

Situace, kdy introvertnost programu vyplývá již z povahy zvoleného algoritmu, však může nastat paradoxně také z právě opačného důvodu — totiž výběrem mimořádně efektivního algoritmu (např. velmi rychlého při řešení úlohy, kde je právě rychlosť rozhodující, nebo zaujmajícího minimum paměťového prostoru v situaci, kdy jsou rozhodující paměťové nároky). Podobně jako geniální lidé jsou i geniální algoritmy ve většině případů introvertní. V takovém případě musíme ovšem menší srozumitelnost algoritmu a programu chápát jako daň za vysokou efektivitu. (Několik jednodušších příkladů na efektivní a zároveň introvertní programy najde čtenář v následující kapitole — viz odstavce *Bílý trpaslík* a *Modrý přízrak*.)

Extroverti

Extroverti jsou ideálem a bylo napsáno velké množství pojednání o tom, jak vytvářet právě takové programy — tj. přehledné a dobře srozumitelné. Mohli bychom zde bez obtíží uvést řadu příkladů, neboť téměř každý krátký program je extrovert nebo je snadné z něj extroverta učinit. Čtenář by však pravděpodobně považoval za urážku své inteligence, kdybychom jako vzor ideálního programu uvedli:

100 REM PROGRAM NANIC

110 PRINT "1+1="; 2

120 END

(přestože se nepochybňuje jedná o velmi přehledný a snadno srozumitelný program).

Pokud se týká velkých a složitých programů, musíme po pravdě přiznat, že jsme zatím neměli to štěstí setkat se s takovým, jehož pochopení by bylo hračkou a nevyžadovalo jistou duševní námahu. Samozřejmě, že prostudování rozsáhlého a komplikovaného programu je náročné. Jde však o to, aby s rostoucí velikostí programu neklesala jeho srozumitelnost neúměrně rychle. Vhodným naprogramováním lze mnohé zachránit a ztráty na srozumitelnosti a přehlednosti udržet i u rozsáhlých programů ve snesitelných mezích. Tím, jak nám v tom může pomoci dobrá metodika programování, se budeme zabývat v kapitole *O programátorském stylu*.

6. MALÉ PANOPTIKUM PROGRAMŮ

Kterýkoli hotový program je zastaralý.

Kterýkoli nový program stojí více a trvá déle.

Kterýkoli program bude růst, až zaplní veškerou paměť, která je k dispozici.

Složitost programu roste, až překročí schopnosti programátora, který ho musí udržovat.

Je-li program užitečný, bude se muset předělat.

Je-li program neužitečný, vypracuje se k němu dokumentace.

Programátorský folklór

Kolega P. je mimořádně vyrovnaný člověk. V situacích, které většina lidí považuje za stresové, pronese — „vždyť o nic nejde“; hromadná doprava, nabídka výrobků spotřebního průmyslu ani rozmary jeho dívky ho nevyvádějí z míry. Pokud je nám známo, jeho duševní rovnováhou otřásl pouze program, který P. poté označil populárním astrofyzikálním termínem:

Černá díra

Pro čtenáře, který s tímto názvem dosud nepřišel do styku, stručné vysvětlení: černá díra je těleso s mimořádně silným gravitačním polem — tak silným, že sebevětší energie je nemůže překonat. Proto ji nic nemůže opustit — ani světlo či jiné záření — zato však černá díra nesmírně silně přitahuje a pochlaje vše ve svém okolí.

Podobně se choval i program kolegy P.: po spuštění začal pohlcovat nesmírné množství vstupních údajů. Kolega P. mu je trpělivě dodával z klávesnice mikropočítače téměř celou hodinu. Poté program pracoval další dvě hodiny. Kolega P. seděl u obrazovky a trpělivě čekal. Pak program skončil, aniž by za spoustu vstupních informací vydal jedinou výstupní.

Program, o kterém tu byla řeč, je ovšem extrém — v takovém případě je v něm zřejmě chyba, protože nemá smysl programovat algoritmy, které nevydávají žádné informace. Mnoho programů má však specifické vlastnosti, které nelze označit za vyloženě chybné, ale přesto jsou nežádoucí; jiné vlastnosti jsou naopak charakteristické pro dobré programy. Několik takových typů programů si nyní ukážeme.

Klepna

sice nemá nic společného s astronomií, ale přesto je v jistém smyslu opakem černé díry — vydává příliš mnoho informací, více, než je žádoucí. Stejně tak se chovají i programy patřící do této kategorie. Zahrnují nás opisy vstupních dat kombinovanými s hlášenými o provedených kontrolách, detailními výpisy mezivýsledků, zprávami o průběhu jednotlivých fází programu, pomocnými a ladícími výpisy apod. Samozřejmě, že tyto informace mohou být v určitých situacích užitečné, ba dokonce i nutné. Pokud jsou však automaticky vypisovány všechny, a navíc navzájem promíchané, mohou se změnit v informační prales, ve kterém jen s obtížemi nalezneme skutečně potřebné informace.

Existují i patologické případy, kdy všechny výstupní údaje tvoří balast, v němž se žádá užitečné informace neskrývají. Na jednom semináři o programování bylo referováno o výpočetním středisku, kde se každý měsíc zpracovává agenda, jež vychrlí menší valník výstupních sestav. Ty se odvážejí do skladistiště postaveného zvlášť pro tento účel. Jiný smysl agenda nemá.

Rudý obr

Tento termín jsme si opět vypůjčili z astronomie. Označují se tak relativně chladné hvězdy s nepatrnou hustotou a obrovskými rozměry. Program „rudý obr“ se vyznačuje tím, že zabírá v paměti mnohem více místa, než kolik je skutečně zapotřebí. Příčin vzniku rudých obrů je mnoho. Jednou z nich může být neefektivní zobrazení dat v počítači. Ujasněme si to na příkladu.

Náš mikropočítač nám může pomoci šetřit pohonné hmoty tím, že vybere nejkratší cestu ze zadaného města A do zadaného města B. Odhlédneme teď od faktu, že nejkratší cesta nemusí být vzhledem ke stavu komunikací tou nejvhodnější, a pro jednoduchost budeme předpokládat, že dvě spolu sousedící města jsou spojena jen jednou silnicí. Náš program tedy bude potřebovat vstupní data obsahující informace o tom, jaká je vzdálenost mezi jednotlivými sousedícími městy. Data můžeme elegantním způsobem uložit v tzv. maticeové formě; v programu deklarujeme dvouzměrné pole, řekněme pole P, a jednotlivá města očíslovujeme postupně od jedné. Do P(I, J) uložíme vzdálenost mezi městy I a J, pokud I sousedí s J, nebo nulu, pokud I a J nesousedí. Předpokládáme-li, že počet měst nepřekročí tisíc, musíme pole P deklarovat příkazem

DIM P(1000, 1000)

čímž vznikl krásný rudy obr — toto jediné pole obsahuje milion čísel (a na mnoha počítacích se proto nedá použít, nebo alespoň nikoli v Basiku). Drtivá většina položek v poli P bude samozřejmě obsahovat pouze nuly, takže velký prostor, který pole v paměti obsazuje, využíváme velmi neekonomicky. Pokud si tuto skutečnost uvědomíme, můžeme tytéž informace uložit zhuštěně: odhadneme-li, že v průměru má jedno město nanejvýš 5 bezprostředních sousedů, vejdu se informace do tří polí M, N, P, z nichž každé má 5000 položek.

Data v nich můžeme organizovat např. tak, že každému silničnímu úseku přidělíme číslo I, do M(I) a N(I) zapíšeme pořadová čísla měst, která I-tý úsek spojuje, a do P(I) délku tohoto úseku. Například M(1)=15, N(1)=60, P(1)=8,5 znamená, že první silniční úsek spojuje patnácté město se sedesátým a je dlouhý 8,5 km. (Podrobněji se s tímto problémem — jak nalézt nejkratší cestu mezi dvěma městy — seznámíme ve 14. kapitole.)

Mohli bychom dokonce ušetřit jedno z polí M, N, kdybychom dvojice čísel odpovídajících sousedním městům shrnuli do jednoho čísla — např. tak, že bychom první z čísel vynásobili tisícem, přičetli k druhému číslu a výsledek uložili do jediné položky v poli — M(I). Dekódování původní dvojice z takto uloženého údaje je velmi snadné:

100 LET M=INT(M(I)/1000)

110 LET N=M(I)-1000*M

V tomto případě je však sporné, zda úspora paměti vyváží zpomalení a určité zkomplikování algoritmu. Podobným způsobem můžeme rozsáhlá pole obsahující jen nuly a jedničky podstatně zkrátit tím, že kombinaci určitého počtu nul a jedniček považujeme za zápis jediného čísla ve dvojkové soustavě.

Rudého obra lze vytvořit nejenom nadměrným rozsahem pracovních polí, ale i nadměrným počtem příkazů. Máme například napsat program pro utajení textů pomocí šifry staré přes 2000 let — používal ji již Caesar. V této šifře se každé písmeno nahrazuje písmenem, které stojí v abecedě o tří pozice dále (abecedu považujeme za uzavřenou do kruhu, po Z následuje opět A). Primitivním naprogramováním stvoříme „rudého obříka“.

Ukážeme zde jenom část programu, šifrující jedno písmeno uložené v proměnné „strojové“ písmo:

100 IF Z\$ <> "A" THEN 130

110 PRINT "D";

120 GOTO 900

130 IF Z\$ <> "B" THEN 160

140 PRINT "E";

150 GOTO 900

...

840 IF Z\$ <> "Y" THEN 870

850 PRINT "B";

860 GOTO 900

870 PRINT "C";

900 REM POKRACOVANI — SIFROVANE PISMENO SE VYTISKLO

Víme-li, že písmena jsou ve vnitřním kódu počítače uložena od 65. pozice (písmeno A) do 90. pozice (písmeno Z), a použijeme-li standardní funkce CHR a CHR\$ (viz příloha 1), můžeme tentýž úsek naprogramovat mnohem stručněji a elegantněji. Navíc bude program pracovat podstatně rychleji:

100 LET N=CHR(Z\$)+3

110 IF N < 91 THEN 130

120 N=N-26

130 PRINT CHR\$(N);

Můžete namítnout, že původní program měl tu výhodu, že by se dal po malých úpravách použít i pro rafinovanější šifru, kdy se písmena nahrazují méně uspořádaně, např. A se mění na F, B na T, C na A atd. I tuto šifru však lze zpracovat, aniž by se z programu stal rudý obr — stačí na začátku programu definovat šifrovací tabulkou dlouhou 26 znaků a n -té písmeno abecedy nahradit n -tým písmenem z této tabulky:

10 REM SIFROVACI TABULKA

20 LET T\$="FTA..."

...

