významnou úlohu. Při užití této varianty by program

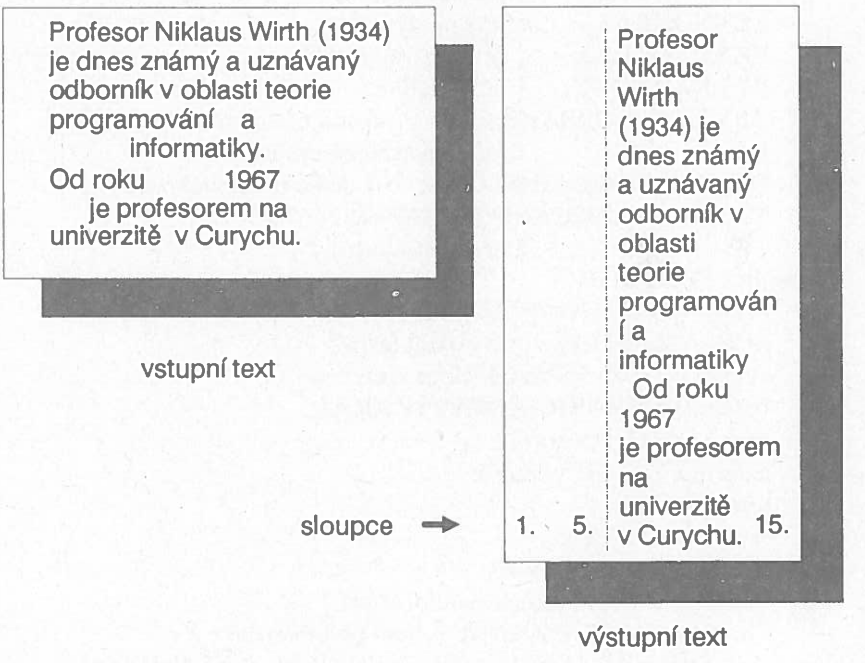
```
program XY (input, output);
var
  I,J : integer;
begin
  { zde se provádí automaticky reset(input) a
  rewrite(output) }
  writeln('ZAHLAVI PROGRAMU');
  writeln('Zadej cele cislo I');
```

```
read (I);
writeln('Zadej dalsi cislo J');
readln;
read (J);
:
```

pracoval (vzhledem k tomu, že jde o čtení celých čísel) rovněž správně.

Všechny programy v této knize předpokládají nestandardní práci se souborem *input* tak, jak ji definuje Turbo Pascal (čtení řádku z klávesnice se provádí při prvním požadavku na vstup hodnoty do programu). Překladače na osobních i malých počítačích tuto variantu práce se souborem také velmi často využívají.

Pokud by překladač Pascalu na některém počítači tento způsob konverzačního vstupu neumožňoval, je nutné zavést do programů nestandardní terminálový vstupní soubor a vstupní příkazy upravit tak, jak byl upraven program se souborem CON v této kapitole.



Obr. 9.4 Formátování textu

Na závěr výkladu textových souborů vytvořme nyní program pro formátování textu. Vstupem programu je textový soubor s libovolnou délkou řádků. Předpokládáme, že text je členěn na slova. Žádné slovo neobsahuje mezeru. Mezi jednotlivými slovy ve vstupním textu může být libovolný počet mezer i prázdných řádků.

Výstupem programu je textový soubor. Je zadán levý a pravý okraj výstupního textu (číslo sloupce, kde bude řádek začínat, a číslo sloupce, kde bude poslední znak řádku). Slova vstupního textu jsou programem umístěna mezi zadané okraje s tím, že jsou vynechány nadbytečné mezery a prázdné řádky (obr. 9.4).

Základní tvar programu pro formátování je :

```
program FORMAT(VSTUP, VYSTUP);
var
   VSTUP,                { vstupní textový soubor }
   VYSTUP : text;        { výstupní textový soubor }
   ZNAK   : char;        { přečtený znak }
   LEVYOKRAJ,            { levý okraj textu }
   PRAVYOKRAJ,           { pravý okraj textu }
   DELKARADKU,           { délka řádku }
   ZBYTEK : ROZSAHRADKU; { dosud nezaplněná část řádku }
   I,                    { řídicí proměnná cyklu }
   DELKA  : ROZSAHSLOVA;     { délka přečteného slova }
   SLOVO  : packed array [1..MAXSLOVO] of char;
                         { přečtené slovo }
begin { FORMAT }
   writeln ('FORMATOVANI TEXTU');
   writeln ('Zadej levy a pravy okraj textu');
   writeln ('jako dve cisla oddelena mezerou :');
   readln (LEVYOKRAJ,PRAVYOKRAJ);
   assign (VSTUP,'vstup');
   assign (VYSTUP,'vystup');
   reset (VSTUP);
   rewrite(VYSTUP);
   if not eof (VSTUP) then
      begin { vstupní soubor není prázdný }
         read (VSTUP,ZNAK); { čtení prvního znaku }
         DELKARADKU := PRAVYOKRAJ - LEVYOKRAJ + 1;
         ZBYTEK := DELKARADKU;
         if LEVYOKRAJ > 1 then { odsazení 1. řádku }
```

```
            write (VYSTUP,' ':LEVYOKRAJ-1);
          VYNECHMEZERY;
        end; { vstupní soubor není prázdný }
      while not eof (VSTUP) do
        begin { přes vstupní soubor }
          DELKA := CTISLOVO; { čtení slova s délkou DELKA }
          if ZBYTEK < DELKA then
            begin   { není dost místa ve zbytku řádku }
              writeln (VYSTUP);
              if LEVYOKRAJ > 1 then { odsazení }
                write (VYSTUP,' ':LEVYOKRAJ-1);
              ZBYTEK := DELKARADKU;
            end; { není dost místa ve zbytku řádku }
          for I:=1 to DELKA do { přenos slova na výstup }
            write (VYSTUP,SLOVO[I]);
          ZBYTEK := ZBYTEK - DELKA; { úprava zbytku }
          if ZBYTEK > 0 then   { zbývá prostor }
            begin   { vlož mezeru }
              write (VYSTUP,' ');
              ZBYTEK := ZBYTEK - 1;
            end; { vlož mezeru }
        end; { přes vstupní soubor }
      close (VYSTUP);
    end. { FORMAT }
```

Na začátku programu je zadán pravý a levý okraj výstupního textu (hodnoty proměnných LEVYOKRAJ a PRAVYOKRAJ). Musí to být nezáporné hodnoty. Procedurami *assign*, *reset* a *rewrite* jsou nastaveny textové soubory VSTUP, resp. VYSTUP do stavu čtení, resp. zápisu (připomeňme, že procedura *assign* je nutná jen pro Turbo Pascal).

Pokud není vstupní soubor prázdný

if not *eof*(VSTUP)

nastaví se počáteční hodnoty proměnných.

Proměnná DELKARADKU udává maximální počet znaků mezi levým a pravým okrajem výstupního textu

PRAVYOKRAJ - LEVYOKRAJ + 1

Proměnná ZNAK obsahuje právě čtený znak vstupního souboru a na začátku je jí přiřazena hodnota prvního znaku souboru VSTUP. Proměnná ZBYTEK

obsahuje hodnotu určující, kolik znaků je možné vložit do výstupního řádku, aniž by byl překročen pravý okraj formátovaného textu (na začátku má hodnotu DELKARADKU).

Pokud není levý okraj nastaven na první sloupec, je vypsán do souboru VYSTUP příslušný počet mezer (je jich LEVYOKRAJ–1) příkazem

```
if LEVYOKRAJ > 1 then
  write (VYSTUP,' ':LEVYOKRAJ-1);
```

Příkazem procedury bez parametrů VYNECHMEZERY jsou vynechány všechny úvodní mezery a prázdné řádky a proměnná ZNAK je nastavena na první znak textu, který není mezerou (pokud takový znak existuje).

Následný cyklus while čte v každém průchodu tělem cyklu jedno slovo a všechny za ním následující mezery a prázdné řádky ze vstupního souboru. Čtení slova provádí funkce CTISLOVO. Výsledkem funkce je délka přečteného slova (bez mezer). Délka je uložena do proměnné DELKA a přečtené slovo do proměnné SLOVO.

Pokud se slovo dané délky již na řádek nevejde

```
if ZBYTEK < DELKA
```

je do výstupního souboru zapsán řádek

```
writeln(VYSTUP);
```

a nastavena počáteční hodnota proměnné ZBYTEK. Jsou rovněž vypsány případné počáteční mezery dalšího výstupního řádku.

Potom je cyklem for přeneseno přečtené slovo do výstupního souboru

```
for I:=1 to DELKA do { přenos slova na výstup }
  write (VYSTUP,SLOVO[I]);
```

a snížena hodnota proměnné ZBYTEK o délku slova

```
ZBYTEK:= ZBYTEK - DELKA; { úprava zbytku }
```

Je-li na řádku místo pro mezeru

```
if ZBYTEK > 0 then    { zbývá prostor }
```

je vypsána mezera oddělující slova a zmenšena hodnota proměnné ZBYTEK

```
begin    { vlož mezeru }
  write (VYSTUP,' ');
  ZBYTEK:= ZBYTEK - 1;
end; { vlož mezeru }
```



Tělo cyklu se opakuje pro všechna slova vstupního textu. Příkaz procedury *close* (VYSTUP) uzavírá výstupní soubor VYSTUP (pouze pro Turbo Pascal).

Procedura VYNECHMEZERY provádí čtení a vynechání všech mezer a prázdných řádků cyklem while.

```
procedure VYNECHMEZERY;
{ vynechání mezer a konců řádků }
begin { VYNECHMEZERY }
   while not eof(VSTUP) and MEZERA(ZNAK) do
      read (VSTUP,ZNAK);
end; { VYNECHMEZERY }
```

Pokud se narazí při čtení na konec souboru, je čtení ukončeno. Výsledná hodnota funkce MEZERA udává, zda znak zadaný jako její parametr je mezerou nebo znakem, který v Turbo Pascalu ukončuje řádky. V takovém případě má funkce výslednou hodnotu *true*, v případě zadání jiného znaku má hodnotu *false*.

Procedura CTISLOVO předpokládá, že v proměnné ZNAK je první nemezerový znak slova.

```
function CTISLOVO: ROZSAHSLOVA;
{ přečtení jednoho slova ze vstupního souboru }
var
  I,                                { řídicí proměnná cyklu }
  DELKA : ROZSAHSLOVA;   { délka slova }
begin
  DELKA := 0; { na začátku je délka slova nulová }
  while not eof (VSTUP) and not MEZERA(ZNAK) and
      (DELKA < DELKARADKU) do
    begin   { čtení slova znak po znaku }
      DELKA := DELKA + 1;
      SLOVO[DELKA] := ZNAK;
      read (VSTUP,ZNAK);
    end; { čtení slova znak po znaku }
  if (DELKA = DELKARADKU) and
    (not MEZERA (ZNAK)) and (not eof (VSTUP)) then
    begin { příliš dlouhé slovo }
      writeln ('Prectene slovo je prilis dlouhe :');
      for I:=1 to DELKA do { výpis chybného slova }
        write (SLOVO[I]);
      writeln;
    end; { příliš dlouhé slovo }
  VYNECHMEZERY;
  CTISLOVO := DELKA;
end; { CTISLOVO }
```

V cyklu while jsou postupně čteny všechny znaky, které jsou součástí slova (test se provádí již popsanou funkcí MEZERA). Je-li během čtení slova dosažen konec souboru (*eof*(VSTUP)), je čtení rovněž ukončeno. Dalším důvodem ukončení čtení je zaplnění celé povolené délky DELKARADKU znaky slova (DELKA >= DELKARADKU).

```
while not eof (VSTUP) and not MEZERA(ZNAK) and
    (DELKA < DELKARADKU) do
```

Je-li celý řádek zaplněn znaky slova a není-li zároveň dosažen konec souboru ani konec slova, je vypsáno varovné hlášení

Prectene slovo prilis dlouhe

Nakonec jsou vynechány všechny případné mezery a konce řádků za přečteným slovem (procedurou VYNECHMEZERY).

Zbývá funkce MEZERA. Ve standardním Pascalu by nebyla nutná. Při čtení mezery i přechodu přes konec řádku je do přístupové proměnné předáván znak mezera. Bylo by proto možné tuto funkci nahradit relací

ZNAK = ' '

Některé překladače, a mezi nimi i Turbo Pascal, při čtení konce řádku předávají programu hodnoty řídicích znaků reprezentujících v souboru skutečný konec řádku. V Turbo Pascalu jsou to znaky CR a LF (čl. 5.1) s ordinálními čísly 13 a 10. Funkce MEZERA má pak hodnotu *true* pro všechny tyto tři znaky (mezera, CR a LF).

```
function MEZERA (ZNAK: char): boolean;
{ test, jde–li o mezeru nebo konec řádku }
const
  LF = 10;     { znaky vyskytující se na konci řádku }
  CR = 13;     { v Turbo Pascalu }
begin { MEZERA }
  MEZERA := (ZNAK = ' ') or
            (ZNAK = chr(CR)) or
            (ZNAK = chr(LF));
end; { MEZERA }
```

Konstanty CR a LF mají hodnotu ordinálních čísel řídicích znaků (znakový literál pro netisknutelné znaky není možné do programu zapsat), proto v relaci je třeba užít funkci *chr* pro převod ordinálního čísla na znak.

Celý program pro formátování je označen 9.4.

```
program FORMAT (VSTUP,VYSTUP);
{ Program kopíruje textový soubor VSTUP do textového souboru
VYSTUP. Výstupní soubor je formátován.

  Vstup: Textový soubor VSTUP, chápaný jako posloupnost
         slov oddělených mezerami a konci řádků.

  Po výzvě
    Zadej levy a pravy okraj textu ...

  je nutné zadat dvě celá čísla určující levý a pravý okraj nového textu.
```

Výstup: Textový soubor VYSTUP, ve kterém je text zarovnán mezi určené okraje a posloupnosti mezer delší než 1 jsou zkráceny na jednu mezeru. Pokud se v souboru VSTUP vyskytne slovo, které se nevejde do zadaného rozsahu řádku, je vypsáno varovné hlášení

```
        Prectene slovo je prilis dlouhe
}
const
  MAXSLOVO = 80; { maximální délka slova }

type
  ROZSAHSLOVA    = 0 .. MAXSLOVO; { typ rozsahu slova }
  ROZSAHRADKU    = 0 .. maxint; { typ rozsahu řádku }

var
  VSTUP,              { vstupní textový soubor }
  VYSTUP : text;      { výstupní textový soubor }
  ZNAK : char;        { přečtený znak }
  LEVYOKRAJ,          { levý okraj textu }
  PRAVYOKRAJ,         { pravý okraj textu }
  DELKARADKU, { délka řádku }
  ZBYTEK : ROZSHRADKU;      { dosud nezaplněná část řádku }
  I,                  { řídicí proměnná cyklu }
  DELKA  : ROZSAHSLOVA;     { délka přečteného slova }
  SLOVO  : packed array [1..MAXSLOVO] of char;
                                { přečtené slovo }

function MEZERA (ZNAK: char): boolean;
{ test, jde-li o mezeru nebo konec řádku }
const
  LF = 10;     { znaky vyskytující se na konci řádku }
  CR = 13;     { v Turbo Pascalu }
begin { MEZERA }
  MEZERA := (ZNAK = ' ') or
            (ZNAK = chr(CR)) or
            (ZNAK = chr(LF));
end; { MEZERA }
```

```
procedure VYNECHMEZERY;
{ vynechání mezer a konců řádků }
begin { VYNECHMEZERY }
  while not eof(VSTUP) and MEZERA(ZNAK) do
    read (VSTUP,ZNAK);
end; { VYNECHMEZERY }

function CTISLOVO: ROZSAHSLOVA;
{ prečtení jednoho slova ze vstupního souboru }
var
  I,              { řídicí proměnná cyklu }
  DELKA : ROZSAHSLOVA;   { délka slova }
begin
  DELKA := 0; { na začátku je délka slova nulová }
  while not eof (VSTUP) and not MEZERA(ZNAK) and
      (DELKA < DELKARADKU) do
    begin   { čtení slova znak po znaku }
      DELKA := DELKA + 1;
      SLOVO[DELKA] := ZNAK;
      read (VSTUP,ZNAK);
    end; { čtení slova znak po znaku }
  if (DELKA = DELKARADKU) and
    (not MEZERA (ZNAK)) and
    (not eof (VSTUP)) then
    begin { příliš dlouhé slovo }
      writeln ('Prectene slovo je prilis dlouhe :');
      for I:=1 to DELKA do { výpis chybného slova }
        write (SLOVO[I]);
      writeln;
    end; { příliš dlouhé slovo }
  VYNECHMEZERY;
  CTISLOVO := DELKA;
end; { CTISLOVO }

begin { FORMAT }
  writeln ('FORMATOVANI TEXTU');
  writeln;
  writeln ('Zadej levy a pravy okraj textu');
  writeln ('jako dve cisla oddelena mezerou :');
```

```
    readln (LEVYOKRAJ,PRAVYOKRAJ);
    assign (VSTUP,'vstup');
    assign (VYSTUP,'vystup');
    reset (VSTUP);
    rewrite(VYSTUP);
    if not eof (VSTUP) then
      begin { vstupní soubor není prázdný }
        read (VSTUP,ZNAK); { čtení prvního znaku }
        DELKARADKU := PRAVYOKRAJ - LEVYOKRAJ + 1;
        ZBYTEK := DELKARADKU;
        if LEVYOKRAJ > 1 then { odsazení 1. řádku }
          write (VYSTUP, ' ':LEVYOKRAJ-1);
        VYNECHMEZERY;
      end; { vstupní soubor není prázdný }
    while not eof (VSTUP) do
      begin { přes vstupní soubor }
        DELKA := CTISLOVO; { čtení slova s délkou DELKA }
        if ZBYTEK < DELKA then
          begin   { není dost místa ve zbytku řádku }
            writeln (VYSTUP);
            if LEVYOKRAJ > 1 then { odsazení }
              write (VYSTUP,' ':LEVYOKRAJ-1);
            ZBYTEK := DELKARADKU;
          end; { není dost místa ve zbytku řádku }
        for I:=1 to DELKA do { přenos slova na výstup }
          write (VYSTUP,SLOVO[I]);
        ZBYTEK := ZBYTEK - DELKA; { úprava zbytku }
        if ZBYTEK > 0 then    { zbývá prostor }
          begin   { vlož mezeru }
            write (VYSTUP,' ');
            ZBYTEK := ZBYTEK - 1;
          end; { vlož mezeru }
      end; { přes vstupní soubor }
    close (VYSTUP);
  end. { FORMAT }
```

Program 9.4

Na závěr kapitoly o souborech rozšíříme ještě náš program SPORT (program 8.1) z kap. 8.

Tento program neumožňoval uložit bázi údajů o závodnících do vnějšího souboru tak, aby nebylo nutné všechny údaje vždy znovu při spuštění programu napsat. Se znalostí souborů doplníme do programu dva nové příkazy U a P. Příkazem U se celý obsah báze údajů uloží do vnějšího souboru DATSPORT a příkazem P se přečte obsah tohoto souboru a naplní se jím báze.

S touto úpravou je pak možné si předem připravit startovní listinu závodníků a při dalším spuštění programu ji opravit i použít. Je také možné kdykoliv během závodu ukončit program (po předchozím uložení obsahu báze) a při novém spuštění programu pokračovat ve vyhodnocování, neboť do souboru jsou ukládány i průběžné časy.

```
program SPORT (input, output, DATSPORT);
{ Úvodní komentář totožný s programem 8.1, pouze jsou doplněny nové
příkazy U a P.
      U ... uložení údajů do souboru
      P ... přečtení údajů ze souboru
}
type
  :
  TSOUBORU   = file of ZAVODNIK; { typ souboru údajů }
  TCINNOSTI  = (CHYBNYKOD,ZSTARTU,ZCILE,
     VYSLEDKY,ZJMENA, ZULOZENI, ZPRECTENI,
     ZLISTINA,KONEC);      { typ kódů činností programu }
:
var
  DATSPORT   : TSOUBORU;   { soubor údajů }
  :
procedure ULOZENI (var BAZE:DATABAZE; var F:TSOUBORU);
{ uložení obsahu databáze do souboru F }
var
  POM: TSTCISLA;       { pomocná proměnná }
begin { ULOZENI }
  assign (F,'datsport');    { příprava pro zápis }
  rewrite (F);
  with BAZE do
    for POM := 1 to MAXSTCIS do { přes všechny údaje }
      write(F,ZAVODNICI[POM]); { zápis údaje }
  close (F);
end; { ULOZENI }
```

```
procedure PRECTENI (var BAZE:DATABAZE; var F:TSOUBORU);
{ uložení obsahu souboru F do proměnné BAZE }
begin { PRECTENI }
  assign (F,'datsport'); { příprava pro čtení }
  reset (F);
  POCATECNINASTAVENI (BAZE);
  with BAZE do
    while not eof (F) do { přes celý soubor }
      begin
        MAXSTCIS := MAXSTCIS + 1; { maximální počet }
        read(F,ZAVODNICI[MAXSTCIS]); { zápis údaje }
      end; { while }
end; { PRECTENI }

{ ostatní procedury }

  :

function URCENITYPU: TCINNOSTI;
{ určení typu činnosti, která se bude provádět }
var
  ZNAK : char;   { pracovní proměnné }
  DALSICINNOST: TCINNOSTI;
begin { URCENITYPU }
  repeat
    writeln('Zadej kod cinnosti');
    readln(ZNAK);
    case ZNAK of
      's','S'  : DALSICINNOST  := ZSTARTU;
      'c','C'  : DALSICINNOST  := ZCILE;
      'v','V'  : DALSICINNOST  := VYSLEDKY;
      'j','J'  : DALSICINNOST  := ZJMENA;
      'u','U'  : DALSICINNOST  := ZULOZENI;
      'p','P'  : DALSICINNOST  := ZPRECTENI;
      'l','L'  : DALSICINNOST  := ZLISTINA;
      'k','K'  : DALSICINNOST  := KONEC;
      else
        begin { chybný kód }
          writeln('Chybny kod cinnosti');
```

```
            writeln;
            writeln('S ... cas startu');
            writeln('C ... cas v cili');
            writeln('V ... vysledkova listina');
            writeln('J ... zadani jmena, prijmeni a statu');
            writeln('U ... ulozeni databaze do souboru');
            writeln('P ... precteni databaze ze souboru');
            writeln('K ... konec programu');
            writeln;
            DALSICINNOST := CHYBNYKOD;
          end; { chybný kód }
      end; { case }
    until DALSICINNOST <> CHYBNYKOD;
    URCENITYPU := DALSICINNOST;
end; { URCENITYPU }

begin { SPORT }
    writeln('VYPOCET VYSLEDKU');
    writeln;
    POCATECNINASTAVENI(DB);
    CYKLUJ:=true;  { příznak nastavit na režim práce }
    repeat
      case URCENITYPU { určení typu činnosti } of
        ZSTARTU  : ZADSCASU (DB);{ zadání start. času }
        ZCILE    : ZADCCILE (DB); { zadání času v cíli }
        VYSLEDKY: VYPVYSLEDKU (DB); { výsledková listina }
        ZJMENA   : ZADJMENA (DB); { zadání jména a státu }
        ZULOZENI : ULOZENI (DB,DATSPORT);
                                  { uložení do souboru }
        ZPRECTENI  : PRECTENI (DB,DATSPORT);
                                  { přečtení ze souboru }
        ZLISTINA  : LISTINA (DB); { startovní listina }
        KONEC     : CYKLUJ := false;{ konec programu }
      end; { case }
    until not CYKLUJ;
    writeln('Konec programu SPORT');
end. { SPORT }
```

Program 9.5

Čtení/zápis údajů z/do souboru DATSPORT provádějí procedury PRECTENI (na příkaz P) a ULOZENI (na příkaz U).

Typ souboru pro ukládání údajů TSOUBORU umožňuje uložení údajů o všech závodnících (jeden údaj za druhým).

Procedura ULOZENI využívá údaj o maximálním startovním čísle v databázi BAZE.MAXSTCIS. Po přípravě souboru pro zápis (*rewrite*(F)) jsou údaje uloženy cyklem for podle startovních čísel do souboru. Není-li některé startovní číslo využito, je do souboru uložen záznam s nedefinovanými hodnotami složek.

Po přípravě souboru (*reset*(F)) provádí procedura PRECTENI vstup cyklem while do konce souboru složku po složce. Průběžně je aktualizováno maximální startovní číslo BAZE.MAXSTCIS a přečtené záznamy ukládány do báze údajů, a to přímo příkazem

read(F,ZAVODNICI[MAXSTCIS]); { zápis údaje }

9.5 Cvičení

1. Vytvořte program, který ze dvou vnějších vstupních souborů se složkami datového typu *integer* setříděnými vzestupně vytvoří vnější výstupní soubor rovněž setříděný vzestupně.
2. Vytvořte program, který čte ze vstupního souboru složky typu **packed array** [1..80] **of** *char*. Do výstupního souboru zapisuje pouze ty složky, které neobsahují samé mezery.
3. Vytvořte program, který čte vnější textový soubor TXT1 a nahradí v celém souboru zadaný znak jiným znakem. Výstup je prováděn do textového souboru TXT2.
4. Vytvořte program, který čte textový soubor *input* a nahradí v něm všechny posloupnosti mezer delší než tři znaky znakem tabelátoru s ordinálním číslem 9.
5. Vytvořte program, který ve vstupním textovém souboru *input* provede součet celkového počtu znaků a řádků a výsledek vytiskne.
6. Vytvořte program, který ve vstupním textovém souboru vyhledá a vypíše řádek obsahující zadanou posloupnost znaků. Pokud se řádek s danou posloupností v souboru nevyskytuje, program vypíše varovné hlášení.
7. Vytvořte program, který vstupní textový soubor bude opisovat na obrazovku displeje, přičemž se výpis vždy po zaplnění obrazovky zastaví. Stisknutím mezerníku a klávesy ENTER, resp. RETURN bude výpis pokračovat. Zadáním znaku 'C' se výpis ukončí.

8. Je dán vstupní soubor se záznamy o zaměstnancích a jejich měsíčních platech. Vytiskněte předpis pro rozdělení výplat jednotlivým zaměstnancům (tj. přidělení bankovek a mincí pro danou částku) a celkovou výčetku všech bankovek a mincí. Výplata musí obsahovat co nejméně bankovek a mincí. Je-li výplata větší než 100 Kčs, musí obsahovat alespoň jednu stokorunu a jednu minci. Výplatu zaokrouhlete na koruny.
9. Vytvořte program, který utvoří soubor záznamů o knihách (autor, název, obor atd.). Vytvořte také program, který
   - vyhledá všechny knihy daného autora (abecedně setříděné podle názvů),
   - vyhledá všechny knihy daného oboru (abecedně setříděné podle autorů),
   - aktualizuje již vytvořený soubor (ruší, opravuje, doplňuje).

   Doplňte program tak, aby automatizoval práci malé knihovny. Kromě souboru s informacemi o knihách bude pracovat s dalším souborem, ve kterém budou záznamy o výpůjčkách. S tímto souborem bude třeba provádět činnosti:
   - zaznamenávat a rušit výpůjčku,
   - vytisknout a evidovat upomínky k danému datu, ap.
10. Textový soubor upravte programem tak, aby tvořil sloupek o 60 znacích umístěný doprostřed stránky o 80 znacích. Rovnoměrným rozložením mezer zarovnejte levý i pravý okraj. Slova se nesmějí rozdělovat. Vytvořte i variantu programu s jednoduchým algoritmem pro rozdělování slov (např. mezi dvěma samohláskami).
11. Z textového souboru čtěte program v Pascalu. Vypište jej do souboru *output* tak, aby vyhrazená slova byla podtržena (podtržení realizujte dalším řádkem obsahujícím pomlčky na příslušných místech).
    Návod: Vytvořte proměnnou TAB typu pole:

```
const
  POCETSLOV = počet vyhrazených slov Pascalu;
type
  VYHRSLOVO = packed array [1..10] of char;
  TABULKASLOV = array [1..POCETSLOV] of VYHRSLOVO;
var
  TAB: TABULKASLOV;
```

    Proměnnou TAB na začátku programu naplňte vzory všech vyhrazených slov Pascalu:

```
TAB[1]          := 'and      ';
TAB[2]          := 'array    ';
:
TAB[POCETSLOV]  := 'with     ';
```

Text pascalského programu je členěn na slova, která jsou oddělena mezerami, poznámkami nebo konci řádků. Každé slovo je třeba přenést do proměnné typu VYHRSLOVO a porovnat s tabulkou vyhrazených slov TAB. Text se čte po celých řádcích a ke každému přečtenému řádku je nutné si poznamenávat (do pole), které znaky budou podtrženy. Po ukončení analýzy řádku se vytiskne jak přečtený řádek, tak řádek, v němž jsou na odpovídajících místech znaky podtržení.

10 MNOŽINY

Máme zjistit, zda znaková proměnná ZNAK je malým nebo velkým písmenem, číslicí, nebo mezerou. Odpovídající logický výraz můžeme se současnými znalostmi jazyka napsat např. takto (předpokládáme souvislost množiny písmen):

```
(((ZNAK >= 'a') and (ZNAK <= 'z')) or
((ZNAK >= 'A') and (ZNAK <= 'Z')) or
((ZNAK >= '0') and (ZNAK <= '9')) or
(ZNAK = ' '))
```

Jde vlastně o zjišťování pravdivosti logického množinového výrazu

$$ZNAK \in \{'a','b',...,'z','A','B',...,'Z','0',...,'9',' '\}$$

Podobně např. výraz

$$I \notin \{3, 5, 7\}$$

bychom mohli napsat v pascalském tvaru

$$(I <> 3) \textbf{ and } (I <> 5) \textbf{ and } (I <> 7)$$

Obdobných logických výrazů využívajících množiny se může v programech vyskytovat mnoho. Často vzniká i potřeba užívat množinové operace sjednocení, průnik i rozdíl. Proto Pascal používá datový typ *množina*, který uvedené výroky umožňuje efektivně do programu zapisovat.

Množina v Pascalu je vždy *konečná*. Prvky, které mohou do definované množiny příslušet, jsou dány ordinálním datovým typem, označovaným při deklaraci typu množina jako *bázový typ*. Množinu nad bázovým typem definujeme takto

set of *bázový typ*

Například je možné deklarovat množiny

```
type
  MNZNAKU      = set of char;   { množina znaků }
  MNCISEL      = set of 3..5;   { množina nad intervalem 3..5 }
  MNTRPASLIKU  = set of TRPASLICI; { typ TRPASLICI; kap. 5 }
```

Podobně jako u ostatních datových typů, je i u množin možné deklarovat libovolné proměnné a parametry procedur i funkcí, např.

```
var
  ZNAKY     : MNZNAKU;
  CISLA     : MNCISEL;
  PRITOMNI  : MNTRPASLIKU;
  PLATNA    : set of 0 .. 127;
```

Hodnoty datového typu množina jsou podmnožiny hodnot bázového typu (včetně prázdné množiny). Tedy např. hodnotami typu MNCISEL jsou množiny

{3,4,5}, {3,4}, {3,5}, {4,5}, {3}, {4}, {5}, {}.

Jde tedy o potenční množinu nad bázovým typem. Konečnost bázového typu zaručuje i konečnost libovolné z množin. Má-li bázový typ *n* hodnot, má potenční množina 2^n hodnot.

Množinové hodnoty se zapisují pomocí *konstruktoru množin*. Je to pascalský zápis obdobný matematickému zápisu množiny definované výčtem prvků s tím rozdílem, že je užito lomených závorek místo obvyklých složených. Možné hodnoty proměnné CISLA typu MNCISEL můžeme zapsat v Pascalu takto:

[3,4,5] [3,4] [3,5] [4,5] [3] [4] [5] [ ]

Libovolnou z těchto množin je možno přiřadit proměnné téhož bázového typu

```
CISLA := [4,5];
CISLA := [ ];          { přiřazení prázdné množiny }
```

popř.

```
PRITOMNI :=  [SMUDLA, BRUMLA, REJPAL];
ZNAKY    :=  ['A', 'C', '%'];
```

Prázdnou množinu lze přiřadit množinové proměnné libovolného bázového typu.

Přestože je konstruktorem množiny možné zapisovat konstantní množiny (literály typu množina), nedovoluje Pascal deklarovat pojmenované množinové konstanty.

Prvkem konstruktoru nemusí být pouze konstanta bázového typu. Do konstruktoru je možné zapisovat i výrazy bázového typu

```
CISLA  :=  [106–101, ord('4')–ord('0')];
ZNAKY  :=  [chr(65), chr(66), chr(67)];
```

a to nejen výrazy konstantní (jako je uvedeno zde), ale i výrazy s proměnnými a funkcemi.

Dále je možné pro zkrácení zápisu v konstruktorech množin uvádět souvislé intervaly, přičemž jak dolní, tak horní mez intervalu tvoří obecně výraz, např.

```
PLATNA    := [ 3..45, 67 ];
ZNAKY     := ['A'..'Z', 'a'..'z', '0'..'9',' '];
PRITOMNI  := [ SMUDLA..BRUMLA, PROFA ];
```

ale i

```
PLATNA := [ I * J .. K mod L, I..J ];
```

kde I, J, K, L jsou proměnné datového typu *integer*. Uživatel však musí zajistit, aby výsledné hodnoty výrazů byly i hodnotami bázového typu, jinak dojde k chybě při spuštění programu.

Nad hodnotami datového typu množina jsou definovány binární operace

- \+ sjednocení množin,
- * průnik množin,
- – rozdíl množin

s výsledkem typu množina stejného bázového typu jako operandy. Mimo to existuje logická operace **in** pro zápis výroků typu

prvek ∈ *množina*

Zápis

výraz bázového typu **in** *množina*

má hodnotu *true*, je-li hodnota daná vyčíslením *výrazu* prvkem *množiny*.

Podobně lze množinovou inkluzi zapsat pomocí operátorů < = a > =. Zápis

$množina_1 <= množina_2$

má hodnotu *true*, je-li $množina_1$ podmnožinou $množiny_2$. Zápis

$množina_1 >= množina_2$

má hodnotu *true*, je-li $množina_2$ podmnožinou $množiny_1$.

Lze též užít relačních operátorů = (rovnost) a < > (nerovnost).

S použitím konstruktoru množiny lze test příslušnosti znaku ZNAK do množiny číslic, písmen a mezery uvedený na začátku této kapitoly (s využitím relačních a logických operátorů) zapsat takto :

```
ZNAK in ['A'..'Z', 'a'..'z', '0'..'9',' '];
```

Obdobně výrok I $\notin$ { 3, 5, 7 } lze zapsat

not (I **in** [3, 5, 7])

Množinový výraz lze zapisovat smíšeně s ostatními výrazy. Nový operátor **in** má stejnou prioritu jako relační operátory (tedy velmi nízkou). Ostatní operátory mají stejnou prioritu jako stejné operátory v číselném výrazu.

Chceme nyní zjistit, která písmena se vyskytují ve vstupním souboru *input*. Vytvoříme proto množinu s bázovým typem *char*:

```
var
  ZNAKY : set of char; { množina znaků }
```

Tuto množinu na začátku programu vyprázdníme přiřazením prázdné množiny.

```
ZNAKY := [ ];    { vyprázdnění množiny }
```

Potom budeme číst textový soubor *input* a pro každý přečtený znak (nachází se v proměnné ZNAK typu *char*) provedeme sjednocení jednoprvkové množiny dané čteným znakem [ZNAK] a množiny ZNAKY

```
read (ZNAK);
ZNAKY := ZNAKY + [ZNAK]; { doplnění čteného znaku }
```

V množině ZNAKY jsou po přečtení celého souboru ty znaky, které se při čtení vyskytly alespoň jednou. Omezení na písmena provedeme např. průnikem

```
ZNAKY := ZNAKY * ['a'..'z', 'A'..'Z']; { omezení na písmena }
```

Výpis výsledků zajistíme cyklem for přes celý bázový typ *char* (předpokládáme, že se pracuje v kódu ASCII s ordinálními čísly znaků od 0 do 127). Přítomnost prvku v množině zjistíme operací **in**.

```
for ZNAK := chr(0) to chr(127) do
  if ZNAK in ZNAKY then { je v množině }
    write (ZNAK);
```

Celý program je označen 10.1.

```
program STATISTIKA (input, output);
{ Program zjišťuje, která písmena se vyskytují v textovém souboru input.
Vstup programu : libovolný
Výstup programu :
  V textovém souboru se vyskytují písmena: xxxxxxxxxx
}
```

```
var
   ZNAKY : set of char;    { množina znaků }
   ZNAK : char;            { čtený znak }
begin { STATISTIKA }
   writeln('STATISTIKA VYSKYTU PISMEN V SOUBORU INPUT');
   writeln;
   writeln('Zadej libovolny obsah souboru');
   ZNAKY := [ ]; { vyprázdnění množiny }
   while not eof (input) do
      begin
         while not eoln do
            begin
               read (ZNAK);
               ZNAKY: = ZNAKY + [ZNAK]; { doplnění čteného znaku }
            end; { while not eoln }
         readln;
      end; { while not eof }
   ZNAKY := ZNAKY * ['a'..'z', 'A'..'Z']; { omezení na písmena }
   writeln('V textovem souboru se vyskytuji pismena :');
   for ZNAK := chr(0) to chr(127) do
      if ZNAK in ZNAKY then { je v množině }
         write(ZNAK,' ');
   writeln;
end. { STATISTIKA }
```

Program 10.1.