100 LET N=CHR(Z\$)-64

110 REM N JE PORADOVE CISLO PISMENA V ABECEDA

120 PRINT SUB\$(T\$,N,1);

(Funkce SUB\$ je opět popsána v příloze 1.)

Bílý trpaslík

je v jistém smyslu opakem rudého obra. V astronomii tak nazýváme horou, velmi malou, ale velmi hustou hvězdu. Zatímco rudý obr má průměrnou hustotu menší než vzduch a Slunce je v průměru o něco hustší než voda, má litr bílého trpaslíka hmotnost miliónů tun.

Podobně je program „bílý trpaslík“ velmi „hutný“, tj. extrémně nenáročný co se týká nároků na paměť počítače. Bohužel vytvořit ho nebývá vůbec snadné. Často je založen na jemných a komplikovaných tricích, vycházejících z matematických teorií.

Bílý trpaslíci jsou většinou dílem profesionálů, avšak překvapivě často se vyskytují také jako výsledek vtipních idejí programátorů-amatérů. Za zárodelek bílého trpaslíka by snad bylo možno považovat poslední šifrovací program z odstavce o rudých obrech.

Klasickým bílým trpaslíkem je algoritmus pro výpočet data, na něž v daném roce připadnou velikonoce. Podle dávné tradice je velikonoční neděli neděle následující po prvním jarním úplňku. Její určení tedy zdánlivě vyžaduje rozsáhlé astronomické výpočty — stanovení okamžiku, kdy začíná astronomické jaro (to je 20. nebo 21. března v různých hodinách) a okamžiku, kdy je Měsíc přesně v úplňku. Přesto geniální německý matematik Karl Friedrich Gauss objevil pro tento výpočet nepříliš složitý algoritmus, který se v Basiku dá zapsat takto:

10 INPUT R

20 LET D=MOD(19*MOD(R,19)+24,30)

30 LET D=22+D+MOD(5+2*MOD(R,4)+4*MOD(R,7)+6*D,7)

40 IF D<57 THEN 60

50 LET D=D-7

60 IF D>31 THEN 90

70 PRINT D;". BREZNA"

80 STOP

90 PRINT D-31;". DUBNA"

100 END

Můžete si ověřit porovnáním s kalendáři, že algoritmus funguje pro R od 1901 do 2000, tj. pro kterýkoli rok ve 20. století (projiná století je nutno změnit konstantu 24 v řádku 20 a konstantu 5 v řádku 30). Nepokoušejte se však pochopit, jak funguje — ztratili byste možná tolík času, že na dočtení této knížky by vám už žádný nezbýl. (Gauss bývá považován za člověka s nejvyšším IQ v dějinách lidstva a sledovat myšlenky géniů zpravidla není snadné.)

Často se s blízkými trpaslíky setkáváme při programování v assembleru nebo ve strojovém kódu. Aniž bychom se uchylovali ke dvojkové soustavě a ke strojovým instrukcím, naznačíme zde základní myšlenku jednoho z nich. Máme za úkol oddělit z čísla pouze celočíselnou část (např. ze 156,72 udělat 156). Tuto operaci provádí, alespoň pro kladná čísla, basikovská funkce INT. Též akci lze však provést jedinou (!) strojovou instrukcí (pokud ovšem má počítač ve strojovém kódu instrukci sčítání v pohyblivé řádové čárce). Pro pochopení použitého triku si však musíme říci něco o tom, jakým způsobem počítač sčítá dvě čísla. Reálná čísla (a v Basiku obvykle všechna čísla) jsou v počítači zobrazena v tzv. pohyblivé čárce. Např. číslo 9560 je zapsáno jako $0,956 \cdot 10^4$, je tedy rozloženo na mocninu deseti, násobenou číslem nepříliš menším než jedna (tzv. mantisou, ležící v rozmezí od 0,1 do 0,99999...). Sčítání takto zobrazených čísel (zůstaňme pro jednoduchost u čísel kladných) se pak děje ve třech krocích:

1. Menší číslo se upraví tak, aby obsahovalo stejnou mocninu desíti jako číslo větší. Např. při sčítání $9560 + 750$ (tj. $0,956 \cdot 10^4 + 0,75 \cdot 10^3$) změníme zápis druhého čísla na $0,075 \cdot 10^4$.

2. Takto upravená čísla nyní snadno sečteme. V našem případě bude $0,956 \cdot 10^4 + 0,075 \cdot 10^4 = 1,031 \cdot 10^4$.

3. V případě potřeby výsledek upravíme tak, aby mantisa zůstala v rozmezí od 0,1 do 0,99999... . V našem případě $1,031 \cdot 10^4$ změníme na $0,1031 \cdot 10^5$. (Hodnota čísla se tím nezměnila, pouze způsob jeho zápisu.) A to je výsledek našeho příkladu — v pohyblivé čárce zapsané číslo 10310.

Na základě znalosti tohoto postupu můžeme náš problém elegantně vyřešit. Musíme si jenom ještě uvědomit, že mantisa je v počítači zobrazena pouze s určitým počtem desetinných míst a vzniknou-li v průběhu výpočtu další místa, počítač je prostě ignoruje. Předpokládejme například, že náš počítač dokáže pracovat pouze s pětimístnou mantisou. Podívejme se, co se stane, když k nějakému číslu přičteme poněkud neobvykle zapsanou nulu, totiž číslo $0,0 \cdot 10^5$. Nezapomeňme na to, že šesté a další místo mantisy se ztrácí. A zkuseme to rovnou na několika číslech:

$$\begin{aligned} 12,345: 0,12345 \cdot 10^2 + 0,00000 \cdot 10^5 &= \\ &= 0,00012 \cdot 10^5 + 0,00000 \cdot 10^5 = \\ &= 0,00012 \cdot 10^5 = \\ &= 0,12000 \cdot 10^2, \text{ tj. } 12 \end{aligned}$$

$$\begin{aligned}
 8204,8 : 0,82048 \cdot 10^4 + 0,00000 \cdot 10^5 &= \\
 &= 0,08204 \cdot 10^5 + 0,00000 \cdot 10^5 = \\
 &= 0,08204 \cdot 10^5 = \\
 &= 0,82040 \cdot 10^4, \text{ tj. } 8204
 \end{aligned}$$

$$\begin{aligned}
 0,625 : 0,62500 \cdot 10^0 + 0,00000 \cdot 10^5 &= \\
 &= 0,00000 \cdot 10^5 + 0,00000 \cdot 10^5 = \\
 &= 0,00000 \cdot 10^5 = \\
 &= 0,00000 \cdot 10^0, \text{ tj. } 0
 \end{aligned}$$

Vidíme, že bez ohledu na velikost čísla se při vyrovnávání na mocninu 10^5 ztratí právě ty číslice, které stojí v původním čísle za desetinnou čárkou.

Závěrem k tomuto triku ještě několik poznámek: Ve skutečnosti je zobrazení čísel téměř ve všech počítacích založeno na dvojkové nebo šestnáctkové soustavě a nikoli na soustavě desítkové. Na principu to však nic nemění a nás postup se dá použít i v takovém případě. (Je zajímavé, že autoři překladačů se k němu zpravidla neuchylují, i když úsporněji než jedinou instrukcí tuto operaci jistě naprogramovat nelze.) Na některých počítacích bychom avšak neuспěli z jiného důvodu: Řada počítaců má logiku operace sčítání natolik chytřou, že pokud je jeden ze sčítanců nulový, sčítání se vůbec neprovádí. Je-li tato logika ještě chytřejší a rozezná-li i netypickou nulu s vysokým exponentem jako nulu, budeme zklamáni. (Autoři však na takový počítac dosud nenařazili.) Další úskalí je svázáno s použitým programovacím jazykem: v řadě jazyků nelze nulu s nestandardním exponentem vůbec zakódovat (překladač vyhodnotí i zápis $\emptyset E 6$ jako obyčejnou nulu). Proto jsme omezeni na jazyky, kde lze nestandardní nulu zadat ve vnitřním zobrazení — např. jako konstantu v šestnáctkové soustavě. Basic to většinou neumožňuje.

Říká se, že není růže bez trnů. Ačkoli jsou bílí trpaslíci programy vzorné a záslužné, jednu nepřijemnou vlastnost bohužel mívají — často jsou totiž typickým příkladem introverta z kapitoly *Program nebo džungle*. Zavrhnout je kvůli tomu nebudeme, to by byla škoda. Lepší je napravit to tím, že činnost, která není jasná ze zdrojového textu programu, vysvětlíme v komentářích. Jimi by se nemělo šetřit ani v jiných případech a příkaz **REM** by měl patřit k nejpoužívanějším příkazům dobrého programátora. Nepoužívat komentáře se rozhodně nevyplácí; je to krátkozraké a připomíná to nechvalně známou zásadu „šetřit se musí, ať to stojí, co to stojí.“ Za takto ušetřené minuty se pozdejší — když se do programu musí zasáhnout — platí hodinami strávenými vzpomínáním na to, jak je to vlastně uděláno.

Želva

*Želva je tu s námi tak dlouho proto,
že ona je lepší než cokoli jiného co
do své želvovitosti.*

E. Fredkin

Želva je tradičním symbolem pomalosti. Želvovitý program si dá na čas, než přestane trápit procesor a obtěžovat vnitřní i vnější paměť. Když konečně skončí, uslyšíte (budete-li dobře poslouchat), jak se někde od aritmetické jednotky ulehčeně ozve sotva slyšitelné „Uff! !“ Některé počítáče sice reagují se stoickým klidem, ale stejně si myslí o tvůrce takového programu své. Bylo již vytvořeno mnoho programů, které by — kdyby nebyly přerušeny vnějším zásahem — počítaly ještě několik příštích století.