Většina implementací Pascalu nedovoluje užívání množin libovolného ordinálního bázového typu. Nejčastější omezení povoluje pouze ty bázové typy, jejichž hodnoty mají ordinální čísla v intervalu 0..127. To zahrnuje pro kód ASCII datový typ *char* a většinu datových typů definovaných výčtem, ale ne např. interval 1001..1020, i když počet jeho hodnot je malý.

Je vhodné si případné omezení zjistit předem z uživatelského manuálu. Existují totiž překladače Pascalu, které nedovolují definici typu **set of** *char*.

10.1 Cvičení

1. Upravte program STATISTIKA tak, aby proměnná ZNAKY byla typu

   packed array [*char*] **of** *boolean*

Návod: Každý datový typ množina je vlastně logickým polem obsahujícím ve složce s indexem daného prvku hodnotu *true* (pokud je prvek v množině) nebo *false* (není-li prvek v množině). Operace * a + realizujte jako procedury a operaci **in** jako logickou funkci.

2. Modifikujte program STATISTIKA tak, aby zjišťoval výskyt číslic, popř. jiných množin znaků.

   Návod: Program je zapsán obecně, stačí proto změnit v operaci průniku masku

   ['a'..'z', 'A'..'Z']

3. Je dán typ MNOZINA

```
const
  DOLNI   = 10;
  HORNI   = 20;
type
  INTER     = DOLNI .. HORNI;
  MNOZINA   = set of INTER;
```

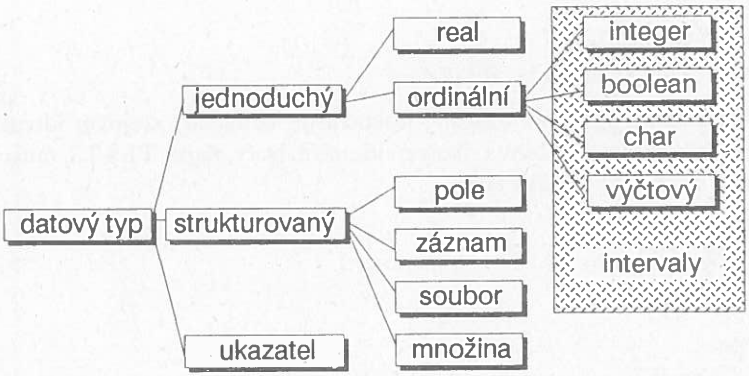
   Vytvořte proceduru, která vytiskne hodnotu typu MNOZINA ve tvaru konstruktoru množiny. Například pro prvky 10, 11, 12 a 16 se vytiskne

   [10..12, 16]

11 UKAZATELE

11.1 Datové typy a jejich vztahy

V předchozích kapitolách jsme poznali jednoduché i strukturované datové typy. Připomeňme, že datový typ určuje množinu hodnot, jichž může jistý objekt jazyka nabýt, a množinu operací nad nimi. Rozdělení všech datových typů jazyka je uvedeno na obr. 11.1.



Obr. 11.1 Rozdělení datových typů Pascalu

Každý objekt jazyka nabývající hodnotu (proměnná, funkce, konstanta) musí mít v Pascalu určen datový typ. V [28] jsou uvedeny některé z důvodů pro zavedení datových typů:

- Znalost datových typů jednotlivých objektů je podstatná pro pochopení algoritmu. Bez jasného popisu datového typu lze těžko zjistit, jaký druh objektů proměnná představuje, což může později ztížit odhalení možných chyb v programu.

- Operátory použité ve výrazech jsou definované jen pro určité datové typy operandů. Překladač je schopen testovat, zda jsou kombinace operátorů a operandů přípustné. Údaj o datových typech slouží jako informace pro přezkoušení správných kombinací již během překladu.
- Realizace operátorů se mohou lišit podle datových typů operandů (např. * znamená pro celá čísla násobení a pro množiny průnik). Znalost datového typu je proto důležitá i pro překlad operátorů.

Datové typy je možné deklarovat v části deklarací typů daného bloku. Zde jsou datové typy pojmenovány identifikátory, např.:

```
type
  NEZAP    = 0 .. maxint;
  RADEK    = packed array [1..80] of char;
```

Kromě toho je možné uvádět popisy datových typů přímo v deklaracích proměnných (bez toho, aby byly předem uvedeny v deklaraci typů):

```
var
  I : 0 .. maxint;
  J : packed array [1..80] of char;
```

Dva datové typy jsou *totožné*, pokud jsou označeny stejným identifikátorem. Mohou být označeny i různými identifikátory, např. T1 a T2, musí však existovat deklarace typu ve tvaru

```
type
  T2 = T1;
```

nebo

```
type
  T1 = T2;
```

Totožné musí být např. datové typy formálního a skutečného parametru procedury (funkce) volaného odkazem.

Dva typy jsou *kompatibilní*, pokud

- jsou totožné,
- jeden je intervalem druhého, nebo jsou oba intervalem téhož ordinálního typu,
- oba jsou typu řetězec se stejnou délkou,
- oba jsou typu množina s kompatibilním bázovým typem.

Kompatibilita typů se využívá pro přesné určení situace, kdy je možné přiřazení výrazu typu T2 do objektu datového typu T1.

Uvažujme objekt datového typu T1 a výraz datového typu T2. Datový typ T2 je *kompatibilní vzhledem k přiřazení* s datovým typem T1, platí-li:

- T1 a T2 jsou totožné a T1, resp. T2 jsou přípustné jako složky souboru (nesmí být typu soubor ani obsahovat typ soubor jako složku),
- T1 je *real* a T2 *integer*,
- T1 a T2 jsou kompatibilní ordinální typy a hodnota výrazu leží v oboru hodnot typu T1,
- T1 a T2 jsou kompatibilní množinové typy a všechny prvky výrazu T2 patří do intervalu, který je určen bázovým typem datového typu T1,
- T1 a T2 jsou kompatibilní řetězce.

Zopakujme nyní některé konstrukce jazyka vyžadující kompatibilitu vzhledem k přiřazení. Výraz T2 musí být kompatibilní vzhledem k přiřazení s typem T1 proměnné v přiřazovacím příkazu

proměnná := *výraz*

Dále musí být výraz T2 kompatibilní vzhledem k přiřazení, jde-li o skutečný parametr volaný hodnotou a odpovídající formální parametr je typu T1. Podobně v cyklu for musí být výrazy určující počáteční a koncovou hodnotu řídicí proměnné kompatibilní vzhledem k přiřazení s datovým typem řídicí proměnné. Stejnou vlastnost musí mít i výraz určující skutečnou hodnotu indexu vzhledem k jeho deklarovanému typu v deklaraci typu pole.

Dodržování požadavků na vlastnosti typů (tzv. *typovou čistotu*) kontroluje překladač. Nesplňuje-li program požadavky, vznikne chyba při překladu.

Možnost pojmenování typů identifikátory je důležitá pro návrh programu shora dolů. Je totiž možné a vhodné nejprve pojmenovat použité datové typy a identifikátory typů používat jak pro deklaraci proměnných, tak parametrů procedur a funkcí. Konkrétní popis datového typu (jeho vnitřní stavbu) není nutné v této fázi ještě znát. Upřesnění typu je tak odloženo až do okamžiku, kdy je pro zápis programu nezbytné (např. pro deklaraci těla procedur a funkcí). Tohoto postupu bylo použito např. při realizaci programů DATUM, NAVIGACE, MERENI, ZIVOT, SPORT atd.

V programu SPORT byl nejdříve pojmenován typ báze údajů DATABAZE a použit pro příkazy procedur a deklaraci proměnné BAZE. Konkrétní deklarace typu byla provedena až nakonec, při deklaraci těl procedur a funkcí, které s typem bezprostředně pracují.

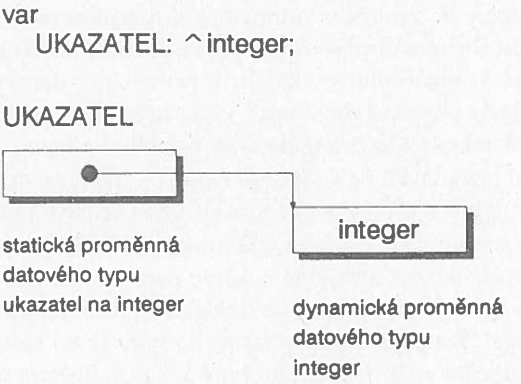
Tím, že využíváme pouze jmen datových typů, abstrahujeme od konkrétní stavby datového typu. Datový typ využíváme jako *abstraktní datový typ* (více o abstraktních datových typech v čl. 11.3).

11.2 Dynamické proměnné

Shrňme nyní některé vlastnosti proměnných datových typů, které jsme dosud poznali.

Proměnné těchto typů jsou označeny identifikátorem. Identifikátor a datový typ jsou každé proměnné přiřazeny v deklaraci. Deklarace je platná v jistém bloku. Doba existence proměnné je určena dobou aktivace tohoto bloku. Je-li ukončeno provádění příkazů bloku, hodnota proměnné zanikne. Typ a počet proměnných daného bloku jsou pevně určeny v deklaraci a nemohou být v průběhu výpočtu změněny. Takovým proměnným říkáme v Pascalu *statické*.

Uvedený způsob práce s proměnnými nemusí vždy vyhovovat našim potřebám. V Pascalu existují i proměnné, jejichž doba existence nesouvisí s dobou aktivace bloků a jejichž počet je určen až v době výpočtu. Těmto proměnným říkáme *dynamické*.



Obr. 11.2 Ukazatel

Programy, které v závislosti na vstupních údajích budují hodnoty vnitřních datových struktur, nemohou bez dynamických proměnných pracovat zcela univerzálně. Užitím statických proměnných je totiž vždy dána maximální velikost údajů, kterou *nesmí* uživatel překročit.

Vezměme např. program SPORT z kap.8. Existuje v něm konstanta MAXSTCISLO omezující velikost pole pro uchování záznamů (v našem případě 100). Rozsáhlejší údaje nelze zpracovávat.

Je však možné vytvořit během činnosti programu předem neomezený počet dynamických proměnných. Je zřejmé, že nemohou být zpřístupňovány identifikátory, kterých je v programu pouze konečné, při psaní programů určené množství. Dále bylo řečeno, že proměnné vznikají a zanikají nezávisle na blokové struktuře programu. Musí proto existovat operace pro vytvoření a zrušení dynamické proměnné.

Dynamická proměnná je v Pascalu zpřístupněna hodnotou jiné proměnné zvláštního datového typu *ukazatel*. Název ukazatel vyjadřuje vlastnost hodnoty datového typu, která ukazuje na dynamickou proměnnou a zpřístupňuje ji. Na obr. 11.2. zpřístupňuje statická proměnná datového typu ukazatel dynamickou proměnnou datového typu *integer*.

Datový typ ukazatel je v Pascalu vázán na datový typ, jehož dynamické proměnné zpřístupňuje. Existuje ukazatel na *integer*, ukazatel na *real*, popř. ukazatel na libovolný jiný jednoduchý i strukturovaný datový typ. Zpřístupňovaný datový typ (říkáme mu *doménový typ*) je svázán s ukazatelem v deklaraci typu ukazatel.

Popis typu ukazatel má tvar

^ *doménový typ*

tedy např.

```
type
  UKNAINTEGER = ^integer;
  ZAZNAM =
    record
      RE, IM: real;
    end;
  UKNAZAZNAM = ^ZAZNAM;
```

Proměnné datového typu ukazatel mohou být deklarovány podobně jako ostatní proměnné.

```
var
  UKAZATEL  : ^integer;
  UKAZATEL1 : UKNAINTEGER;
  UKAZATEL2 : UKNAZAZNAM;
  UKAZATEL3 : UKNAINTEGER;
```

Pokud proměnná datového typu ukazatel zpřístupňuje dynamickou proměnnou, je možné s hodnotou dynamické proměnné pracovat zápisem

proměnná datového typu ukazatel ^

tedy např.

```
UKAZATEL ^      := 1;
UKAZATEL1^      := UKAZATEL1^ + 3;
UKAZATEL2^.RE := 2.36;
```

Zápis zpřístupňující dynamickou proměnnou je totožný se zápisem přístupové proměnné souboru. Podobně jako u souborů, je i zde možné znak ^ nahradit v popisu ukazatelů i zpřístupňování přes ukazatele znakem @.

Dynamické proměnné vznikají vyvoláním standardní procedury *new* (new = nový)

new (*proměnná datového typu ukazatel*)

Tímto zápisem je vytvořena dynamická proměnná takového datového typu, jaký zpřístupňuje zadaná proměnná typu ukazatel (doménového typu ukazatele). Ukazateli je přiřazena hodnota, která ukazuje na nově vytvořenou proměnnou. Pokud ukazatel již zpřístupňoval dynamickou proměnnou, jeho vazba na tuto proměnnou zanikne. Nebyla-li tato dynamická proměnná zpřístupněna jiným ukazatelem, přestane být přístupnou a *nelze s ní dále pracovat.*

Například zápis

new (UKAZATEL);

zajistí vytvoření dynamické proměnné a vazby zobrazené na obr. 11.2. Nově vytvořená dynamická proměnná *nemá přiřazenu hodnotu*. Tu je nutno jí přiřadit jinými prostředky jazyka (přiřazovacím příkazem nebo čtením), např.:

```
UKAZATEL ^ := 1;
read ( UKAZATEL ^);
```

Dynamická proměnná vytvořená procedurou *new* existuje nazávisle na blokové struktuře programu (existuje dokonce i tehdy, zanikne-li hodnota proměnné typu ukazatel, která ji zpřístupňovala). Jedinou možností, jak dynamickou proměnnou během činnosti programu zrušit, je využít standardní proceduru *dispose* (dispose = mít k volnému použití, k dispozici)

dispose (*proměnná datového typu ukazatel*)

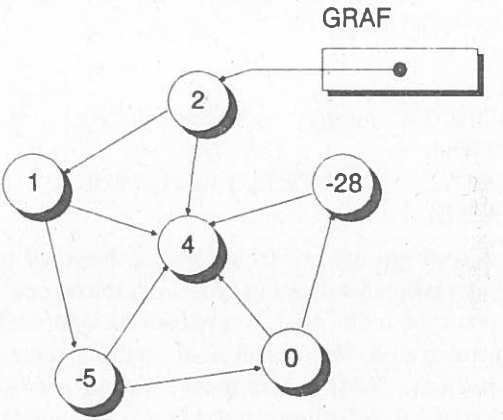
Proměnná typu ukazatel uvedená jako skutečný parametr procedury *dispose musí* zpřístupňovat dynamickou proměnnou. Tato proměnná provedením procedury zanikne. Proceduru *dispose*(UK) je vhodné volat, jen když v celém programu není využívána další proměnná (ať statická či dynamická), která by zpřístupňovala tutéž dynamickou proměnnou jako ukazatel UK.

Správnost volání z tohoto hlediska se obvykle nekontroluje, proto chybné použití *dispose* vede k těžkým chybám při spuštění programu (použití ukazatele na neexistující proměnnou). Odhalení takové chyby může být velmi obtížné !

Ukazatele na týž datový typ lze navzájem přiřazovat a porovnávat na rovnost a nerovnost, např.

```
UKAZATEL1 := UKAZATEL3;
if UKAZATEL1 < > UKAZATEL3 then ...
```

Jak přiřazované, tak porovnávané ukazatele musí být téhož datového typu.



Obr. 11.3 Dynamická datová struktura

Každému ukazateli lze přiřadit hodnotu standardní ukazatelové konstanty **nil** (nil = žádný), která vyjadřuje hodnotu datového typu ukazatel nezpřístupňující žádnou dynamickou proměnnou.

V předchozích příkladech jsme základní vlastnosti ukazatelů předváděli na případech, kdy statická proměnná typu ukazatel zpřístupňovala jedinou dynamickou proměnnou. Takové použití ukazatelů však nemá praktický význam, neboť využitelných dynamických proměnných může být maximálně stejný počet jako deklarovaných proměnných typu ukazatel. Praktického významu nabývají ukazatele tehdy, když dynamické proměnné jsou zpřístupňovány ukazateli v jiných dynamických proměnných. Tak vzniká *dynamická datová struktura*.

Dynamickou datovou strukturu si lze představit jako orientovaný graf (obr. 11.3). Jednotlivé uzly nesou vždy jistou hodnotu a zpřístupňují jiné uzly hod-

notou ukazatelů (na obr. 11.3 vždy dvou ukazatelů). Každý uzel obsahuje proto část hodnotovou, která vyjadřuje vlastní hodnotu uzlu, a část vazební tvořenou hodnotami ukazatelů na jiné uzly.

Proto typickým datovým typem pro definici uzlu dynamické datové struktury je *záznam*. Je to totiž jediný datový typ, který umožňuje popsat hodnotu uzlu skládající se z několika hodnot různých typů (v našem případě hodnoty uzlu a ukazatelů).

Datový typ uzlu z obr. 11.3 lze deklarovat takto:

```
type
  UKUZEL = ^ UZEL;
  UZEL =
    record
      HODNOTA : integer;   { hodnotová část }
      VAZBA1,
      VAZBA2    : UKUZEL;  { vazební část }
    end; { UZEL }
```

(11.1)

Vidíme, že datový typ uzlu obsahuje kromě hodnoty uzlu jeden, popř. několik ukazatelů na svůj vlastní datový typ. Procedurou *new* můžeme vytvořit libovolný počet takových uzlů (např. v závislosti na vstupních údajích) a vzájemně je propojit ukazateli. Počet uzlů není omezen, pouze nelze překročit velikost paměti počítače. Vždy ovšem musí existovat alespoň jedna statická proměnná, která umožní zpřístupnit některý z uzlů struktury (na obr. 11.3 proměnná GRAF).