Želvy vznikají hlavně tím, že zvolíme nevhodný algoritmus. Obratem ruky lze přeměnit na želvu poslední verzi našeho šifrovacího programu z odstavce o rudých obrech. Stačí jinak zorganizovat šifrovací tabulku: místo principu „na n -té místě v tabulce je to písmeno, kterým se má nahradit n -té písmeno z abecedy“ můžeme použít princip „na n -té místě v tabulce je písmeno, které se má nahradit n -tým písmenem abecedy“. Tabulka začínající znaky „EJ...“ bude tedy nyní znamenat: písmeno E je třeba nahradit písmenem A, místo J se šifruje B atd. I podle takto uspořádané tabulky se dá šifrovat: je-li třeba zašifrovat určité písmeno, prohlédneme tabulku a zjistíme, kde se v ní toto písmeno nachází. Do zašifrovaného textu potom zapíšeme tolikáte písmeno z abecedy, na kolikátém místě jsme původní písmeno v tabulce našli. Pro každé písmeno musíme tedy prohlížet šifrovací tabulku, což je pomalé — a želva se úspěšně plouží.

Ostatně, řada rudých obrů má želvovitý charakter. Vezměme např. problém hledání nejkratšího spojení mezi dvěma městy (uvedený rovněž v odstavci *Rudý obr*). Kdybychom zvolili jako algoritmus pro řešení tohoto problému prohledání všech možných cest spojujících daná města (samozřejmě takových, na kterých se žádné město nevyskytne dvakrát) a vybrání cesty s minimální vzdáleností, stvoříme přímo superželvu; zkuste sami odhadnout počet možných cest pro nějaký jednoduchý případ a čas potřebný pro vyřešení úlohy tímto způsobem (odhad můžete konfrontovat s kap. 14.).

Velmi často vzniká želva tím, že jsou dovnitř cyklu zařazeny výpočty, které stačí provést pouze jednou mimo cyklus. Například v následujícím programovém úseku se zbytečně padesátkrát počítá funkce logaritmus:

```

100 DIM A(50)
110 LET S=0
120 FOR I=1 TO 50
130   LET S=S+A(I)*LOG(X)
140 NEXT I

```

Drobnou úpravou získáme program, který dá tentýž výsledek a přitom spoří třeba podstatně méně strojového času:

```

100 DIM A(50)
105 LET P=LOG(X)
110 LET S=0
120 FOR I=1 TO 50
130 LET S=S+A(I)*P
140 NEXT I

```

Modrý přízrak

Původně se tak jmenovalo raketové auto, které vytvořilo na Solném jezeře několik světových rychlostních rekordů. Také program „modrý přízrak“ je, pokud se týká rychlosti, nedostižný. Triky, které se k tomu účelu používají, často ohromují svou důmyslností a bývají založeny na matematických teoriích. Uvedeme si pro ilustraci několik nejednodušších.

Osmou mocninu X podle příkazu:

```
100 LET Y=X^8
```

bude většina překladačů Basiku počítat oklikou přes logaritmy podle vztahu $x^8 = e^{8 \cdot \ln x}$. Výpočet exponenciální a logaritmické funkce je značně pomalý a při jeho častém opakování to pocítíme. Chytřejší překladač může zjistit, že exponent je celé číslo, a vyhne se těmto funkcím tak, že X sedmkrát znásobí sebou samým. Horší překladač donutíme k tomuto způsobu výpočtu tím, že příkaz napišeme ve tvaru:

```
100 LET Y=X*X*X*X*X*X*X
```

V uspokojení nad tím, jak jsme obelstili překladač, nás možná ani nenapadne, že výpočet mohl být proveden pouze se třemi operacemi násobení:

```
100 LET Y=X*X (tedy Y = X2)
```

```
110 LET Y=Y*Y (tedy už Y = X4)
```

```
120 LET Y=Y*Y (a nyní Y = X8)
```

Poznamenejme, že existují chytré překladače, které tímto způsobem přeloží i příkaz zapsaný v původním tvaru. Současně je třeba upozornit na to, že u interpretačních překladačů (viz slovníček) takovým postupem situaci spíše zhoršíme; podrobněji viz str. 85.

Pravděpodobně jedním z prvních algoritmů patřících do kategorie modrých přízraků je známé Hornerovo schéma pro výpočet hodnoty polynomů (viz slovníček). Příkaz

```
100 LET Y = A*X^4 + B*X^3 + C*X^2 + D*X + E
```

odpovídá — i když se vyhneme oklice přes logaritmy — deseti násobením a čtyřem sčítáním. Uzávorkováním podle Hornerova schématu dostaneme příkaz

```
100 LET Y = (((A*X + B)*X + C)*X + D)*X + E
```

který obsahuje pouze čtyři násobení a čtyři sčítání a dává stejný výsledek.

Uvedené příklady patří mezi nejednodušší. Krása rychlých algoritmů však vynikne právě u těch složitějších a rafinovanějších, na které už v naší knížce nezbývá místo a které také obvykle vyžadují jisté speciální matematické zna-

losti. Toho, kdo se jimi chce pokochat a nezalekne se potřebné matematiky, odkazujeme na zajímavou knihu [MIKL79]. Zde uvedeme pouze (bez programu) jeden příklad, k němuž postačí běžná znalost aritmetiky.

Na začátek malá rozvíčka: Časový údaj v minutách a sekundách máme zapsán jako čtyřmístné celé číslo (např. 15 minut a 41 sekund jsme zapsali jako 1541). Máme převést tento údaj na sekundy (pochopitelně to není 1541 sekund). Většinou se takový úkol řeší tak, že se oddělí zvlášť minuty (15) a zvlášť sekundy (41), a výsledek je potom šedesátinásobek počtu minut plus počet sekund, tj. $15 \cdot 60 + 41 = 941$.

Toto řešení lze poněkud zjednodušit. Číslo 1541 se dá zapsat jako $15 \cdot 100 + 41$. Abychom z něj dostali požadovanou hodnotu $15 \cdot 60 + 41$, stačí odečíst $15 \cdot 40$. Ze čtyřmístného čísla tedy oddělíme první dvě místa, vynásobíme je 40 a odečteme od původního čísla. (Pro praktický výpočet je výhodnější místo oddělení prvních dvou číslic znulovat poslední dvě a takto vzniklé číslo násobit 0,4 místo 40.) Postup výpočtu bude následující:

Původní číslo:	1541
----------------	------

Číslice vytiskněny kurzívou znulujeme:	1500
--	------

Násobíme 0,4:	0600
---------------	------

Odečteme od původního čísla:	0941
------------------------------	------

Dostali jsme skutečně 941 sekund.	
-----------------------------------	--

Na první pohled se zdá, že jsme tímto trikem ušetřili jen málo — totiž to, že poslední dvě místa již z čísla oddělovat nemusíme. To však byla, jak jsme již řekli, pouze rozvíčka. Přejděme nyní k poněkud složitějšímu případu. V některých zemích se sekunda dále dělí na šedesát tercií. Vezměme nyní časový údaj v hodinách, minutách, sekundách a terciích, zapsaný jako osmimístné číslo. Jak jej převedeme na tercie? Tentokrát můžeme výpočet provádět do jisté míry souběžně. Ukažme si to opět na konkrétním příkladě. Máme na tercie převést 51 hod 14 min 26 a 39 tercií:

51142839

51002800

20401120

30741719

Číslice vytiskněny kurzívou znulujeme:
--

Takto vzniklé číslo násobíme 0,4:

Odečteme od původního čísla:

Tím jsme převedli hodiny s minutami na minuty a souběžně sekundy s terciemi na tercie: máme 3074 minut a 1719 tercií. Samo číslo 30741719 znamená $3074 \cdot 10000 + 1719$. Protože minuta má 3600 tercií, potřebujeme z něj udělat číslo $3074 \cdot 3600 + 1719$. Musíme tedy obdobně oddělit počet minut, násobit jej 6400 a odečíst, resp. (což je totéž) v osmimístném čísle poslední 4 číslice nahradit nulami, výsledek násobit 0,64 a odečíst.

30741719

30740000

19673600

11068119

Pokračujme tedy ve výpočtu:

Číslice vytiskněny kurzívou znulujeme:
--

Násobíme 0,64:

Odečteme od 30741719:

Poslední číslo udává hledaný počet tercií.
--

Uvedený postup je možno dále rozširovat na vícemístné skupiny čísel zapsaných v šedesátkové soustavě (tj. takových, kde následující jednotka je šede-

sátkrát větší než předcházející). Každý další krok umožnuje zpracování dvojnásobně delšího čísla. Po nepatrné úpravě lze algoritmus použít i pro převod z libovolné jiné soustavy do soustavy desítkové.

Počítač, který pracuje ve dvojkové soustavě, může zcela obdobným algoritmem převádět čísla z libovolné (např. desítkové) soustavy do soustavy dvojkové. Každý krok přitom vyžaduje pouze 4 strojové instrukce (zapamatování, znulování některých číslic, násobení a přičtení zapamatovaného čísla); převod osmimístného desítkového čísla lze tedy provést 12 instrukcemi, na šestnáctimístné číslo je zapotřebí 16 instrukcí. To je skutečně rychlosť, která opravňuje označit tento algoritmus jako modrý přízrak.

Mnohdy je modrý přízrak současně bílým trpaslíkem, jindy mohou být požadavky rychlosti a paměťové úspornosti protikladné.