Bylo řečeno, že každý použitý identifikátor je nutné předem deklarovat. Při deklaraci datové struktury tento požadavek není vždy možné dodržet. Existují dvě možnosti definice typu dynamické datové struktury (11.1) a (11.2).

```
type
UZEL =
  record
    HODNOTA : integer;   { hodnotová část }
    VAZBA1,
    VAZBA2    : UKUZEL;  { vazební část }
  end; { UZEL }
UKUZEL = ^ UZEL;
```

(11.2)

V případě (11.1) použití identifikátoru UZEL předchází jeho deklaraci, podobně v případě (11.2) použití identifikátoru UKUZEL při deklaraci složky rovněž předchází jeho deklaraci.

Stávajícími prostředky nelze tedy typ dynamické datové struktury řádně deklarovat.

Jelikož jde o často používaný zápis deklarace typu, povoluje zde Pascal výjimku. Doménový typ může být deklarován až po deklaraci ukazatele na tento typ, musí však být deklarován ještě v rámci téhož bloku. Proto z předchozích dvou deklarací je (11.1) správná, zatímco (11.2) nesprávná, protože v ní je využíván ukazatel UKUZEL dříve, než byl deklarován, a deklarace ukazatele není umístěna před deklarací doménového typu.

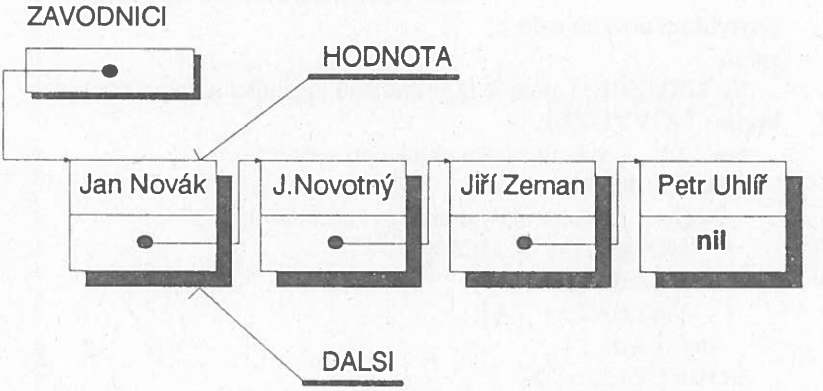
Úsek programu vytvářející graf z obr. 11.3. může mít např. tento tvar (předpokládáme definici typů UKUZEL a UZEL z předchozího odstavce)

```
var
  GRAF : UKUZEL; { proměnná zpřístupňující graf }
  STRED : UKUZEL; { střední uzel s hodnotou 4 }
function NOVYUZEL (HOD: integer;
                   VA1,VA2: UKUZEL):UKUZEL;
{ vytvoření nového uzlu }
var
  X : UKUZEL; { pomocná proměnná ukazující na nový uzel }
begin { NOVYUZEL }
  new (X);  { vytvoření dynamické proměnné }
  with X^ do
    begin  { naplnění dynamické proměnné }
      HODNOTA := HOD;
      VAZBA1 := VA1;
      VAZBA2 := VA2;
    end; { with }
  NOVYUZEL := X;
end; { NOVYUZEL }
begin { tělo programu }
  :
  STRED  := NOVYUZEL (4 , nil, nil);
  GRAF   := NOVYUZEL (-28, STRED, nil);
  GRAF   := NOVYUZEL ( 0, STRED, GRAF);
  GRAF   := NOVYUZEL (-5, STRED, GRAF);
  GRAF   := NOVYUZEL ( 1, STRED, GRAF);
  GRAF   := NOVYUZEL ( 2, STRED, GRAF);
  :
end.
```

Předchozí příklad ukazuje, že je možné deklarovat funkce s výsledkem datového typu ukazatel. Doposud bylo možno deklarovat funkce pouze s výsledkem jednoduchého typu.

Vraťme se nyní k programu SPORT pro vyhodnocování výsledků závodů. Vadí nám, že maximální počet závodníků je omezen deklarovanou velikostí pole (nejvíce 100 složek). Program vyhodnocující libovolný počet závodníků musí proto pracovat s dynamickou datovou strukturou, místo ukládání údajů v poli.

Při návrhu dynamické datové struktury budeme předpokládat, že většina činností s údaji (ukládání i zjišťování údajů) bude probíhat postupně ve směru stoupajících startovních čísel. Dále budeme předpokládat, že položky o jednotlivých závodnících se budou vytvářet zadáváním startovní listiny (tj. jména, příjmení a státní příslušnosti) setříděně podle startovních čísel a další změny ve startovní listině nebudou prováděny.



Obr. 11.4 Jednosměrně vázaný seznam

Vytvořme proto dynamickou strukturu, kde každý uzel má hodnotovou složku typu ZAVODNIK (stejně jako původní pole) a obsahuje vazbu (ukazatel) na uzel následující. Na první uzel v posloupnosti bude ukazovat statická proměnná typu ukazatel, poslední uzel v posloupnosti bude označen hodnotou **nil** vazebního ukazatele. Příklad struktury je uveden na obr. 11.4.

Grafové struktuře takových vlastností říkáme *jednosměrně vázaný seznam*. Její vlastnosti jsou obdobné vlastnostem souboru (uzly lze procházet jen po jednom od začátku do konce). Na rozdíl od souboru je však jednosměrně vázaný

seznam uložen v paměti, a proto jeho zpracování probíhá rychleji (nepoužívá vstupní a výstupní zařízení). Lze jej také přizpůsobovat konkrétním potřebám.

Datový typ uzlu jednosměrně vázaného seznamu můžeme zapsat takto:

```
type
  UKUZEL = ^ UZEL;
  UZEL   =
    record
      HODNOTA : ZAVODNIK; { hodnotová část }
      DALSI   : UKUZEL;   { vazební část }
    end; { UZEL }
```

Celou deklaraci datového typu DATABAZE včetně potřebných konstant můžeme ve formě jednosměrného seznamu zapsat např. takto:

```
const
  MAXSTCISLO = maxint;        { maximální startovní číslo }

type
  NEZAP      = 0 .. maxint; { nezáporné celé číslo }
  TSTCISLA   = 0..MAXSTCISLO;{ typ startovního čísla }
  RET3       = packed array [1.. 3] of char; { řetězec délky 3 }
  RET10      = packed array [1..10] of char; { řetězec délky 10 }
  ZAVODNIK   =
    record    { typ záznamu o závodníkovi }
      STARTCISLO:TSTCISLA;  { startovní číslo }
      JMENO,                { jméno }
      PRIJMENI: RET10;      { příjmení }
      STAT   : RET3;        { státní příslušnost }
      CASSTARTU,            { čas startu }
      VYSLEDNYCAS : NEZAP;  { výsledný čas }
    end; { ZAVODNIK }
  UKUZEL   = ^UZEL;     { ukazatel na uzel seznamu }
  UZEL     =
    record    { typ uzlu jednosměrně vázaného seznamu }
      HODNOTA : ZAVODNIK; { údaje o závodníkovi }
      DALSI   : UKUZEL;   { ukazatel na následníka }
    end; { UZEL }
  DATABAZE =
    record    { typ báze údajů }
      MAXSTCIS : TSTCISLA;{ maximální uvedené startovní číslo }
```

```
    ZAVODNICI  : UKUZEL; { údaje o závodnících }
    { pomocné ukazatele na poslední použitý uzel }
    POSLZAPISJMEN,
    POSLZISKANIJMEN,
    POSLZAPISCS,
    POSLZISKANICS,
    POSLZAPISVC,
    POSLZISKANIVC : UKUZEL;
  end; { DATABAZE }
```

Pro zápis jména, příjmení a státní příslušnosti slouží (jak víme z kap.8) procedura ZAPISJMEN. S využitím nově deklarovaného datového typu DATABAZE bude mít tvar

```
procedure ZAPISJMEN ( var BAZE:DATABAZE;
                      STCISLO:TSTCISLA; JM, PR:RET10; ST: RET3);
{ zápis jména JM, příjmení PR a státní příslušnosti ST startovnímu číslu
STCISLO }
var
  A : UKUZEL; { pomocný ukazatel }
begin { ZAPISJMEN }
  with BAZE do
    begin
      new(A); { nová dynamická proměnná }
      with A^, HODNOTA do
        begin
          STARTCISLO    := STCISLO;
          JMENO         := JM;
          PRIJMENI      := PR;
          STAT          := ST;
          VYSLEDNYCAS := 0;
          DALSI         := nil;
        end; { with A^, HODNOTA }
      if POSLZAPISJMEN = nil then { seznam prázdný }
        ZAVODNICI := A
      else
        POSLZAPISJMEN^.DALSI := A;
      POSLZAPISJMEN := A;
      if MAXSTCIS < STCISLO then
        { nové maximální startovní číslo }
```

```
            MAXSTCIS := STCISLO;
        end; { with BAZE }
    end; { ZAPISJMEN }
```

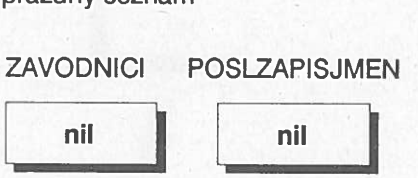
Nejprve je vytvořena nová dynamická proměnná zpřístupněná ukazatelem A. To je provedeno příkazem

new(A);

Do složky dynamické proměnné A^.HODNOTA jsou uloženy zadané údaje o závodníkovi. Pro zkrácení zápisu je použit příkaz with

with A^,HODNOTA **do**

Do vazební složky DALSI je uložena hodnota **nil** (právě vytvářený uzel je v seznamu vždy poslední).



Obr. 11.5a Vytváření uzlů jednosměrného seznamu

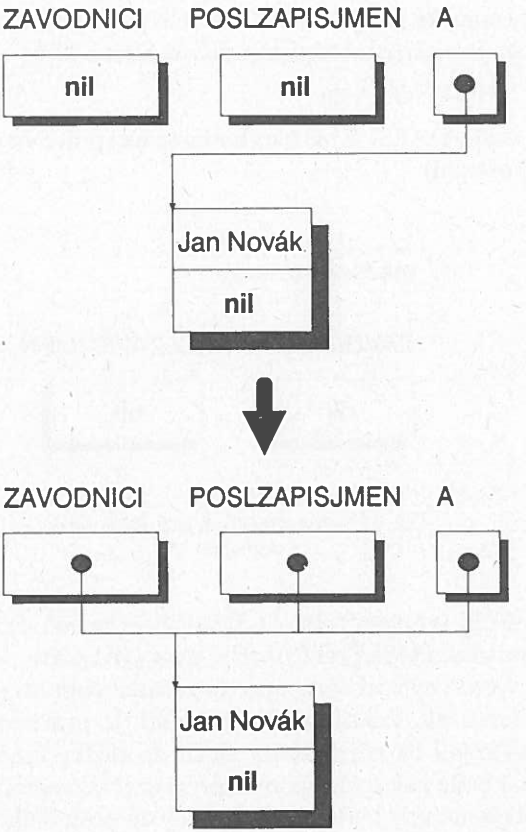
Pro vytváření seznamu (obr. 11.5) slouží pomocná složka záznamu DATABAZE nazvaná POSLZAPISJMEN typu UKUZEL, která ukazuje vždy na poslední vytvořený uzel seznamu. Na začátku činnosti programu ji nastavíme na hodnotu **nil**. Pokud má hodnotu **nil**, je právě vytvářen první uzel seznamu a ukazatel na něj je třeba uložit do složky ZAVODNICI. Složka ZAVODNICI bude pak zpřístupňovat první uzel seznamu. Ukazatele na druhý a další vytvářené uzly budou uloženy vždy do posledního vytvářeného uzlu do složky DALSI.

```
if POSLZAPISJMEN = nil then { seznam prázdný }
  ZAVODNICI := A
else
  POSLZAPISJMEN^.DALSI := A;
```

Teprve potom je ukazatel A na nový uzel uložen do složky POSLZAPISJMEN.

Je také zapsáno do složky MAXSTCIS dosud nejvyšší použité startovní číslo (při zadávání startovní listiny podle startovních čísel je poslední zadané startovní číslo vždy největší).

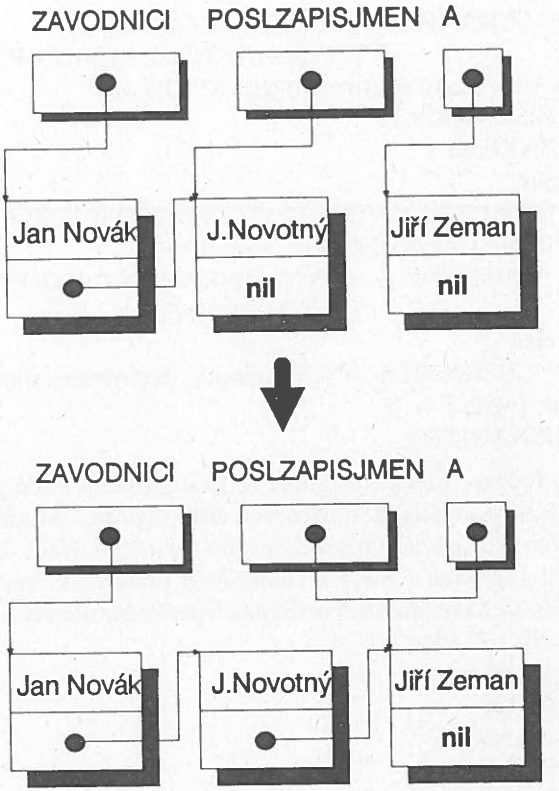
2. vložení prvního uzlu



Obr. 11.5b Vytváření uzlů jednosměrného seznamu

Procedury a funkce ZISKANIJMEN, ZAPISCS, ZISKANICS, ZAPISVC a ZISKANIVC budou mít velmi podobnou stavbu. Zvolme tedy např. proceduru a funkci pro zápis a získání času startu ZAPISCS a ZISKANICS.

3. vložení druhého a dalších uzlů



Obr. 11.5c Vytváření uzlů jednosměrného seznamu

```
procedure ZAPISCS ( var BAZE:DATABAZE; STCISLO:TSTCISLA;
                    CAS:NEZAP);
{ zápis času startu CAS startovnímu číslu STCISLO }
begin { ZAPISCS }
  with BAZE do
    begin
      VYHLEDANI (BAZE, STCISLO, POSLZAPISCS, true);
      if POSLZAPISCS < > nil then
        POSLZAPISCS ^.HODNOTA.CASSTARTU : = CAS;
```

```
        end; { with }
    end; { ZAPISCS }

    function ZISKANICS ( var BAZE:DATABAZE;
                         STCISLO:TSTCISLA):NEZAP;
    { získání času startu startovního čísla STCISLO }
    begin { ZISKANICS }
      with BAZE do
        begin
          VYHLEDANI (BAZE, STCISLO, POSLZISKANICS, false);
          if POSLZISKANICS < > nil then
            ZISKANICS : = POSLZISKANICS ^ .HODNOTA.
                          CASSTARTU
          else
            ZISKANICS : = 0; { hodnota při chybném startovním čísle }
        end; { with }
    end; { ZISKANICS }
```

Jak bylo řečeno, předpokládáme, že práce s údaji bude prováděna převážně ve smyslu stoupajících startovních čísel. Každá z uvedených procedur bude mít proto k dispozici složku datového typu UKUZEL, která bude ukazovat na uzel seznamu, s nímž se naposledy pracovalo. Jména pomocných složek jsou odvozena ze jmen odpovídajících procedur a funkcí uvedením předpony POSL. Jde o složky:

POSLZISKANIJMEN
POSLZAPISCS
POSLZISKANICS
POSLZAPISVC
POSLZISKANIVC

Procedura a funkce ZAPISCS a ZISKANICS budou mít k dispozici pomocné složky POSLZAPISCS a POSLZISKANICS.

Na začátku činnosti programu bude pomocným složkám přiřazena (stejně jako již popsané složce POSLZAPISJMEN) hodnota **nil**.

Prvním příkazem každé z uvedených procedur a funkcí je vyhledání uzlu seznamu s odpovídajícím startovním číslem. K vyhledání slouží procedura VYHLEDANI deklarovaná s hlavičkou

```
procedure VYHLEDANI (var BAZE:DATABAZE;
        STCISLO:TSTCISLA; var POSL: UKUZEL; VYPIS: boolean);
```

Tato procedura na základě zadaného startovního čísla STCISLO a pomocného ukazatele POSL vyhledá v seznamu uzel obsahující údaje o závodníkovi se startovním číslem STCISLO. Pomocný ukazatel POSL upraví tak, aby ukazoval na nalezený uzel. Není-li závodník s příslušným číslem nalezen, je ukazateli POSL přiřazena hodnota **nil**. Parametr VYPIS určuje, zda při nenalezení závodníka se má (VYPIS = *true*) nebo nemá (VYPIS = *false*) vypisovat varovné hlášení

Startovni cislo xx neni v seznamu

Varovné hlášení budeme požadovat vždy při zápisu do báze údajů, když uživatel zadá neexistující startovní číslo.

Prvním příkazem procedur a funkcí ZISKANIJMEN, ZAPISCS, ZISKANICS, ZAPISVC a ZISKANIVC je vyhledání závodníka určeného startovního čísla uvedenou procedurou VYHLEDANI. Další příkazy zapisují, popř. získávají údaje z báze.

Není-li startovní číslo STCISLO nalezeno, není zápis proveden

```
if POSLZAPISCS < > nil then
  POSLZAPISCS ^.HODNOTA.CASSTARTU := CAS;
```

Při získávání údajů je v tomto případě předávána jistá hodnota, která pro okolí funkce znamená, že startovní číslo STCISLO v bázi není. U funkce ZISKANICS je to např. hodnota 0:

```
if POSLZISKANICS < > nil then
  ZISKANICS := POSLZISKANICS ^.HODNOTA.CASSTARTU
else
  ZISKANICS := 0; { hodnota při chybném startovním čísle }
```

U funkce ZISKANIVC je při nenalezení rovněž jako výsledek předávána 0, u procedury ZISKANIJMEN jsou výstupní parametry, obsahující při správné funkci jméno, příjmení a státní příslušnost, zaplněny otazníky.

Procedury pro uložení báze do souboru a přečtení ze souboru (ULOZENI a PRECTENI) využívají podoby dynamické struktury jednosměrně vázaného seznamu a souboru:

```
procedure ULOZENI (var BAZE:DATABAZE; var F:TSOUBORU);
{ uložení obsahu databáze do souboru F }
var
  POM: UKUZEL;          { pomocná proměnná }
begin { ULOZENI }
  assign (F,'datsport');  { příprava pro zápis }
```

```
    rewrite (F);
    POM := BAZE.ZAVODNICI; { začátek seznamu }
    while POM <> nil do      { přes celý seznam }
      begin
        write(F,POM^.HODNOTA); { zápis údaje }
        POM := POM^.DALSI;    { na další }
      end;
    close (F);
  end; { ULOZENI }

  procedure PRECTENI (var BAZE:DATABAZE;
                      var F:TSOUBORU);
  { uložení obsahu souboru F do proměnné BAZE }
  var
    A,
    POSL: UKUZEL;          { pomocné proměnné }
  begin { PRECTENI }
    assign (F,'datsport'); { příprava pro čtení }
    reset (F);
    POCATECNINASTAVENI (BAZE);
    POSL := nil;
    while not eof (F) do   { přes celý soubor }
      begin
        new (A);      { nová dynamická proměnná }
        if POSL = nil then
            BAZE.ZAVODNICI := A { začátek seznamu }
        else
          POSL^.DALSI:=A; { pokračování seznamu }
        POSL := A;
        read(F,A^.HODNOTA); { čtení údaje }
        with BAZE, A^.HODNOTA do { úprava počtu výsledků }
          if (STARTCISLO > MAXSTCIS) then
            BAZE.MAXSTCIS := A^.HODNOTA.STARTCISLO;
      end; { while }
    if POSL <> nil then { alespoň jedna složka se přečetla }
      POSL^.DALSI := nil; { ukončení seznamu hodnotou nil }
  end; { PRECTENI }
```

Procedura ULOZENI prochází jednosměrně vázaný seznam od začátku do konce (konec je označen hodnotou **nil** vazebního ukazatele DALSI). K prů-

chodu seznamem se užívá cyklu while, protože seznam může být prázdný a tělo cyklu se nemusí provést ani jednou. Na uzel seznamu, který se vypisuje do souboru, ukazuje pomocný ukazatel POM. Na začátku je této proměnné přiřazena hodnota ukazatele na první uzel (BAZE.ZAVODNICI). Po přenesení jedné složky je hodnota ukazatele přesunuta na další uzel

```
POM := POM^.DALSI;
```

Obdobně i procedura PRECTENI prochází všemi složkami souboru (cyklem while) a jednu po druhé ji přepisuje do báze. Je nutné rozlišit (podobně jako v proceduře ZAPISJMEN), zda je v seznamu vytvářen první, druhý nebo další uzel (obr. 11.5, pouze POSLZAPISJMEN je nahrazeno jménem lokální proměnné POSL). Na první uzel ukazuje složka BAZE.ZAVODNICI, na další uzly ukazují již vazební složky DALSI předchozích dynamických proměnných.

```
if POSL = nil then
  BAZE.ZAVODNICI := A   { začátek seznamu }
else
  POSL^.DALSI:=A;       { pokračování seznamu }
POSL := A;
```

Prvním průchodem je hodnota pomocné proměnné POSL nastavena na počáteční hodnotu **nil**.

Procedura využívá dvou pomocných proměnných A a POSL. Proměnná A ukazuje vždy na nově vytvořený uzel a proměnná POSL na uzel předchozí. Ukazatel na předchozí uzel je nutný pro vytvoření vazebního ukazatele

```
POSL^.DALSI:=A; { pokračování seznamu }
```

Během vytváření seznamu se průběžně aktualizuje hodnota maximálního použitého startovního čísla

```
with BAZE, A^.HODNOTA do { úprava počtu výsledků }
  if (STARTCISLO > MAXSTCIS) then
    BAZE.MAXSTCIS := A^.HODNOTA.STARTCISLO;
```

Je-li vytváření seznamu ukončeno, je do poslední vazební složky (pokud byl vytvořen alespoň jeden uzel) přiřazena hodnota **nil**.

```
if POSL <> nil then      { alespoň jedna složka se přečetla }
  POSL^.DALSI := nil; { ukončení seznamu hodnotou nil }
```

Procedura ZISKANIMAXCISLA pro získání maximálního startovního čísla se z minulé verze programu nezměnila. Zbývá procedura pro počáteční nastavení hodnot v bázi POCATECNINASTAVENI.

```
procedure POCATECNINASTAVENI (var BAZE:DATABAZE);
{ počáteční nastavení }
begin { POCATECNINASTAVENI }
  with BAZE do
    begin
      MAXSTCIS          := 0; { maximální startovní číslo }
      ZAVODNICI         := nil; { údaje o závodnících }
      POSLZAPISJMEN     := nil;
      POSLZISKANIJMEN   := nil;
      POSLZAPISCS       := nil;
      POSLZISKANICS     := nil;
      POSLZAPISVC       := nil;
      POSLZISKANIVC     := nil;
    end; { with BAZE }
end; { POCATECNINASTAVENI }
```

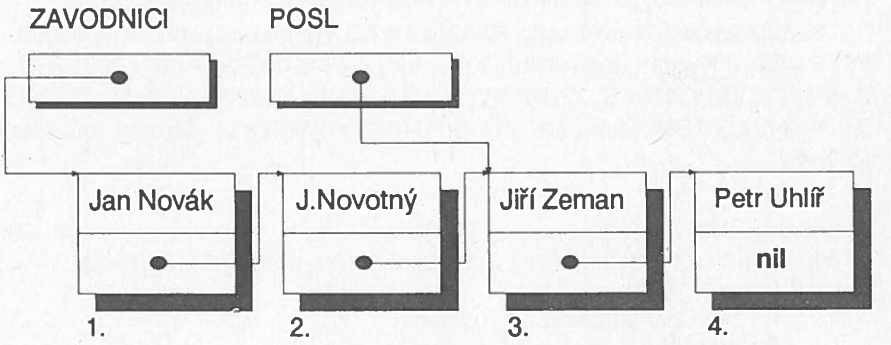
Tato procedura nastaví hodnotu maximálního použitého startovního čísla na 0 a přiřadí hodnoty **nil** všem ukazatelům.

Zbývá deklarovat proceduru VYHLEDANI.

```
procedure VYHLEDANI (var BAZE:DATABAZE;
        STCISLO:TSTCISLA; var POSL: UKUZEL; VYPIS: boolean);
{ vyhledání podle zadaného startovního čísla }
var
  NALEZEN : boolean;   { nalezení shody }
begin { VYHLEDANI }
  with BAZE do
    begin
      if POSL = nil then { POSL ještě nebyl použit }
        POSL := ZAVODNICI    { bude ukazovat na první }
      else if POSL^.HODNOTA.STARTCISLO >= STCISLO then
      { hledané číslo není směrem ke konci, vyhledává se od začátku }
        POSL := ZAVODNICI
      else { hledané číslo je směrem ke konci, vyhledává se od POSL }
        POSL := POSL^.DALSI;
      NALEZEN := false; { příznak nalezení }
      while (not NALEZEN) and (POSL <> nil) do
        begin { vyhledávání }
          NALEZEN :=
            POSL^.HODNOTA.STARTCISLO = STCISLO;
```

```
            if not NALEZEN then
              POSL := POSL^.DALSI;
          end; { vyhledávání }
        { je-li dosaženo konce seznamu, je hodnota POSL = nil }
        if (POSL = nil) and VYPIS then
          writeln('Startovni cislo ',STCISLO:1,' neni v seznamu');
      end; { with }
    end; { VYHLEDANI }
```

Procedura vyhledává v jednosměrně vázaném seznamu uzel s odpovídajícím startovním číslem závodníka. Na poslední použitý uzel ukazuje parametr procedury POSL (obr. 11.6).



Obr. 11.6 Vyhledávání v seznamu

Mohou nastat tři případy:

- POSL = **nil** , tj. POSL nebyl dosud použit a musí se vyhledávat od začátku seznamu (do POSL se přiřadí BAZE.ZAVODNICI),
- zadané startovní číslo STCISLO není větší než poslední použité startovní číslo POSL^.HODNOTA.STARTCISLO, tzn. že zpracování neprobíhá ve směru stoupajících startovních čísel a bude se také vyhledávat od začátku (POSL := BAZE.ZAVODNICI),
- nenastal-li žádný z předchozích případů, je vše v pořádku (startovní číslo je větší než minulé použité) a vyhledává se od uzlu zpřístupněného ukazatelem POSL. Předpokládá se, že tento případ bude nejčastější.

Po provedení úvodního rozhodnutí (odkud se bude hledat) probíhá vyhledávání cyklem while. Cyklus je ukočen nalezením odpovídajícího uzlu

POSL^.HODNOTA.STARTCISLO = STCISLO

popř. dosažením konce seznamu

POSL = **nil**

Nalezení uzlu s odpovídajícím startovním číslem závodníka je vyjádřeno hodnotou *true* proměnné NALEZEN. Je-li konce seznamu dosaženo bez nalezení vhodného uzlu a je požadováno varovné hlášení (VYPIS = *true*), je proveden příkaz

writeln('Startovni cislo ',STCISLO:1,' neni v seznamu');

Tím, že při nenalezení vhodného uzlu má parametr POSL hodnotu **nil** (na konci seznamu), je nenalezení uzlu také oznámeno mimo proceduru.

Kromě úpravy datového typu DATABAZE a deklarací procedur a funkcí, které s tímto typem bezprostředně pracují (ZAPISJMEN, ZISKANIJMEN, ZAPISCS, ZISKANICS, ZAPISVC, ZISKANIVC, ULOZENI, PRECTENI, ZISKANIMAXCISLA a POCATECNINASTAVENI), se původní program nezměnil.

```
program SPORT (input, output, DATSPORT);
{ Úvodní komentář je totožný s programem 8.1, pouze jsou doplněny
nové činnosti U a P.
      U ... uložení databáze do souboru
      P ... přečtení databáze ze souboru
}
const
   MAXSTCISLO = maxint;          { maximální startovní číslo }
   VELIKOSTOBRAZOVKY = 24; { počet řádků na obrazovce }

type
   NEZAP          = 0 .. maxint; { nezáporné celé číslo }
   TSTCISLA       = 0..MAXSTCISLO; { typ startovního čísla }
   RET3           = packed array [1.. 3] of char; { řetězec délky 3 }
   RET10          = packed array [1..10] of char; { řetězec délky 10 }
   ZAVODNIK       =
      record     { typ záznamu o závodníkovi }
         STARTCISLO:TSTCISLA;  { startovní číslo }
         JMENO,                { jméno }
         PRIJMENI: RET10;      { příjmení }
         STAT  : RET3;         { státní příslušnost }
```

```
      CASSTARTU,                { čas startu }
      VYSLEDNYCAS : NEZAP; { výsledný čas }
    end; { ZAVODNIK }
  UKUZEL   = ^UZEL;           { ukazatel na uzel seznamu }
  UZEL     =
    record     { typ uzlu jednosměrně vázaného seznamu }
      HODNOTA : ZAVODNIK; { údaje o závodníkovi }
      DALSI : UKUZEL;          { ukazatel na následníka }
    end; { UZEL }
  DATABAZE   =
    record     { typ báze údajů }
      MAXSTCIS : TSTCISLA;   { maximální uvedené startovní číslo }
      ZAVODNICI : UKUZEL;    { údaje o závodnících }
        { pomocné ukazatele na naposledy použitý uzel }
      POSLZAPISJMEN,
      POSLZISKANIJMEN,
      POSLZAPISCS,
      POSLZISKANICS,
      POSLZAPISVC,
      POSLZISKANIVC : UKUZEL;
    end; { DATABAZE }
  TSOUBORU   = file of ZAVODNIK; { soubor pro ukládání }
  TCINNOSTI = (CHYBNYKOD,ZSTARTU,ZCILE,VYSLEDKY,
          ZJMENA,ZULOZENI,ZPRECTENI,ZLISTINA,KONEC);
                          { typ kódů činností programu }

var
  DB        : DATABAZE;    { báze údajů }
  CYKLUJ : boolean; { proměnná určující dobu činnosti programu }
  DATSPORT    : TSOUBORU;    { soubor pro uložení údajů }

{ procedury pro práci s bází údajů }

procedure VYHLEDANI(var BAZE:DATABAZE;
                    STCISLO:TSTCISLA;
                    var POSL: UKUZEL;
                    VYPIS: boolean);
{ vyhledání podle zadaného startovního čísla }
var
  NALEZEN : boolean;   { nalezení shody }
begin { VYHLEDANI }
  with BAZE do
    begin
```

```
        if POSL = nil then { POSL ještě nebyl použit }
          POSL := ZAVODNICI   { bude ukazovat na první }
        else if POSL^.HODNOTA.STARTCISLO >= STCISLO then
          { hledané číslo není směrem ke konci,
          vyhledává se od začátku }
          POSL := ZAVODNICI
        else { hledané číslo je směrem ke konci,
            vyhledává se od POSL }
          POSL := POSL^.DALSI;
          NALEZEN := false; { příznak nalezení }
        while (not NALEZEN) and (POSL <> nil) do
          begin { vyhledávání }
            NALEZEN :=
              POSL^.HODNOTA.STARTCISLO = STCISLO;
            if not NALEZEN then
              POSL := POSL^.DALSI;
          end; { vyhledávání }
        { je-li dosaženo konce seznamu, je hodnota POSL = nil }
        if (POSL = nil) and VYPIS then
          writeln('Startovni cislo ',STCISLO:1,' neni v seznamu');
      end; { with }
end; { VYHLEDANI }

procedure ZAPISCS ( var BAZE:DATABAZE;STCISLO:TSTCISLA;
                    CAS:NEZAP);
{ zápis času startu CAS startovnímu číslu STCISLO }
begin { ZAPISCS }
  with BAZE do
    begin
      VYHLEDANI (BAZE, STCISLO, POSLZAPISCS, true);
      if POSLZAPISCS <> nil then
        POSLZAPISCS^.HODNOTA.CASSTARTU := CAS;
    end; { with }
end; { ZAPISCS }

function ZISKANICS ( var BAZE:DATABAZE;STCISLO:TSTCISLA)
                     :NEZAP;
{ získání času startu startovního čísla STCISLO }
begin { ZISKANICS }
  with BAZE do
    begin
      VYHLEDANI (BAZE, STCISLO, POSLZISKANICS, false);
      if POSLZISKANICS <> nil then
```

```
            ZISKANICS :=
              POSLZISKANICS^.HODNOTA.CASSTARTU
          else
            ZISKANICS := 0; { hodnota při chybném startovním čísle }
        end; { with }
end; { ZISKANICS }

procedure ZAPISVC ( var BAZE:DATABAZE;STCISLO:TSTCISLA;
                    CAS:NEZAP);
{ zápis výsledného času CAS startovnímu číslu STCISLO }
begin { ZAPISVC }
  with BAZE do
    begin
      VYHLEDANI (BAZE, STCISLO, POSLZAPISVC, true);
      if POSLZAPISVC <> nil then
          POSLZAPISVC^.HODNOTA.VYSLEDNYCAS := CAS;
    end; { with }
end; { ZAPISVC }

function ZISKANIVC ( var BAZE:DATABAZE;STCISLO:TSTCISLA)
                   :NEZAP;
{ získání výsledného času startovního čísla STCISLO }
begin { ZISKANIVC }
  with BAZE do
    begin
      VYHLEDANI (BAZE, STCISLO, POSLZISKANIVC, false);
      if POSLZISKANIVC <> nil then
        ZISKANIVC :=
          POSLZISKANIVC^.HODNOTA.VYSLEDNYCAS
      else
        ZISKANIVC := 0; { hodnota při chybném startovním čísle }
    end; { with }
end; { ZISKANIVC }

procedure ZAPISJMEN(var BAZE:DATABAZE;STCISLO:TSTCISLA;
                          JM, PR:RET10; ST: RET3);
{ zápis jména JM, příjmení PR a státu ST startovnímu číslu STCISLO }
var
  A : UKUZEL; { pomocný ukazatel }
begin { ZAPISJMEN }
  with BAZE do
    begin
      new(A); { nová dynamická proměnná }
```

```
        with A^, HODNOTA do
          begin
            STARTCISLO      := STCISLO;
            JMENO           := JM;
            PRIJMENI        := PR;
            STAT            := ST;
            VYSLEDNYCAS     := 0;
            DALSI           := nil;
          end; { with A^, HODNOTA }
        if POSLZAPISJMEN = nil then { seznam prázdný }
          ZAVODNICI := A
        else
          POSLZAPISJMEN^.DALSI := A;
        POSLZAPISJMEN := A;
        if MAXSTCIS < STCISLO then
          { nové maximální startovní číslo }
          MAXSTCIS := STCISLO;
      end; { with BAZE }
end; { ZAPISJMEN }

procedure ZISKANIJMEN ( var BAZE:DATABAZE;
                        STCISLO:TSTCISLA;
                        var JM, PR:RET10; var ST: RET3);
{ získání jména JM, příjmení PR a státu ST startovního čísla STCISLO }
begin { ZISKANIJMEN }
  with BAZE do
    begin
      VYHLEDANI (BAZE, STCISLO, POSLZISKANIJMEN, false);
      if POSLZISKANIJMEN <> nil then
        with POSLZISKANIJMEN^.HODNOTA do
          begin   { získání jména příjmení a státní příslušnosti }
            JM  := JMENO ;
            PR  := PRIJMENI ;
            ST  := STAT ;
          end { with POSLZISKANIJMEN^.HODNOTA }
      else
        begin { hodnoty při nenalezení }
          JM := '??????????';
          PR := '??????????';
          ST := '???';
        end; { hodnoty při nenalezení }
    end; { with BAZE }
end; { ZISKANIJMEN }
```

```
function ZISKANIMAXCISLA(var BAZE:DATABAZE ): TSTCISLA;
{ získání maximálního startovního čísla }
begin { ZISKANIMAXCISLA }
   ZISKANIMAXCISLA := BAZE.MAXSTCIS;
end; { ZISKANIMAXCISLA }

procedure POCATECNINASTAVENI (var BAZE:DATABAZE);
{ počáteční nastavení výsledných časů }
begin { POCATECNINASTAVENI }
   with BAZE do
      begin
         MAXSTCIS          := 0; { maximální startovní číslo }
         ZAVODNICI         := nil; { údaje o závodnících }
         POSLZAPISJMEN     := nil;
         POSLZISKANIJMEN   := nil;
         POSLZAPISCS       := nil;
         POSLZISKANICS     := nil;
         POSLZAPISVC       := nil;
         POSLZISKANIVC     := nil;
      end; { with BAZE }
end; { POCATECNINASTAVENI }

procedure ULOZENI (var BAZE:DATABAZE; var F:TSOUBORU);
{ uložení obsahu databáze do souboru F }
var
   POM: UKUZEL;     { pomocná proměnná }
begin { ULOZENI }
   assign (F,'datsport'); { příprava pro zápis }
   rewrite (F);
   POM := BAZE.ZAVODNICI; { začátek seznamu }
   while POM <> nil do  { přes celý seznam }
      begin
         write(F,POM^.HODNOTA); { zápis údaje }
         POM := POM^.DALSI;    { na další }
      end;
   close (F);
end; { ULOZENI }

procedure PRECTENI (var BAZE:DATABAZE; var F:TSOUBORU);
{ uložení obsahu souboru F do proměnné BAZE }
var
   A,
```

```
    POSL: UKUZEL;        { pomocné proměnné }
begin { PRECTENI }
    assign (F,'datsport'); { příprava pro čtení }
    reset (F);
    POCATECNINASTAVENI (BAZE);
    POSL := nil;
    while not eof (F) do   { přes celý soubor }
      begin
        new (A);       { nová dynamická proměnná }
        if POSL = nil then
            BAZE.ZAVODNICI := A { začátek seznamu }
        else
          POSL^.DALSI:=A; { pokračování seznamu }
        POSL := A;
        read(F,A^.HODNOTA); { čtení údaje }
        with BAZE, A^.HODNOTA do { úprava počtu výsledků }
          if (STARTCISLO > MAXSTCIS) then
            BAZE.MAXSTCIS := A^.HODNOTA.STARTCISLO;
      end; { while }
    if POSL <> nil then { alespoň jedna složka se přečetla }
      POSL^.DALSI := nil; { ukončení seznamu hodnotou nil }
end; { PRECTENI }

{ ostatní procedury

Procedury a funkce CTENICASU, ZADSCASU, ZADCCILE,
VYPVYSLEDKU, ZADJMENA a LISTINA mají deklarace totožné
s programem 8.1. Při vytváření spustitelného programu je proto nutné
následující hlavičky nahradit úplnými deklaracemi z programu 8.1.
Zde jsou bloky funkcí a procedur pro úsporu místa vynechány.
}
function CTENICASU :NEZAP;
procedure ZADSCASU (var BAZE:DATABAZE);
procedure ZADCCILE (var BAZE:DATABAZE);
procedure VYPVYSLEDKU (var BAZE: DATABAZE);
procedure ZADJMENA ( var BAZE:DATABAZE );
procedure LISTINA (var BAZE: DATABAZE);

function URCENITYPU: TCINNOSTI;
{ určení typu činnosti, která se bude provádět }
var
  ZNAK              :char;  { pracovní proměnné }
```

```
    DALSICINNOST : TCINNOSTI;
  begin { URCENITYPU }
    repeat
      writeln('Zadej kod cinnosti');
      readln(ZNAK);
      case ZNAK of
        's','S'   : DALSICINNOST  := ZSTARTU;
        'c','C'   : DALSICINNOST  := ZCILE;
        'v','V'   : DALSICINNOST  := VYSLEDKY;
        'j','J'   : DALSICINNOST  := ZJMENA;
        'u','U'   : DALSICINNOST  := ZULOZENI;
        'p','P'   : DALSICINNOST  := ZPRECTENI;
        'l','L'   : DALSICINNOST  := ZLISTINA;
        'k','K'   : DALSICINNOST  := KONEC;
        else
          begin { chybný kód }
            writeln('Chybny kod cinnosti');
            writeln;
            writeln('S ... cas startu');
            writeln('C ... cas v cili');
            writeln('V ... vysledkova listina');
            writeln('J ... zadani jmena, prijmeni a statu');
            writeln('U ... ulozeni databaze do souboru');
            writeln('P ... precteni databaze ze souboru');
            writeln('K ... konec programu');
            writeln;
            DALSICINNOST := CHYBNYKOD;
          end; { chybný kód }
      end; { case }
    until DALSICINNOST <> CHYBNYKOD;
    URCENITYPU := DALSICINNOST;
  end; { URCENITYPU }

  begin { SPORT }
    writeln('VYPOCET VYSLEDKU');
    writeln;
    POCATECNINASTAVENI(DB);
    CYKLUJ:=true;  { příznak nastavit na režim práce }
    repeat
      case URCENITYPU { určení typu činnosti } of
        ZSTARTU     : ZADSCASU (DB); { zadání startovního času }
        ZCILE       : ZADCCILE (DB); { zadání času v cíli }
        VYSLEDKY    : VYPVYSLEDKU (DB); { výsledková listina }
```

```
        ZJMENA      : ZADJMENA (DB); { zadání jména a státu }
        ZULOZENI    : ULOZENI (DB,DATSPORT);
                      { uložení do souboru }
        ZPRECTENI   : PRECTENI (DB,DATSPORT);
                      { přečtení ze souboru }
        ZLISTINA    : LISTINA (DB); { startovní listina }
        KONEC       : CYKLUJ := false;{ konec programu }
    end; { case }
  until not CYKLUJ;
  writeln('Konec programu SPORT');
end. { SPORT }
```