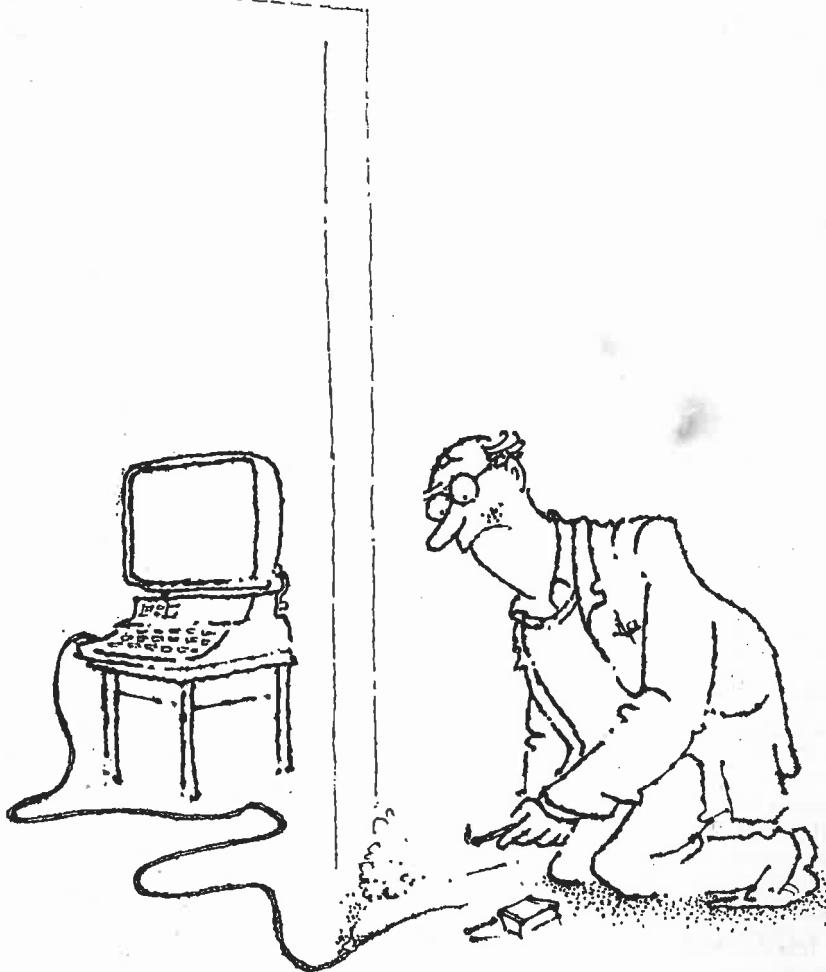
Šrapnel

*Jestliže program bezchybně funguje, je to jenom proto, aby mohl v nestřeženém okamžiku způsobit tím větší škodu.
Programátorský folklór*

Jak známo, tento druh dělové koule při neopatrné manipulaci vybuchne i tehdy, když se to od něj právě nežádá, a způsobí škodu. Něco analogického se může stát také při neopatrném zacházení s některým programem — i pokud jde o program, který při obezretném používání svou funkci plní. Podívejme se na dva příklady utajených programových šrapnelů.

Ve většině dialektů Basiku lze vytvořený program uložit do vnější paměti direktivou **SAVE název** (viz *Příloha 1*). Existují však dialekty, v nichž program nelze uložit pod názvem totožným s již existujícím programem. Co dělat v případě, že pod daným jménem máme již zapsanou starší verzi programu? Novou verzi můžeme uložit pod jiným názvem (to je však často nežádoucí) nebo nejdříve starou verzi z vnější paměti vymazat (k tomu máme k dispozici např. direktivu **UNSAVE**) a potom nám již počítač dovolí direktivou **SAVE** zapamatovat novou verzi. Zdánlivě zcela bez problému — dokud se nestane například to, že ihned po zrušení staré verze praskne pojistka. V tom případě šrapnel „vybuchne“ — nová verze z vnitřní paměti zmizela výpadkem proudí, ve vnější paměti není dosud zapsána, starou verzi ve vnější paměti jsme právě vymazali a zbyly nám jen oči pro pláč.

Jiný pěkný šrapnel vytrestal řadu našich kolegů — a posléze i jednoho z autorů. Zadá-li programátor požadavek na vytisknutí určitého souboru (programu, výsledků apod.), neprovedou obvykle moderní počítače tento příkaz ihned. Příslušný požadavek na tisk si operační systém zapamatuje a vytiskne všechno najednou teprve po ukončení práce. Často se stane, že soubor, jehož



vytištění jsme původně požadovali, nakonec tisknout nepotřebujeme. V zájmu šetření nedostatkovým papírem do tiskárny byl proveden drobný zásah do operačního systému našeho počítače, takže na konci práce počítač vypíše na terminál seznam všech požadavků na tisk a položí dva dotazy: „Chceš některé z uvedených souborů zrušit?“ a „Chceš některé z nich nevytisknout?“ Odpovíme-li na první dotaz kladně, příslušný soubor se zlikviduje (tato možnost se použije například tehdy, zjistíme-li, že výsledky výpočtu jsou špatné, protože v programu byla chyba). Kladná odpověď na druhý dotaz znamená, že se soubor ponechá beze změny, ale tisknout se nebude.

Všechno pracuje k naší plné spokojenosti, dokud si nespletejeme oba dotazy. Snad každý z uživatelů našeho počítače si již vymazal nějaký soubor prostě proto, že jej nejprve chtěl vytisknout, pak si to rozmyslel a místo zrušení požadavku na tisk nechal zrušit i soubor. Velmi nepříjemnou hodinu prožil jeden z autorů poté, co tímto způsobem vymazal soubor, který byl součástí operačního systému. Musel pak zoufale hledat jeho kopii a modlit se, aby do té doby, než se mu podaří soubor obnovit, nebyla přerušena dodávka proudu nebo nehavaroval operační systém počítače. Přesně v okamžiku, kdy odesílal z terminálu inkriminovaný příkaz, mu totiž došlo, co provedl, a ihned mu bylo jasné, že bez tohoto souboru není možné spustit znova operační systém. Naštěstí elektrárna vydržela a operační systém také.

Jsou známy i případy, kdy šrapnel byl do programu vložen úmyslně. (Propuštěný programátor tak zajistil, aby se půl roku po jeho odchodu smazaly životně důležité soubory. V takovém případě je již případnější mluvit o „časované bombě“.) Došlo už dokonce k vydírání typu: „Ve vaší databance je časovaná bomba — za sto tisíc dolarů v použitých bankovkách vám prozradíme, jak ji likvidovat, jinak zítra v poledne vaše záznamy zmizí.“ Tím jsme se však dostali do oblasti, o které si víc povíme v intermezzu *O počítačích a zločinu*.

Hroch

Představa tohoto mohutného zvířete a jeho kůže tlusté jako podrážka bytelné obuví je v naší mysli spojena s línou netečností, ba přímo otrlostí k vnějším podnětům. Program-hroch prostě není žádná citlivka. Nedá se vyvést z míry neočekávanými hodnotami vstupních dat a je zabezpečen proti všem možným výjimečným a neočekávaným situacím, při kterých by mohl natropit škodu. Většina zkazek typu „počítače jsou k ničemu — dostal jsem už třetí upomínu na zaplacení dlužné částky 0,— Kčs a počítač mi vyhrožuje soudním vymáháním“ by nespříhala světlo světa, kdyby příslušný program byl hrochem. (Je paradoxní, že nejlépe bývají zabezpečeny proti nesmyslným vstupním údajům či chybám obsluze programy, u nichž vcelku o nic nejde — totiž počítačové hry.)

Kanón na vrabce

se používá doslu výjimečně. Zato se poměrně často aplikují složité algoritmy i tam, kde by se vystačilo s mnohem jednoduššími.

Napsat program, který vybere maximální prvek z pole o (řekněme) tisíci prvcích a vytiskne jej, je velmi snadné. Lze to udělat například takto:

10 DIM A(1000)

40 LET M=A(1)

50 FOR I=2 TO 1000

60 IF A(I) <=M THEN 80

70 LET M=A(I)

80 NEXT I

90 PRINT "NEJVETSI PRVEK:" ; M

...

Mohli bychom si však vzpomenout, že v knihovně podprogramů máme k dispozici třídicí podprogram, který dokáže seřadit prvky souboru od největšího k nejmenšímu. Stačí jej tedy vyvolat a ze setříděného souboru vytisknout první prvek. Tím sice ušetříme několik instrukcí ve zdrojovém tvaru našeho programu, zato jsme stvořili kanón na vrabce, který je navíc rudým obrem a želvou. Třídicí program je totiž napsán pro mnohem složitější úlohu, než pro jakou jsme ho použili, vyžaduje na vyřešení našeho jednoduchého problému zbytečně mnoho paměťového prostoru a použije složitý algoritmus, v důsledku čehož také spotřebuje podstatně více strojového času než výše uvedený jednoúčelový program.

UFO

„Byl to skutečně zajímavý případ,“ poznamenal Holmes, když naši hosté odešli. „Ukazuje totiž, jak jednoduše se dá osvětlit záhadu, která vypadá zpočátku naprostě nevysvětlitelně.“

A. Conan Doyle

UFO je původně, jak je všeobecně známo, zkratka anglického termínu „Unidentified Flying Object“ (česky „neidentifikovaný létající předmět“, jinak též „létající talíř“). Označuje předmět, nejčastěji tvaru disku nebo doutníku, který se občas záhadně objeví a předvádí podivné kousky — patrně proto, aby pozorovatele přesvědčil, že se nejedná o nic, co by bylo možné nějak rozuměně vysvětlit.

Věda je proti létajícím talířům naprostě bezbranná. I ty největší vědecké kapacity bezradně krčí rameny, když čtou v tisku zaručené zprávy o tom, že „major lucemburského námořního (?) letectva Jean Schmidt-Duval pozoro-

val při letu ve výšce 15 000 stop záhadné létající objekty tvaru oranžových disků, které posléze vtáhly do sebe dva stroje i s piloty, kteří s ním letěli v letce, začaly rušit gravitační pole a ke všemu na něj nějací pidimužíci z těch disků dělali dlouhý nos.“

Létající talíře se nedají vysvětlit vědecky, dají se vysvětlit pouze přirozeně — například tak, že reportér listu „Sober Monthly“ zjistí, že major Schmidt-Duval dostal za svou reportáž od listu „Honky-tonky News“ 10 000 dolarů a že v době, kdy na něj údajně pidimužíci z létajícího talíře dělali dlouhý nos, se nacházel v jenom zapadlém baru ve stavu, ve kterém je spatření pidimužíků a létajících talířů vcelku pochopitelné.