Program 11.1

Dynamické proměnné se obvykle přidělují z konečného paměťového prostoru. Část paměti, vrácená procedurou *dispose* k dalšímu volnému využití, může být znovu přidělena procedurou *new* jiné dynamické proměnné. Po ukončení programu všechny dynamické proměnné zaniknou. Proto v případě, že prostor pro dynamické proměnné není již potřeba, není nutné využívat procedury *dispose*.

11.3 Abstraktní datové typy

Vraťme se ještě jednou k předchozímu popisu změn v programu SPORT. Přestože došlo ke změně základního datového typu DATABAZE, zůstala značná část programu beze změny. Modifikace datového typu DATABAZE se odrazila pouze do procedur a funkcí pracujících s datovým typem DATABAZE (ZAPISCS, ZAPISVC, ZISKANICS, ZISKANIVC, ZAPISJMEN, ZISKANIJMEN, ULOZENI, PRECTENI, ZISKANIMAXCISLA a POCATECNINASTAVENI).

Tyto procedury zbavují zbytek programu nutnosti znát vnitřní stavbu datového typu DATABAZE a přístup do databáze převádí na množinu volání ovládacích procedur a funkcí. Při modifikaci datového typu se kromě příslušného datového typu změní jen deklarace ovládacích procedur. Zbytek programu se nemění. Takto deklarovaný a využívaný datový typ je z hlediska modifikace i ladění programu velmi výhodný a říká se mu *abstraktní datový typ*.

Jednoduché datové typy byly deklarovány množinou hodnot a operacemi nad těmito hodnotami. Vnitřní reprezentaci jednoduchých datových typů při psaní programů nemusíme znát (a ani ji většinou neznáme). Máme definovánu pouze množinu operací a jméno datového typu, např. *integer*. Při změně vnitřní stavby typu (např. při přenosu programu na jiný počítač) nám překladač zajistí,

že operace stejného označení budou pracovat s jinou vnitřní stavbou typu a program se z našeho hlediska nezmění. U strukturovaných datových typů vnitřní stavbu datového typu neurčuje překladač, ale tvůrce programu. Datový typ je v programu *konkrétně* deklarován. Například

```
type
  DATABAZE  =
    record     { typ báze údajů }
      :
    end; { DATABAZE }
```

Známe tedy nejen jméno typu DATABAZE, ale i jeho vnitřní stavbu (zde konkrétně jde o jednosměrně vázaný seznam).

Tvůrce programu také může určit konečnou množinu operací, která se s datovým typem bude provádět. Například s datovým typem DATABAZE souvisí 10 operací:

- *zápis* a *získání* času startu,
- *zápis* a *získání* výsledného času,
- *zápis* a *získání* jména, příjmení a státní příslušnosti,
- *uložení* do souboru a *přečtení* ze souboru,
- *získání* maximálního startovního čísla,
- *počáteční nastavení*.

Na rozdíl od jednoduchého datového typu máme u datové struktury dvě možnosti realizace operací. V prvním případě bude uživatel datového typu znát konkrétní vnitřní stavbu typu (z jeho hlediska jde o *konkrétní datový typ*) a programovat operace přímo se znalostí konkrétního typu, např. (DB je proměnná typu DATABAZE):

```
DB.POSLZAPISCS^.HODNOTA.CASSTARTU := 6583;
I := DB.POSLZISKANICS^.HODNOTA.CASSTARTU;
```

Jde-li o strukturovaný datový typ v programu často používaný, pak jsou konkrétní metody přístupu ke složkám rozmístěny po celém programu. Při nutnosti změny datového typu je třeba celý program přepsat. Ladění datového typu musí také probíhat současně s laděním zbytku programu.

V druhém případě nebude uživatel znát vnitřní stavbu typu, ale bude pracovat pouze s identifikátorem typu a množinou operací povolených nad tímto typem ve formě volání procedur a funkcí.

Uživatel nezná vnitřní strukturu typu, ani těla operací. Užívá *abstraktní datový typ*. To ovšem nemusí vadit. Zná-li identifikátory typů, může deklarovat

proměnné i parametry funkcí a procedur. Zná-li hlavičky procedur a funkcí, může využívat jejich volání. Například

```
var
  DB: DATABAZE;
begin
  ZAPISCS (DB,CISLO,6583);
  ZISKANICS (DB,CISLO,I);
  :
```

Tak je dána jasná hranice mezi abstraktním datovým typem a zbytkem programu. Tvoří ji hlavičky ovládacích operací. Zbytek programu nezná vnitřní stavbu typu a nevyužívá ji. Změna struktury typu proto neovlivní zbytek programu. Operace pracující s typem lze ladit zvlášť pomocí vhodného testovacího programu a zcela odladěné využít v programu. Snadno lze (jak jsme viděli) také zaměnit popis abstraktního datového typu za jiný.

Provádíme-li návrh programu metodou shora dolů, je využívání abstraktních datových typů výhodné. Umožňuje nám oddělit úvahy o vnitřní stavbě datového typu (abstraktního) od ostatních úvah (podobně jako jsme u návrhu programu SPORT oddělili úvahy o algoritmech a datový typ databáze popsali až nakonec) a také vhodně rozložit problém na dílčí podproblémy.

Řešíme-li program metodou shora dolů a vlastní realizaci datového typu odložíme až na konec, získáme během řešení přehled o tom, které operace nad datovým typem budeme potřebovat a jaké parametry musí mít.

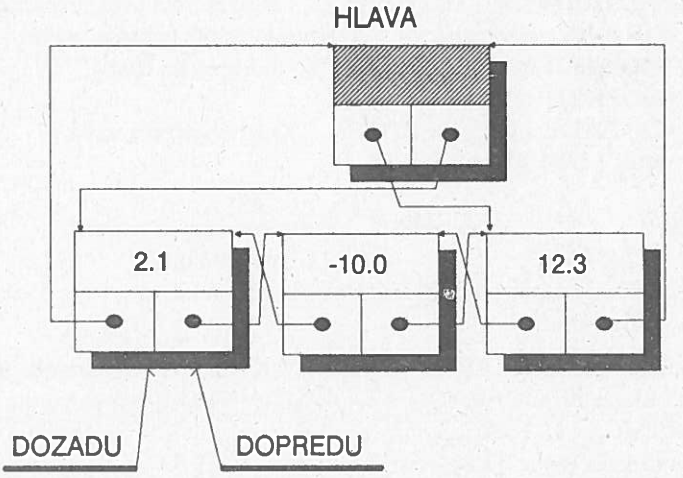
11.4 Editor

Dynamickou strukturou podobnou jednosměrně vázanému seznamu je *dvousměrně vázaný seznam* (obr. 11.7). Na rozdíl od jednosměrně vázaného seznamu obsahuje každý uzel dvousměrně vázaného seznamu dvě vazby. První vazba (DOZADU) ukazuje na předchozí uzel v posloupnosti a druhá (DOPREDU) na uzel následující. To umožňuje průchod seznamem nejen směrem od začátku ke konci (vpřed), ale i směrem od konce k začátku (vzad). Pokud bychom realizovali dvousměrně vázaný seznam ve formě abstraktního datového typu, mohl by být definován např. takto:

```
type
  DVS           = ... ;  { dvousměrně vázaný seznam }
  TYPHODNOTY    = ... ;  { typ hodnoty uchovávané v uzlu }
```

```
procedure PRIPRAVA (var X: DVS);
{ nastavení počáteční hodnoty seznamu }
function JEKONEC (var X:DVS): boolean;
{ test konce seznamu }
procedure VLOZ (var X: DVS; H: TYPHODNOTY);
{ vložení nového uzlu do seznamu }
procedure ZRUS (var X: DVS;);
{ zrušení aktuálního uzlu seznamu }
procedure ZACATEK (var X: DVS);
{ přesun aktuálního uzlu na začátek seznamu }
procedure VPRED (var X: DVS);
{ posun aktuálního uzlu směrem ke konci seznamu }
procedure VZAD (var X: DVS);
{ posun aktuálního uzlu směrem k začátku seznamu }
function CTI (var X: DVS; var H : TYPHODNOTY): boolean;
{ čtení hodnoty aktuálního uzlu }
procedure PIS (var X: DVS; H : TYPHODNOTY);
{ zápis hodnoty do aktuálního uzlu }
```

Typ DVS je vlastní dvousměrně vázaný seznam. TYPHODNOTY definu-



Obr. 11.7 Dvousměrně vázaný seznam

je hodnotu uchovávanou v hodnotové části uzlu (na obr. 11.7 se předpokládá typ hodnoty *integer*).

V seznamu existuje speciální uzel HLAVA, který je vytvořen před začátkem práce se seznamem a není nikdy zrušen. Vazby uzlu HLAVA ukazují na první a poslední uzel vlastního seznamu. Hodnotová část uzlu HLAVA se nevyužívá.

Seznam je navržen tak, aby vždy jeden z uzlů byl aktuální (podobně jako přístupová proměnná u souborů). Z aktuálního uzlu je možné číst hodnotu uzlu operací CTI a zapisovat ji operací PIS. Vzhledem k tomu, že seznam může být při čtení i prázdný, nemusí být vždy možné provést operaci CTI tak, aby skutečně přečetla hodnotu aktuálního uzlu. Proto jde o funkci. Existuje-li aktuální uzel a čtení proběhne řádně, má funkce CTI hodnotu *true*, v opačném případě má hodnotu *false*.

Za aktuální uzel je možné vkládat nový uzel operací VLOZ. Aktuální uzel zrušíme operací ZRUS, měnit jej lze těmito operacemi:

Operací ZACATEK se aktuálním uzlem stane první uzel. Operace VPRED, resp. VZAD posouvají aktuální uzel směrem ke konci, resp. k začátku seznamu. Operace JEKONEC zjistí, je-li aktuálním poslední uzel v seznamu.

Datový typ DVS dvousměrně vázaného seznamu může mít např. deklaraci:

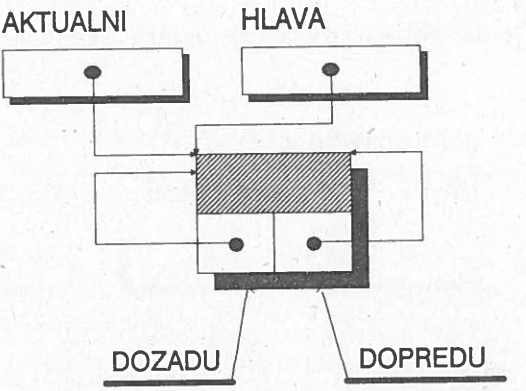
```
type
  UKUZELDVS = ^UZELDVS;  { ukazatel na uzel seznamu }
  UZELDVS   = record     { uzel dvousměrného seznamu }
    HODNOTA : TYPHODNOTY;{ hodnotová část }
    DOPREDU,
    DOZADU : UKUZELDVS;    { vazba vpřed a vzad }
    end; { UZELDVS }

  DVS             = record      { dvousměrný seznam }
    HLAVA ,                     { hlava seznamu }
    AKTUALNI: UKUZELDVS;  { aktuální uzel }
    end; { DVS }
```

Typ uzlu seznamu UZELDVS má obvyklý tvar uzlu dynamické struktury se dvěma ukazateli DOPREDU a DOZADU. Vlastní typ seznamu DVS je tvořen záznamem, v němž jsou obsaženy ukazatele na hlavu a aktuální uzel. Je-li seznam prázdný (obr. 11.8), existuje pouze uzel HLAVA. Vazební ukazatele tohoto uzlu ukazují na sebe sama. Na uzel HLAVA ukazuje na počátku i ukazatel AKTUALNI.

Procedura PRIPRAVA připraví seznam pro další činnost. Bude volána

před započetím práce se seznamem. Vytvoří hlavu seznamu a nastaví počáteční hodnoty ukazatelů HLAVA a AKTUALNI.



Obr. 11.8 Prázdný dvousměrně vázaný seznam

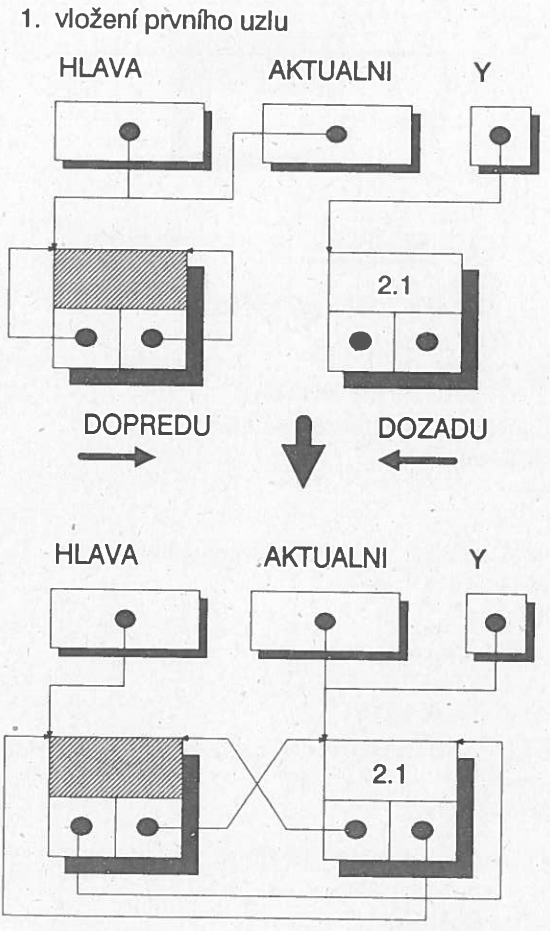
```
procedure PRIPRAVA (var X: DVS);
{ nastavení počáteční hodnoty seznamu }
begin { PRIPRAVA }
  with X do
    begin
      new(HLAVA); { vytvoření pomocného uzlu }
      with HLAVA^ do
        begin
          DOPREDU:=HLAVA;
          DOZADU:=HLAVA;
        end; { with HLAVA }
      AKTUALNI := HLAVA; { aktuální uzel neexistuje }
    end; { with }
end; { PRIPRAVA }
```

Funkce JEKONEC testuje, zda aktuální uzel je v seznamu poslední. Lze to poznat podle hodnoty vazebního ukazatele DOPREDU (musí být rovna hodnotě ukazatele HLAVA).

```
function JEKONEC (var X: DVS): boolean;
{ test existence následujícího uzlu }
```

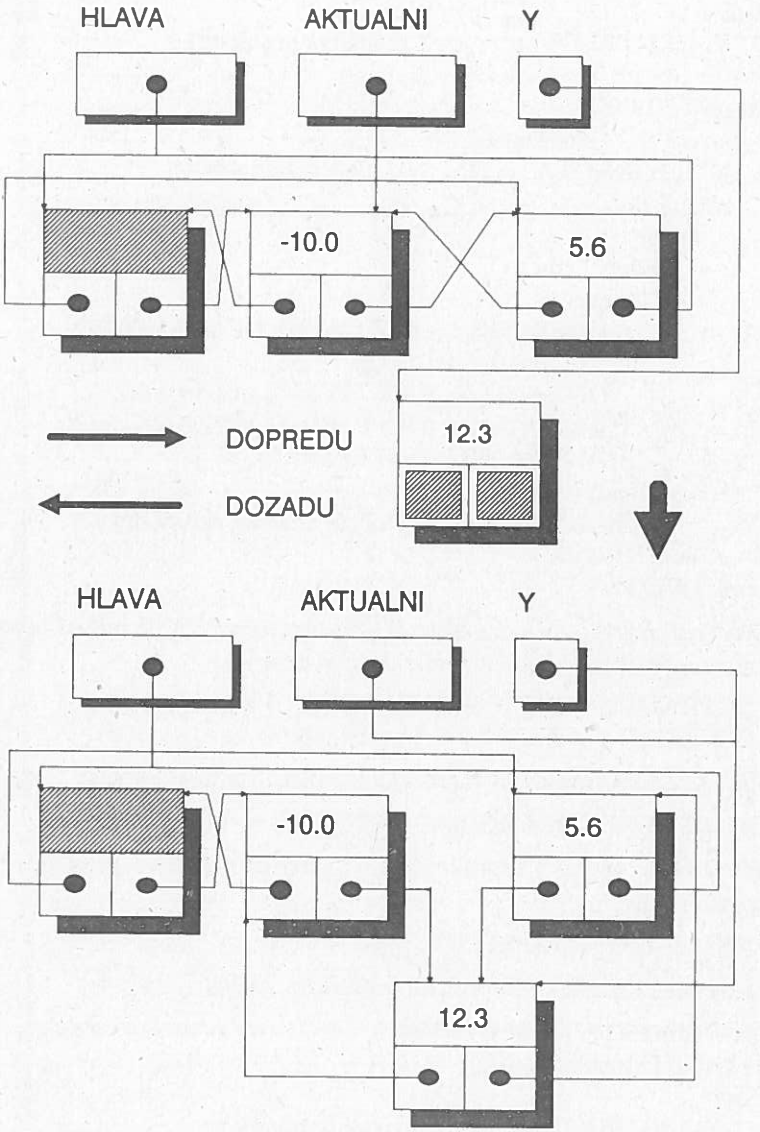
```
begin { JEKONEC }
  with X do
    JEKONEC := AKTUALNI^.DOPREDU = HLAVA;
end; { JEKONEC }
```

Procedura VLOZ vkládá nový uzel za stávající aktuální uzel (obr. 11.9).



Obr. 11.9a Vkládání uzlu dvousměrně vázaného seznamu

2. vložení za aktuální uzel



Obr. 11.9b Vkládání uzlu dvousměrně vázaného seznamu

```
procedure VLOZ (var X: DVS; H: TYPHODNOTY);
{ vložení nového uzlu }
var
  Y: UKUZELDVS;          { pomocná proměnná }

begin { VLOZ }
  new(Y);   { nový uzel }
  Y^.HODNOTA := H;       { vložení hodnoty }
  with X do
    begin
      with Y^ do
        begin
          DOPREDU := AKTUALNI^.DOPREDU;
            { vazba vpřed }
          DOZADU := AKTUALNI; { vazba vzad }
          AKTUALNI^.DOPREDU^.DOZADU := Y;
          AKTUALNI^.DOPREDU := Y;
        end; { with Y }
      AKTUALNI := Y;  { vložený se stává aktuálním }
    end; { with X }
end; { VLOZ }
```

Po vytvoření nového uzlu, na který ukazuje proměnná Y, se upraví ukazatele v seznamu. Nejdříve se naplní vazby nového uzlu

```
DOPREDU := AKTUALNI^.DOPREDU; { vazba vpřed }
DOZADU := AKTUALNI; { vazba vzad }
```

Vazba DOZADU následujícího uzlu bude ukazovat na nový uzel

```
AKTUALNI^.DOPREDU^.DOZADU := Y;
```

Podobně bude na nový uzel ukazovat i vazba DOPREDU předchozího, tj. aktuálního uzlu

```
AKTUALNI^.DOPREDU := Y;
```

Rušení aktuálního uzlu provádí procedura ZRUS.

```
procedure ZRUS (var X: DVS);
{ zrušení aktuálního uzlu }
var
  Y: UKUZELDVS;   { pomocná proměnná }
```

```
begin { ZRUS }
  with X do
    begin
      AKTUALNI^.DOPREDU^.DOZADU:=
        AKTUALNI^.DOZADU;
      AKTUALNI^.DOZADU^.DOPREDU:=
        AKTUALNI^.DOPREDU;
      Y:= AKTUALNI;
      AKTUALNI:= Y^.DOPREDU;
      if AKTUALNI = HLAVA then { byl poslední }
        AKTUALNI:= Y^.DOZADU;
      if Y <> HLAVA then
        dispose(Y); { odstranění zrušeného uzlu }
    end; { with }
end; { ZRUS }
```

Procedura ZRUS (obr. 11.10) musí nejdříve vyřadit aktuální uzel ze seznamu. Do vazby DOZADU následujícího uzlu se dosadí ukazatel na uzel předchozí a do vazby DOPREDU předchozího uzlu ukazatel na uzel následující.

```
AKTUALNI^.DOPREDU^.DOZADU:=
                                    AKTUALNI^.DOZADU;
AKTUALNI^.DOZADU^.DOPREDU:=
                                    AKTUALNI^.DOPREDU;
```

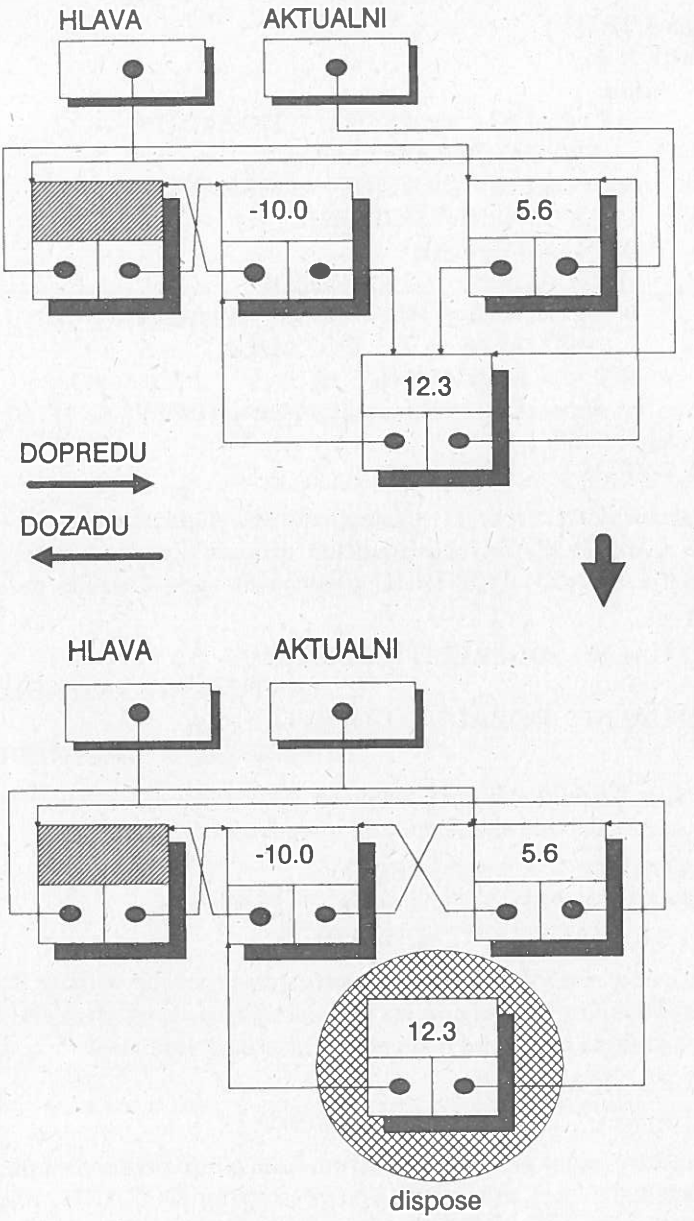
Po zrušení uzlu se stává aktuálním uzel následující. Byl-li zrušený uzel poslední v seznamu, bude aktuálním uzlem předcházející uzel.

```
AKTUALNI:= Y^.DOPREDU;
  if AKTUALNI = HLAVA then { byl poslední }
    AKTUALNI:= Y^.DOZADU;
```

Nakonec je zrušený uzel odstraněn příkazem procedury *dispose*. Pokud byl zrušen poslední uzel v seznamu, stane se aktuálním uzlem HLAVA. Ta však nesmí být zrušena nikdy, proto je operace *dispose* podmíněna.

```
if Y <> HLAVA then
  dispose(Y); { odstranění zrušeného uzlu }
```

Procedury změny aktuálního uzlu provedou pouze přechod na předchozí, resp. následující uzel uzlu aktuálního přes vazbu DOZADU, resp. DOPREDU. Procedura ZACATEK využívá složky HLAVA datového typu DVS pro nastavení prvního uzlu.



Obr. 11.10 Rušení uzlu dvousměrně vázaného seznamu

```
procedure ZACATEK (var X: DVS);
{ přesun na začátek }
begin { začátek }
  with X do
    AKTUALNI := HLAVA^.DOPREDU;
end; { ZACATEK }
procedure VPRED (var X: DVS);
{ posun směrem ke konci seznamu }
begin { VPRED }
  with X do
    if AKTUALNI^.DOPREDU <> HLAVA then{lze posouvat}
      AKTUALNI := AKTUALNI^.DOPREDU;
end; { VPRED }
procedure VZAD (var X: DVS);
{ posun směrem k začátku seznamu }
begin { VZAD }
  with X do
    if AKTUALNI^.DOZADU <> HLAVA then{je kam posouvat}
      AKTUALNI := AKTUALNI^.DOZADU;
end; { VZAD }
```

Funkce CTI přenese do výstupního parametru obsah hodnotové části aktuálního uzlu. Neexistuje-li aktuální uzel (AKTUALNI = HLAVA), je výsledkem funkce *false*, jinak *true*.

```
function CTI (var X: DVS; var H : TYPHODNOTY):boolean;
{ čtení hodnoty aktuálního uzlu }
begin { CTI }
  with X do
    begin
      H := AKTUALNI^.HODNOTA;
      CTI := AKTUALNI <> HLAVA;
    end; { with X }
end; { CTI }
```

Procedura PIS provede opačný přenos:

```
procedure PIS (var X: DVS; H : TYPHODNOTY);
{ zápis hodnoty do aktuálního uzlu }
begin { PIS }
  X.AKTUALNI^.HODNOTA := H;
end; { PIS }
```

Realizujme nyní program EDITOR využívající abstraktní datový typ DVS. Program představuje zjednodušený řádkový editor. Řádkový znamená, že při editaci textu je vždy jeden řádek aktuální a pouze s tímto jedním řádkem je možné provádět opravy.

Editor bude umět číst a zapisovat text z vnějšího souboru do pracovní oblasti editoru a zpět, vkládat, rušit a opravovat aktuální řádky textu a posouvat aktuální řádek v textu oběma směry. Další funkce textového editoru (vyhledávání, systematická náhrada textu ap.) může čtenář realizovat do navržené struktury programu EDITOR v rámci cvičení.

Struktura programu může být:

```
program EDITOR (input, output, SOUBOR);
var
  CYKLUJ : boolean; { příznak činnosti programu }
  ZNAK   : char;    { znak určující prováděný příkaz }
  SEZNAM : DVS;     { dvousměrně vázaný seznam }
  SOUBOR : text;    { textový soubor }
begin { EDITOR }
  INICIALIZACE;
  repeat
    read(ZNAK);   { čtení znaku určujícího příkaz }
    case ZNAK of { podle typu příkazu }
      ' '     : NOVYRADEK; { vložení nového řádku }
      '+'     : POSUN (1); { posun o jeden řádek vpřed }
      '-'     : POSUN (-1); { posun o jeden řádek vzad }
      'C','c' : OPRAVARADKU; { oprava řádku }
      'D','d' : ZRUSENIRADKU; { zrušení řádku }
      'R','r' : VLOZENITEXTU; { vložení textu ze souboru }
      'W','w' : ZAPISTEXTU;  { zápis textu do souboru }
      '1'     : NAPRVNI;    { na začátek textu }
      'E','e' : CYKLUJ:=false; { konec programu }
    end; { case }
    if CYKLUJ then      { dokud není konec }
      VYPISRADKU;  { výpis aktuálního řádku }
  until not CYKLUJ;
  writeln ('Konec programu EDITOR');
end. { EDITOR }
```

Program má známou formu konverzačního programu. Hlavní cyklus repeat se provádí tak dlouho, dokud logická proměnná CYKLUJ má hodnotu *true*.

Příkazy jsou zadávány jedním znakem (příkazy je možné zadávat jako malá i velká písmena), prozatím v počtu 9:

< mezera > text	přečte text a vloží jej za aktuální řádek,
D	zruší aktuální řádek,
C text	opraví aktuální řádek (nahradí původní text novým),
W	přečte text z vnějšího textového souboru,
R	zapíše text do vnějšího textového souboru,
+ resp. –	provede posun aktuálního řádku vpřed, resp. vzad,
1	provede přesun aktuálního řádku na začátek seznamu,
E	ukončí činnost programu.

Pro reprezentaci textu v programu užijeme abstraktní datový typ dvousměrně vázaný seznam (DVS). Typem hodnoty v uzlu TYPHODNOTY bude řádek textu reprezentovaný zhuštěným polem znaků

```
const
  MAXDELKARADKU = 80;   { délka řádku }
type
  TYPHODNOTY = packed array [1..MAXDELKARADKU] of char;
                        { řádek }
```

Nyní můžeme realizovat jednotlivé výkonné procedury programu. Úvodní procedura INICIALIZACE nastaví počáteční hodnoty proměnným a vypíše úvodní text na obrazovku:

```
procedure INICIALIZACE;
{ nastavení počátečních hodnot proměnných }
begin { INICIALIZACE }
  CYKLUJ := true;     { příznak činnosti programu }
  PRIPRAVA (SEZNAM); { příprava seznamu }
  writeln ('TEXTOVY EDITOR'); { úvodní výpis }
  writeln ('pro cteni ze souboru pouzij prikaz R');
  writeln;
end; { INICIALIZACE }
```

Procedura NOVYRADEK přečte ze vstupu řádek textu a vloží jej za právě aktuální řádek. Vložený řádek se stane aktuálním (je to vlastnost procedury VLOZ). Znaky se čtou z klávesnice tak dlouho, dokud není zapsán konec řádku; zbytek proměnné RADEK (je-li *eoln* pravdivé) se doplní mezerami. Nakonec se operací VLOZ uloží nový řádek do seznamu. Příkaz *readln* na konci procedury přečte další příkaz uživatele. Podobně je tomu na konci všech dalších popisovaných procedur.

```
procedure NOVYRADEK;
{ vložení nového řádku }
var
  RADEK : TYPHODNOTY; { pomocný řádek }
  I  : TYPDELKY;      { řídicí proměnná }
begin { NOVYRADEK }
  for I:=1 to MAXDELKARADKU do { zaplnění jednoho řádku }
    if eoln then
      RADEK [I] := ' ' { za koncem čteného textu mezery }
    else
      read(RADEK [I]); { znak čteného textu }
  readln;
  VLOZ(SEZNAM,RADEK); { vložení do seznamu }
end; { NOVYRADEK }
```

Procedura POSUN provede posun aktuálního řádku o zadaný počet řádků KOLIK směrem ke konci seznamu (KOLIK je kladné), popř. směrem k začátku (KOLIK je záporné). Pokud je zadaný počet příliš velký, posun vždy končí na posledním, resp. na prvním řádku (vlastnost procedur VPRED a ZPET).

```
procedure POSUN (KOLIK: integer);
{ posun aktuálního řádku o KOLIK řádků }
var
  I : integer; { řídicí proměnná }
begin { POSUN }
  for I:=1 to abs(KOLIK) do { počet posunů }
    if KOLIK>0 then     { posun vpřed }
      VPRED(SEZNAM)
    else                { posun vzad }
      VZAD(SEZNAM);
  readln;
end; { POSUN }
```

Procedura OPRAVARADKU provede čtení nového textu z klávesnice (stejně jako procedura NOVYRADEK) a příkazem procedury PIS se nahradí původní text novým.

```
procedure OPRAVARADKU;
{ oprava aktuálního řádku }
var
  RADEK : TYPHODNOTY; { pomocný řádek }
  I  : TYPDELKY;      { řídicí proměnná }
```

```
begin { OPRAVARADKU }
  for I: = 1 to MAXDELKARADKU do { zaplnění jednoho řádku }
    if eoln then
      RADEK [I] : = ' ' { za koncem čteného textu mezery }
    else
      read(RADEK [I]); { znak čteného textu }
  readln;
  PIS(SEZNAM,RADEK);     { vložení do seznamu }
end; { OPRAVARADKU }
```

Procedura ZRUSENIRADKU zruší aktuální řádek příkazem procedury ZRUS:

```
procedure ZRUSENIRADKU;
{ zrušení aktuálního řádku }
begin { ZRUSENIRADKU }
  ZRUS (SEZNAM);
  readln;
end; { ZRUSENIRADKU }
```

Následující dvě procedury ukládají/čtou text do/z vnějšího souboru. Procedura VLOZENITEXTU nejprve konverzací s uživatelem zjistí jméno souboru a potom jej příkazem procedury *assign* a *reset* připraví ke čtení. Tento postup je typický pro Turbo Pascal a v případě užití jiného překladače je nutné tuto část upravit. Charakteristický pro Turbo Pascal je i standardní datový typ *string*. Jde vlastně o řetězec znaků, s tím, že jej např. lze jako celek číst příkazem *read*. Do proměnné datového typu *string* lze také přiřazovat řetězce znaků různých délek (ne jediné délky, jak tomu bylo u datových typů **packed array** [1..N] **of** *char*). Zavádíme jej, protože druhý parametr standardní procedury Turbo Pascalu *assign* je tohoto typu.