S programem typu UFO je to podobné, s tím rozdílem ovšem, že tvrdé valuty zde nehrájí roli a programy „UFO“ jsou pozorovány i ve střízlivém stavu. UFO je program, který se objeví zničehonic někde, kde by se vlastně vůbec vyskytovat neměl, navíc často nemá kupodivu žádného autora a způsobuje naprosto nevysvětlitelné úkazy. Nicméně takové programy nebo jejich důsledky byly již mnohokrát zaregistrovány.

Jako je těžké UFO popsat, tak je těžké ho i blíže charakterizovat. Uvedeme si aspoň jeden typický a vysvětlený případ.

U většiny větších počítačů (či spíše operačních systémů) jsou zdrojové tvary programů uchovávány ve zdrojových knihovnách a jejich přeložené tvary v tzv. modulových knihovnách (viz slovníček). Název zdrojového tvaru se určuje prostřednictvím řídicího jazyka, avšak název přeloženého programu se u mnoha překladačů určuje ze záhlaví programu samotného.

Často se stává, že z jednoho zdrojového tvaru vytváříme editováním jiný program nebo podprogram s jiným názvem, který se obvykle zadává editovacímu programu. Stačí však zapomenout změnit také název v záhlaví programu samotného a nový program se přeloží do modulové knihovny pod starým názvem. Při pokusu o spuštění nového programu (podprogramu) ohláší operační systém, že program s tímto názvem neexistuje; podíváte se do seznamu zdrojových programů a ověříte si, že musí existovat, a jste si jisti, že jste jej překládali.

Po delším pátrání zjistíte, že program je přeložen pod starým názvem; opravíte proto tento název ve zdrojovém tvaru a ten znova přeložíte. Zatím snadno zapomenete, že jste z modulové knihovny neodstranili verzi, která má obsah nového programu, ale starý název. Až na to po nějaké době zapomenete úplně a náhodou budete pracovat s starým programem, nebudeste se stačit divit, co to ten starý, mnohokrát ověřený program najednou vyvádí — a začnete věřit na UFO.

7. O PROGRAMÁTORSKÉM STYLU

Je-li nějaký způsob, jak něco udělat složitěji a s větší námahou, někdo na něj přijde.

Z Parkinsonových zákonů

Zkušenosti ukazují, že zadáme-li dvacet programátorům naprogramovat tutéž úlohu, dostaneme dvacet různých programů — každý bude chybou v něčem jiném. Existuje nepřeberné množství způsobů, jak i sebeprůhlednější, sebejasnější a sebejednodušší algoritmy naprogramovat tak, aby byly naprostě nepřehledné, nečitelné, komplikované a oplývaly chybami nejrůznějšího druhu a kalibru. Většina programátorů si rychle některé z těchto způsobů oblíbí, a vytvoří si tak vlastní, často velmi osobitý styl.

Potíže začínají, když se takový „stylový“ program dostane do rukou jiného programátorů. Ten obvykle začne brzo prskat a je celý bez sebe z toho, jak strašným způsobem je to naprogramováno. Navíc takový svérazný styl komplikuje život i jeho autorovi, neboť je přičinou vzniku chyb, znesnadňuje jejich odstranění a zhoršuje čitelnost programu. Jaké jsou tedy zásady správného, racionálního programátorského stylu?

Nejprve si trochu ujasněme samotný pojem *styl*. Mluvíme-li v souvislosti s tvorbou nějakého díla (a program je jistě možno za dílo považovat) o stylu, můžeme tím myslet jednak souhrn znaků díla samého, jednak způsob, jakým autor toto dílo vytváří. Styl v prvním smyslu souvisí úzce se srozumitelností programu — z tohoto hlediska jsme se programy zabývali v kapitole *Program nebo džungle*. Zde si všimneme hlavně druhého okruhu problémů, totíž toho, jakým způsobem (či chcete-li stylem) postupovat při programování. Oba významy slova „styl“ spolu souvisejí: dobrý styl programátorské práce silně napomáhá tvorbě stylově dobrých a tedy i srozumitelných programů.

Strukturované programování

Je velmi jednoduché něco zkompplikovat, zato bývá značně komplikované něco zjednodušit.

Tzv. Meyerův zákon

Nejznámější metodikou podporující dobrý programátorský styl je *strukturované programování*. Bylo o něm napsáno mnoho rozsáhlých monografií. Jejich autoři však nejsou zcela jednotni v tom, co všechno se má pod uvedený název zahrnovat. Obecně lze říci, že je to metodika, která napomáhá tvorbě

přehledných a srozumitelných (a proto také spolehlivých a snadno modifikovatelných) programů a kterou lze charakterizovat následujícími základními principy:

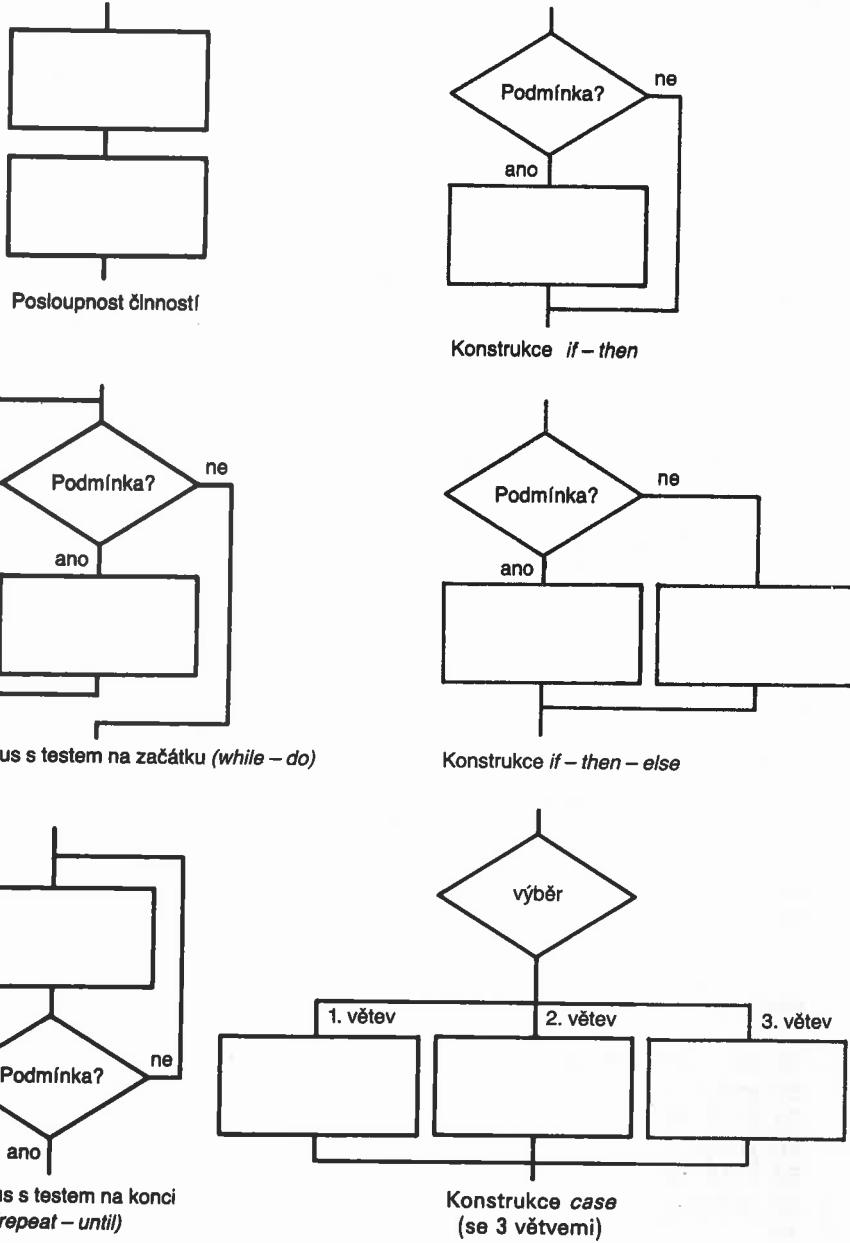
- Program je třeba tvořit systematicky „shora dolů“, tj. začít návrhem jeho celkové logiky s tím, že jednotlivé dílčí části jsou zpočátku popsány pouze v hrubých rysech a postupně se — zpravidla ve více krocích — zpřesňují.
- Program se člení na jednotlivé části (moduly) řešící ucelené dílčí činnosti (strukturovanost „ve velkém“). Moduly nemají mít příliš velký rozsah, aby zůstávaly dostatečně přehledné (obvykle se jako horní hranice uvádí jedna stránka výpisu na počítačové tiskárně, což odpovídá 2 až 3 stránkám na obrazovce).
- Moduly se skládají z bloků navzájem kombinovaných pouze s použitím několika standardních řídicích struktur (strukturovanost „v malém“).
- Souběžně s tvorbou a zpřesňováním algoritmu se specifikují a zpřesňují i datové struktury (viz slovníček). Často se cituje heslo *algoritmy + datové struktury = programy*, jímž je nazvana známá učebnice strukturovaného programování ([WIRT88]). Pro řadu aplikací jsou datové struktury stejně důležité jako algoritmus. V typických úlohách hromadného zpracování dat lze dokonce algoritmus zpracování víceméně automaticky odvodit z datových struktur (na tom je založena populární Jacksonova metoda strukturovaného programování).
- V maximální míře se používají i drobné prostředky zvyšující srozumitelnost programu (viz odst. *Čitelnost programu*).

V programech se mají vyskytovat jen následující řídicí struktury:

1. prostá posloupnost — jednotlivé bloky se provádějí postupně;
2. výběr (konstrukce *if – then*, *if – then – else* a *case*);
3. cyklus (s testem na začátku: *while – do* a s testem na konci: *repeat – until*).

Smysl téhoto řídicích struktur je zřejmý z vývojových diagramů na obrázku 6. Povšimněte si toho, že do každé z uvedených struktur se vchází na jediném místě a rovněž se z ní na jediném místě vychází. To umožňuje vkládat jednotlivé struktury do sebe — každý obdélník na obrázku představuje buď první (dále nestrukturovanou) akci nebo opět kteroukoli ze základních řídicích struktur.

Některé programovací jazyky (například Pascal) obsahují příkazy, které přímo realizují zmíněné řídicí struktury. V takových jazycích jsou ideálně strukturované programy charakterizovány mimo jiné tím, že v nich není použit příkaz **GOTO**. (Ostatně pravděpodobně první článek, kde byly naznačeny ideje strukturovaného programování, se jmenoval „Příkaz GOTO je považován za škodlivý“.) Proto se někdy strukturovanému programování zjednodušeně říká „programování bez GOTO“ a lze se setkat i s názory, že jazyky jako Basic či Fortran neumožňují strukturované programování, protože se v nich bez příkazu **GOTO** obvykle neobejdeme. Takové tvrzení ovšem svědčí o naprostém nepochopení podstaty strukturovaného programování — záleží na přístupu k tvorbě programu a nikoli na konkrétních použitých jazykových pro-



obr. 6

středcích. Základní konstrukce strukturovaného programování je možno převést do jakéhokoli jazyka (třeba i do strojového kódu) a příkaz GOTO bude v mnoha jazyčích právě nástrojem, pomocí něhož budou obecné řídicí struktury do těchto jazyků převáděny.

Podívejme se, jak se dají řídicí struktury z obr. 6 zapsat v Basiku:

while podmínka do příkaz:

```
10 IF opačná podmínka THEN 40
20 příkaz
30 GOTO 10
40 ...
```

repeat příkaz until podmínka:

```
10 příkaz
20 IF opačná podmínka THEN 10
```

if podmínka then příkaz:

```
10 IF opačná podmínka THEN 30
20 příkaz
30 ...
```

if podmínka then příkaz 1 else příkaz 2:

```
10 IF podmínka THEN 40
20 příkaz 2
30 GOTO 50
40 příkaz 1
50 ...
```

case i of 1: příkaz 1; 2: příkaz 2; 3: příkaz 3 end:

```
10 ON i GOTO 20, 40, 60
20 příkaz 1
30 GOTO 70
40 příkaz 2
50 GOTO 70
60 příkaz 3
70 ...
```

Poznamenejme ještě, že sortiment základních konstrukcí není ve všech učebnicích strukturovaného programování stejný: struktury if — then — else, repeat — until a case nejsou nezbytně nutné, neboť je lze nahradit ostatními strukturami; na druhé straně se někdy uvádějí i další struktury (cyklus typu for — do nebo struktura case doplněná větví otherwise).

Modulární programování

Již při popisu strukturovaného programování jsme se zmínili o členění programu na moduly. Přesto se metodika zvaná *modulární programování* často chápe jako alternativa, ne-li přímo popření strukturovaného programování. Při používání modulárního programování se totiž nerespektuje zásada programování „shora dolů“ a zastánci modulárního programování uvádějí pro svůj postup pádné argumenty. Programování „shora dolů“ je totiž krásné a výhodné — pokud programujeme zcela osamocený programový systém, který nemůže využít ničeho, co je již hotovo, a nemůže poskytnout nic, co by se popřípadě dalo použít i jinde.

V praxi se však „na zelené louce“ zpravidla neprogramuje. Ve volném návrhu býváme omezeni tím, že chceme použít již existující moduly, nebo předpokládáme, že některé naše nově vytvořené moduly se mohou v budoucnosti hodit i k jiným účelům. Funkce takových samostatných modulů, jejich návaznost a použité datové struktury jsou potom diktovány i dalšími hledisky a jejich návrh by měl začínat právě u specifikace těchto prvků, a ne u požadavků programu, pro nějž jsou primárně vyvíjeny.

Na druhé straně je strukturované programování natolik rozšířeným a osvědčeným nástrojem, že není možné se ho kvůli zmíněným námitkám vzdávat. Ideální postup je v takových případech asi v kompromisu: univerzální moduly navrhnut (případně i naprogramovat a odladit) předem (s maximálním využitím ostatních zásad strukturovaného programování) a ve zbývající části práce se řídit metodikou strukturovaného programování zcela.

Čitelnost programu

Jak už bylo řečeno, účelem strukturovaného programování (a samozřejmě i ostatních technik pro racionalizaci programátorské práce) je napomáhat tomu, aby programy byly přehledné, spolehlivé, srozumitelné... — řečeno terminologií předcházejících kapitol, aby to byly extrovertní modré přízraky, bílí trpaslíci a hroši. Ke srozumitelnosti (a koneckonců nepřímo i k ostatním kladným vlastnostem) přispívají ve značné míře i zdánlivé maličnosti:

- Používejme v co nejbohatší míře komentářů — speciálně v Basiku může k čitelnosti programu přispět, popíše-li v komentářích řídicí struktury převedené do primitivních basikovských prostředků.
- Označujme proměnné srozumitelnými (mnemotechnickými) názvy: POČET

místo N, SKORE místo I7 apod. (Současně se však vyhýbejme nevhodným zkratkám, o nichž byla zmínka v intermezzu O češtině a „pocitacstine“.) V některých verzích Basiku bohužel nejsou takové názvy přípustné a pokud využijeme toho, že v „našem“ Basiku to jde, riskujeme, že program nebude přenosný na jiný počítač; v tom případě alespoň můžeme označovat jméno jako J\$, součet S, rychlosť V apod.

- Podřízené části řídicích struktur odsazujme o určitý počet sloupců doprava, takže se zviditelní logika programu.

Uvedené zásady ilustruje drobný příklad:

```
10 REM VYTISTENI NEJMENSIHO KLADEHO PRVKA POLE
10 DIM A(100)
20 REM PRECTI POLE
30 MAT INPUT A
40 REM NA ZACATKU M=NEKONECNO
50 LET M=INF()
60 REM PROHLEDEJ CELE POLE
70 FOR I=1 TO 100
80 REM NEKLADENE PRVKY NAS NEZAJIMAJI
90 IF A(I)<=0 THEN 140
100 REM JE MENSNI NEZ DOSAVADNI MAXIMUM?
110 IF A(I)>=M THEN 140
120 REM ZAZNAMEJ JAKO DOSUD NEJMENSI
130 LET M=A(I)
140 NEXT I
150 REM BYL NEKTERY PRVEK KLADENY?
160 IF M=INF() THEN 200
170 REM VETEV "ELSE" (NENALEZEN)
180 PRINT "NEJMENSI KLAONY PRVEK =",M
190 GOTO 220
200 REM VETEV "THEN" (NALEZEN)
210 PRINT "POLE NEOBSAHUJE ZADNY KLAONY PRVEK"
220 END
```

Dialekty programovacího jazyka

Snad pouze jazyk, který je implementován na jediném počítači, existuje pouze v jediné verzi (a to ještě ne vždy). Rozšířenější jazyky existují zpravidla v různých variantách – dialektech. Zejména Basic je na různé dialekty mimořádně bohatý. To umožňuje na určitých počítačích psát v Basiku programy, které by v primitivnějším dialekту vůbec napsat nešly (zvlášť cenné je to u těch mikropočítačů, na nichž se jiný jazyk než Basic nedá použít) nebo nám možnosti poskytované dialektem program zjednoduší. Za tyto výhody však draze platíme tím, že příslušné programy „nechodí“ na počítači, kde je k dispozici jiný dialekt Basiku, a musíme je při přenosu na jiný počítač předě-

lávat. Než použijeme nestandardní prvek jazyka, dostupný jenom v některých dialektech, je proto vhodné rozmyslet si, zda případné komplikace v budoucnosti vyváží to, co získáme.

Za rok se vrátím ...

Čím snadněji se něco udělá, tím obtížněji se to pak mění.

Tzv. Enguv princip

Práce na programu nekončí v okamžiku, kdy je dokončen a funguje. Pokud nějaký program skutečně používáme, přijdeme po čase na to, že je v něm chyba, nebo alespoň na to, že by bylo vhodné v něm něco změnit, a tak se k práci na něm musíme vrátit. To je přirozené – a nebezpečné. Dochází tak postupně k tomu, že od téhož programu máme několik verzí, a ty se nám mohou snadno poplést. Snažíme se pak starou verzi programu provádět akce, které tato verze neumí, nebo ke své nelibosti naopak zjistíme, že nová verze – v důsledku zavlečení chyby – už neumí to, co dřívějším verzím nečinilo potíže.

Zkušení programátoři jsou si těchto úskalí vědomi a svůj programátorský styl zaměřují tak, aby podobné nepříjemnosti omezili nebo alespoň zmírnili jejich dopad. Je to další dobrý důvod, abychom nešetřili komentáři a v průběhu modifikací evidovali změny provedené v programu. Vyplatí se také, když nám program při svém spuštění dá viditelně najevo, se kterou verzí „máme tu čest“. (To znamená, že při vytvoření nové verze musíme modifikovat první zprávu, kterou se program ohlásí.)

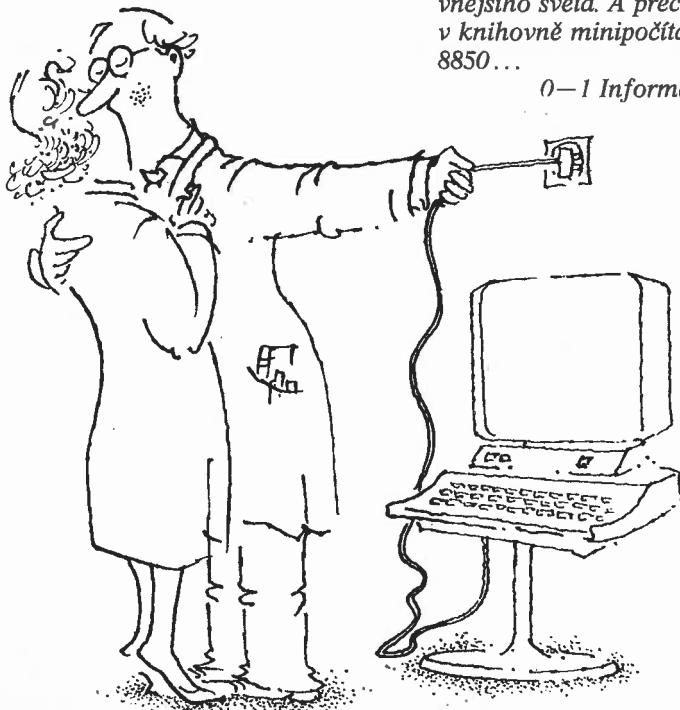
8. INTERMEZZO O SHAKESPEAROVÍ V BASIKU

V informačním letáku jednoho dětského letního tábora s počítačovou výukou (v USA — pozn. autorů) jsou rodiče uklidňováni, že jejich ratolestem nebude dovoleno sedět u terminálů celý den, že budou nuceni též jezdit na koni, plavat či hrát tenis.