Následuje čtení celého textového souboru cyklem while, přičemž vytváření nových řádků textu je obdobné jako v proceduře NOVYTEXT.

```
procedure VLOZENITEXTU;
{ vložení textu přečteného ze souboru }
var
  JMVSTUPU : string; { jméno vstupního souboru }
  RADEK    : TYPHODNOTY; { čtený řádek }
  I        : TYPDELKY; { řídicí proměnná }
begin { VLOZENITEXTU }
  writeln ('Zadej jmeno souboru :');
  readln;
```

```
    read (JMVSTUPU); { čtení jména souboru }
    assign (SOUBOR,JMVSTUPU);
    reset (SOUBOR); { příprava pro čtení }
    while not eof (SOUBOR) do
      begin
        for I: = 1 to MAXDELKARADKU do {zaplnění jednoho řádku}
          if eoln(SOUBOR) then
            RADEK [I] : = ' ' { za koncem čteného textu mezery }
          else
            read(SOUBOR,RADEK [I]); { znak čteného textu }
        VLOZ (SEZNAM,RADEK);
        readln(SOUBOR);
      end; { while }
    readln;
  end; { VLOZENITEXTU }
```

Procedura ZAPISTEXTU rovněž konverzací s uživatelem zjistí jméno souboru pro uložení textu a připraví jej pro zápis (příkazem procedury *assign* a *rewrite*). Aktuální řádek se nastaví na začátek (příkazem procedury ZACATEK). Není-li seznam prázdný (funkce CTI má hodnotu *true*), je cyklem repeat proveden postupný výpis všech řádků do souboru. Doba cyklování je dána hodnotou *false* logické proměnné POSLEDNI.

```
POSLEDNI : = JEKONEC (SEZNAM);
```

Hodnota této proměnné je nastavována funkcí JEKONEC, která nabude hodnoty *true* u posledního řádku textu. Posun na další řádek je prováděn příkazem procedury VPRED (nejde-li o poslední řádek). Nakonec je výstupní soubor uzavřen příkazem procedury *close* (je nutná jen v Turbo Pascalu).

```
procedure ZAPISTEXTU;
{ zápis editovaného textu do souboru }
var
  RADEK       : TYPHODNOTY;
  JMVYSTUPU   : string; { jméno souboru }
  POSLEDNI    : boolean; { příznak posledního řádku }
begin { ZAPISTEXTU }
  writeln ('Zadej jmeno souboru :');
  readln;
  read (JMVYSTUPU); { čtení jména }
  assign (SOUBOR,JMVYSTUPU);
  rewrite (SOUBOR); { příprava pro zápis }
```

```
    ZACATEK (SEZNAM);
    if CTI(SEZNAM,RADEK) then
      repeat { přes všechny řádky textu }
        if CTI (SEZNAM,RADEK) then
          writeln (SOUBOR,RADEK);
        POSLEDNI := JEKONEC (SEZNAM);
        if not POSLEDNI then
          VPRED (SEZNAM);
      until POSLEDNI;
    close (SOUBOR);
    readln;
  end; { ZAPISTEXTU }
```

Procedura NAPRVNI provede přesun aktuálního řádku na začátek seznamu (příkazem procedury ZACATEK).

```
  procedure NAPRVNI;
  { posun aktuálního řádku na začátek textu }
  begin { NAPRVNI }
    ZACATEK (SEZNAM);
    readln;
  end; { NAPRVNI }
```

Zbývá procedura pro výpis aktuálního řádku VYPISRADKU. Výpis se provede po provedení každého příkazu editoru, ale pouze v případě, že aktuální řádek existuje. V opačném případě (což je situace, kdy je seznam prázdný) se vypíše varovné hlášení.

```
  procedure VYPISRADKU;
  { výpis aktuálního řádku na obrazovku }
  var
    RADEK : TYPHODNOTY;
  begin { VYPISRADKU }
    if CTI(SEZNAM,RADEK) then
      writeln(RADEK)      { výpis na obrazovku }
    else
      writeln('Aktualni radek neexistuje');
  end; { VYPISRADKU }
```

Tím jsou realizovány všechny procedury a program je možné spustit. Jeho konečný tvar je označen 11.2.

```
program EDITOR (input,output,SOUBOR);
{ Program provádí editaci textu. Jsou realizovány tyto příkazy editoru :

<mezera><text>  vložení nového řádku za aktuální řádek; nový řádek
                  má obsah <text>,
+               posun aktuálního řádku o jeden vpřed,
-               posun aktuálního řádku o jeden vzad,
1               posun aktuálního řádku na začátek,
C<text>         přepsání obsahu aktuálního řádku; opravený řádek má
                  obsah <text>,
D               zrušení aktuálního řádku,
R               čtení textu ze souboru,
W               zápis textu do souboru,
E               konec edice.

Po provedení příkazu se vždy vypíše obsah aktuálního řádku
}
const
  MAXDELKARADKU = 80;  { délka řádku }
  CR = 13; { ordinální číslo znaku ukončujícího řádek v Turbo Pascalu }

type
  TYPDELKY    = 0 .. MAXDELKARADKU; { typ délky }
  TYPHODNOTY = packed array [1..MAXDELKARADKU] of char;
                    { řádek }
  UKUZELDVS = ^UZELDVS; { ukazatel na uzel seznamu }
  UZELDVS  =
    record  { uzel dvousměrného seznamu }
      HODNOTA : TYPHODNOTY;{ hodnotová část }
      DOPREDU,
      DOZADU : UKUZELDVS; { vazba vpřed a vzad }
    end; { UZELDVS }
  DVS      =
    record  { dvousměrně vázaný seznam – typ }
      HLAVA  ,          { hlava seznamu }
      AKTUALNI: UKUZELDVS; { aktuální uzel }
    end; { DVS }

var
  CYKLUJ    : boolean; { příznak činnosti programu }
  ZNAK      : char;    { znak určující prováděný příkaz }
  SEZNAM    : DVS;     { dvousměrně vázaný seznam }
  SOUBOR    : text;    { textový soubor }
```

```
procedure PRIPRAVA (var X: DVS);
{ nastavení počáteční hodnoty seznamu }
begin { PRIPRAVA }
  with X do
    begin
      new(HLAVA);     { vytvoření pomocného uzlu }
      with HLAVA^ do
        begin
          DOPREDU:=HLAVA;
          DOZADU:=HLAVA;
        end; { with HLAVA }
      AKTUALNI := HLAVA; { aktuální uzel neexistuje }
    end; { with }
end; { PRIPRAVA }

function JEKONEC (var X: DVS): boolean;
{ test existence následujícího uzlu }
begin { JEKONEC }
  with X do
    JEKONEC := AKTUALNI^.DOPREDU = HLAVA;
end; { JEKONEC }

procedure VLOZ (var X: DVS; H: TYPHODNOTY);
{ vložení nového uzlu }
var
  Y: UKUZELDVS;   { pomocná proměnná }
begin { VLOZ }
  new(Y);   { nový uzel }
  Y^.HODNOTA := H;   { vložení hodnoty }
  with X do
    begin
      with Y^ do
        begin
          DOPREDU:=AKTUALNI^.DOPREDU;{ vazba vpřed }
          DOZADU := AKTUALNI; { vazba vzad }
          AKTUALNI^.DOPREDU^.DOZADU := Y;
          AKTUALNI^.DOPREDU := Y;
        end; { with Y }
      AKTUALNI := Y;   { vložený se stává aktuálním }
    end; { with X }
end; { VLOZ }
```

```
procedure ZRUS (var X: DVS);
{ zrušení aktuálního uzlu }
var
  Y: UKUZELDVS;   { pomocná proměnná }
begin { ZRUS }
  with X do
    begin
      AKTUALNI^.DOPREDU^.DOZADU:=
        AKTUALNI^.DOZADU;
      AKTUALNI^.DOZADU^.DOPREDU:=
        AKTUALNI^.DOPREDU;
      Y:= AKTUALNI;
      AKTUALNI:= Y^.DOPREDU;
      if AKTUALNI = HLAVA then { byl poslední }
        AKTUALNI:= Y^.DOZADU;
      if Y <> HLAVA then
        dispose(Y); { odstranění zrušeného uzlu }
    end; { with }
end; { ZRUS }

procedure ZACATEK (var X: DVS);
{ přesun na začátek }
begin { začátek }
  with X do
    AKTUALNI:= HLAVA^.DOPREDU;
end; { ZACATEK }

procedure VPRED (var X: DVS);
{ posun směrem ke konci seznamu }
begin { VPRED }
  with X do
    if AKTUALNI^.DOPREDU <> HLAVA then
      { je kam posouvat }
      AKTUALNI:= AKTUALNI^.DOPREDU;
end; { VPRED }

procedure VZAD (var X: DVS);
{ posun směrem k začátku seznamu }
begin { VZAD }
  with X do
    if AKTUALNI^.DOZADU <> HLAVA then { je kam posouvat }
      AKTUALNI:= AKTUALNI^.DOZADU;
end; { VZAD }
```

```
function CTI (var X: DVS; var H : TYPHODNOTY):boolean;
{ čtení hodnoty aktuálního uzlu }
begin { CTI }
  with X do
    begin
      H := AKTUALNI^.HODNOTA;
      CTI := AKTUALNI <> HLAVA;
    end; { with X }
end; { CTI }

procedure PIS (var X: DVS; H : TYPHODNOTY);
{ zápis hodnoty do aktuálního uzlu }
begin { PIS }
  X.AKTUALNI^.HODNOTA := H;
end; { PIS }

{ ostatní procedury }

procedure INICIALIZACE;
{ nastavení počátečních hodnot proměnných }
begin { INICIALIZACE }
  CYKLUJ := true;     { příznak činnosti programu }
  PRIPRAVA (SEZNAM); { příprava seznamu }
  writeln ('TEXTOVY EDITOR'); { úvodní výpis }
  writeln ('pro cteni ze souboru pouzij prikaz R');
  writeln;
end; { INICIALIZACE }

procedure NOVYRADEK;
{ vložení nového řádku }
var
  RADEK : TYPHODNOTY; { pomocný řádek }
  I  : TYPDELKY;      { řídicí proměnná }
begin { NOVYRADEK }
  for I:=1 to MAXDELKARADKU do { zaplnění jednoho řádku }
    if eoln then
      RADEK [I] := ' ' { za koncem čteného textu mezery }
    else
      read(RADEK [I]); { znak čteného textu }
  readln;
  VLOZ(SEZNAM,RADEK);      { vložení do seznamu }
end; { NOVYRADEK }
```

```
procedure OPRAVARADKU;
{ oprava aktuálního řádku }
var
   RADEK : TYPHODNOTY; { pomocný řádek }
   I  : TYPDELKY;     { řídicí proměnná }
begin { OPRAVARADKU }
   for I: = 1 to MAXDELKARADKU do { zaplnění jednoho řádku }
      if eoln then
         RADEK [I] : = ' ' { za koncem čteného textu mezery }
      else
         read(RADEK [I]); { znak čteného textu }
   readln;
   PIS(SEZNAM,RADEK);     { vložení do seznamu }
end; { OPRAVARADKU }

procedure ZRUSENIRADKU;
{ zrušení aktuálního řádku }
begin { ZRUSENIRADKU }
   ZRUS (SEZNAM);
   readln;
end; { ZRUSENIRADKU }

procedure POSUN (KOLIK: integer);
{ posun aktuálního řádku o KOLIK řádků }
var
   I : integer; { řídicí proměnná }
begin { POSUN }
   for I: = 1 to abs(KOLIK) do { počet posunů }
      if KOLIK > 0 then     { posun vpřed }
         VPRED(SEZNAM)
      else                  { posun vzad }
         VZAD(SEZNAM);
   readln;
end; { POSUN }

procedure NAPRVNI;
{ posun aktuálního řádku na začátek textu }
begin { NAPRVNI }
   ZACATEK (SEZNAM);
   readln;
end; { NAPRVNI }
```

```
procedure VLOZENITEXTU;
{ vložení textu přečteného ze souboru }
var
   JMVSTUPU : string; { jméno vstupního souboru }
   RADEK  : TYPHODNOTY; { čtený řádek }
   I      : TYPDELKY; { řídicí proměnná }
begin { VLOZENITEXTU }
   writeln ('Zadej jmeno souboru :');
   readln;
   read (JMVSTUPU); { čtení jména souboru }
   assign (SOUBOR,JMVSTUPU);
   reset (SOUBOR); { příprava pro čtení }
   while not eof (SOUBOR) do
      begin
         for I:=1 to MAXDELKARADKU do { zaplnění jednoho řádku }
            if eoln(SOUBOR) then
               RADEK [I] := ' ' { za koncem čteného textu mezery }
            else
               read(SOUBOR,RADEK [I]); { znak čteného textu }
         VLOZ (SEZNAM,RADEK);
         readln(SOUBOR);
      end; { while }
   readln;
end; { VLOZENITEXTU }

procedure ZAPISTEXTU;
{ zápis editovaného textu do souboru }
var
   RADEK : TYPHODNOTY;
   JMVYSTUPU : string; { jméno souboru }
   POSLEDNI : boolean; { příznak posledního řádku }
begin { ZAPISTEXTU }
   writeln ('Zadej jmeno souboru :');
   readln;
   read (JMVYSTUPU); { čtení jména }
   assign (SOUBOR,JMVYSTUPU);
   rewrite (SOUBOR); { příprava pro zápis }
   ZACATEK (SEZNAM);
   if CTI(SEZNAM,RADEK) then
      repeat { přes všechny řádky textu }
         if CTI (SEZNAM,RADEK) then
            writeln (SOUBOR,RADEK);
         POSLEDNI := JEKONEC (SEZNAM);
```

```
        if not POSLEDNI then
          VPRED (SEZNAM);
      until POSLEDNI;
    close (SOUBOR);
    readln;
end; { ZAPISTEXTU }

procedure VYPISRADKU;
{ výpis aktuálního řádku na obrazovku }
var
  RADEK : TYPHODNOTY;
begin { VYPISRADKU }
  if CTI(SEZNAM,RADEK) then
    writeln(RADEK)      { výpis na obrazovku }
  else
    writeln('Aktualni radek neexistuje');
end; { VYPISRADKU }

begin { EDITOR }
  INICIALIZACE;
  repeat
    read(ZNAK);   { čtení znaku určujícího příkaz }
    if ord(ZNAK) = CR then { pouze RETURN }
      begin
        read(ZNAK); { ještě čtení LF }
        VPRED (SEZNAM);
      end
    else
      case ZNAK of { podle typu příkazu }
        ' '      : NOVYRADEK; { vložení nového řádku }
        '+'      : POSUN (1); { posun o jeden řádek vpřed }
        '-'      : POSUN (-1); { posun o jeden řádek vzad }
        'C','c'  : OPRAVARADKU; { oprava řádku }
        'D','d'  : ZRUSENIRADKU; { zrušení řádku }
        'R','r'  : VLOZENITEXTU; { vložení textu ze souboru }
        'W','w'  : ZAPISTEXTU;  { zápis textu do souboru }
        '1'      : NAPRVNI;    { na začátek textu }
        'E','e'  : CYKLUJ:=false; { konec programu }
        else
          begin { nápověda }
            writeln ('Neznamy prikaz');
            writeln;
            writeln ('mezera text ... novy radek');
```

```
                writeln ('+ ... posun o radek ke konci');
                writeln ('- ... posun o radek k zacatku');
                writeln ('1 ... na prvni radek');
                writeln ('C text ... oprava radku');
                writeln ('D ... zruseni radku');
                writeln ('R ... cteni textu ze souboru');
                writeln ('W ... zapis textu do souboru');
                writeln ('E ... konec cinnosti programu');
                writeln;
                readln; { přechod na další řádek }
              end; { nápověda }
        end; { case }
        if CYKLUJ then { dokud není konec }
          VYPISRADKU;      { výpis aktuálního řádku }
      until not CYKLUJ;
      writeln ('Konec programu EDITOR');
    end. { EDITOR }
```

Program 11.2

V konečné verzi programu jsou provedeny ještě dvě úpravy. Před provedením příkazu case se testuje, zda uživatel nezadal prázdný příkaz (tj. pouze stiskl klávesu RETURN, resp. ENTER). To se v Turbo Pascalu projeví přečtením znaku CR s ordinálním číslem 13. V takovém případě se provádí posun na další řádek příkazem procedury VPRED.

Pro standardní Pascal by bylo možné test

if *ord*(ZNAK) = CR **then** { pouze RETURN }

nahradit testem konce řádku

if *eoln* **then** { pouze RETURN }

Dále je do příkazu case provádějícímu větvení podle zadaného příkazu doplněna varianta else. Tedy varianta, která se provede v případě, že uživatel nezadá ani jeden z povolených znaků. V případě, že uživatel zadal znak, jenž neurčuje žádný ze známých příkazů, je vypsána nápověda, tj. kódy a popis všech možných příkazů editoru. Uživatel má možnost zjistit, které příkazy editoru má k dispozici, a zopakovat zadání původně nesprávného příkazu.

11.5 Cvičení

1. Realizujte jednosměrně vázaný seznam jako abstraktní datový typ. Volbu operací proveďte obdobně jako u abstraktního datového typu DVS.

2. Vytvořte program, který z textového souboru *input* přečte předem neomezený počet hodnot typu *integer* a vytvoří z nich jednosměrně vázaný seznam. Po vytvoření seznamu program vypíše hodnoty ze seznamu ve stejném pořadí, v němž byly čteny ze souboru *input*. (Využijte abstraktního datového typu z příkl. 1.)
3. Vytvořte program, který z textového souboru *input* přečte předem neomezený počet hodnot typu *real* a vytvoří z nich dvousměrně vázaný seznam. Po vytvoření seznamu program vypíše hodnoty ze seznamu v opačném pořadí, než byly čteny ze souboru *input*.
4. Příklad 3 doplňte tak, aby přečtené hodnoty byly před výpisem v seznamu setříděny vzestupně bublinovým tříděním. Třídění provádějte přímo v seznamu, nepoužívejte pomocné pole.
5. Abstraktní datový typ DVS doplňte o operace
   - JEZACATEK – test, zda aktuální uzel je v seznamu první,
   - KONEC – přesun na konec seznamu,
   - VLOZPRED – vložení uzlu před aktuální uzel,
   - RUSPRED – rušení uzlu před aktuálním uzlem.