M. Bodenová

Vysoké zdi opatství Saint-Wandrille poblíž Rouenu, založeného v 7. století, působí dojmem hráze, o niž se zdánlivě marně tráší nárazy pokroku vnějšího světa. A přece objevíte v knihovně minipočítac Nixdorf 8850...

0-1 Informatique, č. 965



Nedávno jsme vyslechli tento názor na počítače: „Stává se z toho morbidní záležitost. Děti si místo s kuličkami hrají s počítači, počítače jsou ve škole, v podnicích, v úřadech, v domácnostech. Pronikají všude, jsou horší než faráni na sídlištích. Nebude to dlouho trvat a lidé si budou lásku vyznávat v assembleru a Shakespearea překládat do Basiku.“

Zajdou skutečně programátořské poklesy tak daleko? Jsou počítače opravdu morbidní záležitostí? Mohou nás deformovat tak, že ztratíme smysl pro věčně zelený strom života?

Některé názory lze nejlépe vyvrátit tak, že je přijmeme. Proto jsme se zde podali pionýrského úkolu — pokusu o basikovský překlad slavného Hamletova monologu ze stejnojmenné Shakespearovy tragédie.

Protože ani do Basiku se Shakespeare nedá překládat bez znalosti kontextu, připomeňme si velmi stručně děj tohoto klasického dramatu:

Dánskému princi Hamletovi se zjevuje duch a sděluje mu, že jeho otec byl otráven svým bratrem — Hamletovým strýcem, který se touto vraždou zmocnil trůnu. Hamlet chystá pomstu, ale mezičím se utápí ve filozofování a nerozhodnosti. Zabije špehujícího přízivnického ministra Polonia, jehož dcera Ofélia je do Hamleta zamilována. Ofélia se zhroutí a Hamlet je vyhoštěn. Vrací se však a v závěru pomstí svého otce — sám však zahyne. Na dánský trůn nastupuje Fortinbras, muž činu.

A zde je onen slavný Hamletův monolog (v překladu E. A. Saudka):

Žít nebo nežít — to je, oč tu běží:
zda je to ducha důstojnější snášet
střely a šípy rozkacené sudby,
či proti moři běd se chopit zbraně
a skoncovat je vzpourou. Zemřít — spát —
nic víc — a vědět, že tím spánkem končí
to srdcebolení, ta sterá strast,
jež patří k tělu, to by byla meta
žádoucí nade všechno. Zemřít — spát —
Spát. Snad i snít? Á, v tom je právě háček!
To, jaké sny by se nám mohly zdát
v tom spánku smrti, až se těla zbudem,
to, to nás zaráží. To je ten ohled,
jenž bídě s nouzí dává sto let žít.
Vždyť kdo by snášel bič a posměch doby,
sprostoty panstva, útlak samozvanců,
soužení lásky, nedobytnost práva,
svévolí úřadů a kopance,
jež od neschopných musí strpět schopný,
sám kdyby moh svůj propouštěcí list
si napsat třeba šíolem?...

Proveděme nyní stručnou analýzu tohoto textu. V monologu se Hamlet snaží vyřešit několik problémů. Jde o odpověď na otázky:

1. Zda žít, nebo zda nežít.

2. Zda je důstojné ducha snášet střely a šípy rozkacené sudby.
3. Jaké sny by se nám mohly zdát.
4. Kolik lidí je ochotno snášet bič a posměch doby a kopance, jež od neschopných musí strpět schopný.

V jazyku Basic tedy máme napsat program, který na základě vstupních dat (údaje o době, fyzickém a duševním stavu Hamleta apod.) kvalifikovaně na tyto otázky odpoví.

Na první otázkou je odpověď jednoduchá; naše etické normy i nezdolný optimismus prověřený chronickým nedostatkem elektroniky na našem trhu nám jednoznačně velí odpovědět (nezávisle na vstupních datech): „Žít!“ Mohlo by se zdát, že počítačový program dost dobře nemůže na další otázky odpovědět. Existují však programy, které si i s podobnými problémy poradí. Říká se jim *expertní systémy* – viz slovníček. (Expertní systém dokáže například na základě příznaků kvalifikovaně stanovit pravděpodobnou diagnózu pacienta.) Expertní systémy jsou ovšem velmi složité a rozsáhlé. My budeme předpokládat, že máme potřebný expertní systém k dispozici. Náš vlastní program pouze vypíše doporučení expertního systému.

Uvažme však situaci, ve které je monolog prosloven: Hamlet má plnou hlavu jiných starostí (duch, otcův vrah, Ofélie), takže je docela dobře možné, že opomene některé vstupní údaje zadat. Může pak nastat situace, že expertní systém neposkytne na některé otázky odpověď – odpoví třeba otazníkem. Musíme proto zadat tzv. *standardní (default)* odpovědi – to jest takové, které se uplatní, jestliže nerozhodne expertní systém. Na základě obecných zkušeností zadáme standardní odpověď na

1. otázkou: Žít! (to jsme si již vyjasnili);
2. otázkou: Asi ano, rozhodně to je obvyklé;
3. otázkou: To závisí především na životosprávě. Nedoporučujeme používat sedativa!
4. otázkou: Prakticky každý.

Nyní můžeme napsat náš překlad Hamletova monologu do jazyka Basic:

```

10 REM HAMLET - KRALEVIC DANSKY. (C) 1601 W. SHAKESPEARE
20 REM PRECTI HAMLETOVU OTAZKU
30 INPUT O$
40 IF POS(O$, "ZIT NEBO NEZIT")=0 THEN 110
50 PRINT "ZIT!"
60 GOTO 30
70 REM "EXPERT" JE PODPROGRAM REALIZUJICI EXPERTNI SYSTEM
80 REM 1. PARAMETREM JE ZAKLADNI OTAZKA
90 REM DO 2. PARAMETRU ZAPISE ODPOVED NEBO OTAZNIK
100 REM DALSI DIALOG S UZIVATELEM (DOPNUJICI OTAZKY)
105 REM ZAJISTI "EXPERT" SAM
110 CALL EXPERT(O$, V$)
120 IF V$<>"?" THEN 220
130 IF POS(O$, "JE DUCHA DUSTOJN")=0 THEN 160
140 PRINT "ASI ANO - ROZHODNE JE TO OBVYKLE."

```

```

150 GOTO 30
160 IF POS(O$, "JAKE SNY")=0 THEN 190
170 PRINT "TO ZAVISI PREDEVSIM NA ZIVOTOSPRAVE."
175 PRINT "NEDOPORUCUJEME UZIVAT SEDATIVA."
180 GOTO 30
190 IF POS(O$, "BIC A POSMECH DOBY")=0 THEN 220
200 PRINT "PRAKTIKY KAZDY."
210 GOTO 30
220 PRINT V$
230 GOTO 30
240 END

```

S využitím mikropočítače, expertního systému a našeho programu může Hamlet vyřešit své problémy na posezení. Uvažme však, jaký vliv by to mohlo mít na další vývoj děje slavné tragédie. Ten by pak mohl probíhat například takto:

- Rozkolisaný a nevyrovnaný Hamlet svěří své problémy svému osobnímu počítači; počítač problémy obratem vyřeší a doporučuje Hamletovi především zlepšit životosprávu.
- Hamlet přestává ponocovat a denně cvičí aerobic.
- Jeho stav se rychle zlepšuje; slušně, ale důrazně odmítne Ofélie. (Program na korelační analýzu ukázal, že jejich manželství by bylo příliš konfliktní.)
- Pomocí svého osobního počítače a ducha svého otce (pro něhož je hračkou zjistit přístupová hesla) pronikne Hamlet do státní databanky. Aniž by zanechal stopy neoprávněného přístupu k datům, nashromádí důkazy proti svému strýci a Poloniovi. Ti jsou oba odsouzeni: strýc na 10 let za vraždu a Polonius na 15 let za rozkrádání.
- Fortinbras do děje nezasáhne; na doporučení svého osobního počítače se začal věnovat joggingu a zrovna se zúčastňuje Bostonského maratónu.
- Hamlet svěří řízení Dánska svému počítači a sám se začne zabývat ekologií.

Na první pohled je patrné, že využití počítačů v Shakespearově tragédii má své klady i záporu. Z děje se odstranila řada chmurných momentů a zachránili jsme mnoha lidských životů – závěrečný masakr odpadl úplně. Hra získala navíc další výchovné aspekty.

Na druhé straně je třeba konstatovat, že děj se stal natolik stupidním, že by jej nebylo možno použít snad ani pro televizní seriál. Nejhorší však je, že našim překladem Hamletova monologu jsme získali v podstatě triviální a neaplikovatelný program. Nezdá se tedy pravděpodobné, že by – byť i ve vzdálené budoucnosti – hrozilo klasikům překládání jejich děl do programovacích jazyků.

Experiment nás spíše utvrzuje v přesvědčení, že počítače nejsou žádným nebezpečím pro ostatní plody lidské civilizace – naopak věříme, že pomohou všemu dobrému, co společnost vytvořila. K tomu jim dopomáhej všechnoucí lidský rozum.

9. TO BY SE PROGRAMÁTOROVI STÁT NEMĚLO

Chybovat je lidské, ale dokonale něco „zašmodrchat“ — na to je potřeba počítat.

Programátorský folklór

*Může-li něco selhat, pak to selže.
Může-li selhat více věcí, pak selže ta,
která napáchá nejvíce škody.*

Z tzv. Murphyho zákonu

Počítače dovedou snad každému člověku přichystat překvapení. Někoho (zejména laika) udivují tím, co všechno dovedou, jiného (uživatele) tím, co nedovedou, a dalšího (programátora) tím, co mu provedou. Někdy je výsledek práce na počítači skutečně neviditelný.

Mnohý z nás se už jistě setkal s historkou typu „počítač pozval nejstarší obyvatelku Dánska — babičku ve věku 106 let — k zápisu do první třídy“ nebo „ve Spojených státech vzbudil velké pobouření počítač, který několika tisícům školáků zaslal nabídkový katalog pornografických kazet“. Na incidentech tohoto typu není nic záhadného. K prvnímu z nich vedla drobná nedomyšlenost v programu: v datu narození se pro úsporu místa vynechávalo století, takže babiččin věk se spočítal jako 6 let. (Počátkem roku 2000 lze v důsledku úsporného kódování letopočtů očekávat mimořádně vysoký počet infarktů mezi vedoucími pracovníky výpočetních středisek.) Druhý incident je klasickou ukázkou principu, který se v anglicky mluvících zemích označuje slovy „garbage in — garbage out“, což je možno do češtiny převést neméně lakonicky: „jaká data, takové výsledky“. Jestliže operátor omylem nasadí do počítače místo magnetické pásky se seznamem předplatitelů Playboye pásku se seznamem žáků místní školy, není se čemu divit. Každý programátor může z vlastní praxe citovat řadu příkladů ilustrujících chybu vzniklé podobnými „drobnými nedostatkami“ a je mu obvykle na první pohled jasné, kde hledat jejich zdroj. Takovými případy se v této kapitole dále zabývat nebudeme.

Občas se však v programátorské praxi setkáme se situacemi, kdy i zkušenému odborníkovi zůstává rozum stát — dokud se neužodí do čela a nezvolá něco jako „panebože, do čehož to duši dal!“ Dříve, než situace dospěje k tomuto prohlédnutí, může program podle zákona schválnosti způsobit katastrofu. Jindy se program sice nechová nijak nebezpečně, zato se však tvrdosíjně odmítá chovat tak, jak bychom potřebovali.

Úskoky, k nimž se záladné programy uchylují, jsou natolik rozmanité, že je v plné šíři není možné systematicky rozebírat ani podrobněji klasifikovat. Budeme se proto muset spokojit s prostým popisem některých úskalí a dát si pozor, aby se nám něco podobného nestalo.

Nejdražší programátorská chyba na světě

Florida, mys Canaveral, 28. července 1962. Na odpalovací rampě kosmodromu je připravena ke startu nosná raketa s kosmickou sondou Mariner, jejímž posláním je let k Venuši a průzkum této planety.

V řídícím středisku NASA vstupuje odpočítávání startu do poslední, vzrušující fáze. Systémy rakety přecházejí do autonomního režimu, všechny kontrolní údaje svědčí o jejich spolehlivé funkci. Raketové motory jsou zažehnuty. Za jejich ohlušujícího rachotu se raketa zprvu jakoby pomalu zvedá a pak prudce vyráží vzhůru.

Vzrušení pomalu opadá — zdá se, že všechno probíhá normálně. Pojednou se začíná bod, který na hlavním kontrolním panelu řídícího střediska označuje polohu rakety, odchylovat od vypočtené dráhy. Ve středisku nastává horečná činnost — snaha identifikovat chybu a korigovat dráhu letu. Raketa se však odchyluje od vypočtené dráhy čím dál víc. Zákrátka mizí ze zorného pole radarů a vzápětí se zřítí do Atlantického oceánu. Drama trvalo pouhé čtyři minuty.

V řídícím středisku vládne trpké rozčarování. Sonda v Atlantickém oceánu je na nic — výzkum mořských hlubin se dá provádět levnějšími prostředky. Začíná usilovné pátrání po příčinách selhání letu. Výsledek je šokující. Havárii nezpůsobila žádná technická závada, organizační nedostatek nebo materiálová vada, ale — záměna čárky za tečku.

O co šlo: V jazyce Fortran existuje příkaz DO, který se zapisuje např. ve tvaru

DO 3 I=1, 3

Má podobný význam jako v Basiku příkaz

FOR I=1 TO 3

tj. zajistí opakování provedení příkazů, které za ním následují.

Když se připravovalo softwarové zabezpečení letu sondy, děrovačka omylem vyděrovala do štítku místo čárky tečku, takže vznikl příkaz

DO 3 I=1, 3

Na neštěstí tento příkaz není chybný. Protože ve Fortranu (stejně jako v některých dialektech Basiku) nezáleží na mezerách, je takto vyděrovaný příkaz zcela smysluplný: je to přířazovací příkaz, který proměnné DO3I přiřadí hodnotu konstanty 1. 3. To ovšem není to, co se od něj očekávalo, totiž opakování určité části programu třikrát.

K podobným chybám dochází při programování dosti často (viz odstavec Záměna podobných znaků v kap. 12). Avšak skutečnost, že tato chyba nebyla v tak důležitém programu zavčas odhalena, svědčí nejen o záladnosti programátorských chyb, ale v tomto případě i o šlendriánství autora programu, který už bezpochyby v NASA nepracuje; při opravdu solidním otestování programu by taková chyba neměla šanci.

Drobný, okem snadno přehlédnutelný rozdíl mezi tečkou a čárkou stál v tomto případě zhruba deset miliónů dolarů.

Jak se 8 rovnalo 0

Počítačový program dělá přesně to, co mu řeknete, nikdy však nedělá to, co byste chtěli, aby dělal.

Tzv. Greenův 3. zákon

V jednom výpočetním středisku pracovala svého času půvabná programátorka — říkem jí Jarmilka. Jednoho dne nakoukla Jarmilka do kanceláře svých zkušenějších kolegů a nesměle se otázala:

„Může se osm rovnat nule?“

„V podstatě je možné všechno,“ odpověděl konejšivým hlasem jeden z programátorů, „ale že by nivelizace u nás zašla až tak daleko, to se mi přece jenom nezdá.“

„Myslela jsem, jestli se v počítači nemůže — nějak — rovnat nula osmi,“ opravila se s rozpaky v hlase a s rozkošně provinilým úsměvem Jarmilka.

„Ledaže by procesor nule zakroutil krkem,“ pronesl druhý z programátorů a věda, že není úniku, dodal:

„Tak nám přines výpis zdrojového tvaru programu, protokol o překladu a hlášení o chybě, my se na to podíváme.“

Jarmilka se odvědčila cukrblikem a vzápětí přicupitala s programem, který obsahoval chybu stejně půvabnou, jako její autorka.

Program (zde jej přepíšeme do poněkud nestandardního Basiku, umožňujícího použití podprogramů s parametry — viz slovníček) obsahoval mj. příkazy

```
10 DIM X(8)  
20 FOR I=1 TO 5  
30 CALL PODPROG(8)  
40 PRINT X(1)  
50 NEXT I
```

...

```
100 SUB PODPROG(I)  
...  
180 LET I=INT(I/2)  
190 SUBEND
```

Při provádění řádku 40 se ohlásila chyba „Index je větší než počet prvků v poli“. Jelikož se měl tisknout první prvek a pole mělo 8 prvků, bylo to skutečně podivuhodné. Výpisem obsahu paměti Jarmilka zjistila, že instrukce, která měla srovnávat index s délkou pole, porovnávala ve skutečnosti index 1 s nulou a nikoli s osmičkou, a moudře s tím zašla za svými zkušenějšími kolegy.

Teprve podrobným prohlédnutím programu příkaz po příkazu byl šotek, který si s Jarmilkou zahrával, objeven. Podprogram PODPROG totiž mění hodnotu svého formálního parametru. Protože podprogram je v řádku 30 volán tak, že jako skutečný parametr je zapsána konstanta 8, při prvním vyvolání podprogramu se hodnota této „konstanty“ změnila z 8 na 4, při druhém vyvolání na 2, při třetím na 1 a při čtvrtém na nulu. V tom okamžiku je ovšem in-

dex (který je rovný jedné) skutečně větší než délka pole, vyjádřená osmičkou, z níž se však zatím stala nula.

Poznamenejme, že mnoho překladačů (zejména interpretačních, které jsou u Basiku obvyklejší) je vůči takové chybě imunní — konstanta 8 zapsaná na jednom místě programu nemá nic společného s touž konstantou zapsanou jinde. U překladačů komplujících program do strojového kódu je sjednocení všech stejných konstant do jediné konstanty mnohem pravděpodobnější.

Chyba, o které tu byla řeč, je vlastně speciálním případem nebezpečného druhu chyb — omylu ve vzájemném přiřazení formálních a skutečných parametrů. K takovým chybám může docházet v jazycích, které dovolují psát podprogramy s parametry. Napíšeme-li například (v dialekту Basiku, který to dovoluje) podprogram:

```
200 SUB VYPOC(X, A())
```

... a při vyvolání oba parametry přehodíme:

```
100 DIM P(50)  
100 CALL VYPOC(P(), 0.5)
```

má proměnné X odpovídá pole P a poli A konstanta 0.5. Pokud překladač neodhalí, že volání je chybné, můžeme očekávat při spuštění programu nejrůznější překvapení. Totéž platí o případu, kdy při vyvolání zadáme jiný počet parametrů, než má podprogram mit.



Žádná programátorská úloha není tak jednoduchá, aby se na ní nedalo něco pokazit.

Programátorský folklór

Způsobů, jak si nechtěně vymazat z disku soubor, je mnoho — tím více, čím je soubor důležitější. Jeden z nich jsme již poznali v odstavci *Šrapnel* v kap. 6. Chybami se člověk učí, a proto si každý trochu zkušenější programátor dává velký pozor na to, aby si omylem soubory nevymazal. Čert však nikdy nespí a vždycky dokáže najít další způsob, jak nám ukázat, že nejsme neomylní. Na jednoho z našich kolegů si počítač nachystal rafinovanou fintu. Stačilo trochu rozptýlit jeho pozornost a kolega si zdrojový program uložil omylem do modulové knihovny (určené pro programy přeložené do strojového kódu). To operační systém dovoluje a zdánlivě se nic nemůže stát — snad až na drobné potíže, které vzniknou, až se bude hledat, kde je zdrojový text uložen.

Avšak situace se vyvíjela jinak. Kolega nezapomněl, že zdrojový text má v modulové knihovně, a bez problémů jej odtud přeložil do strojového kódu. Bývá dobrým zvykem, že název zdrojového a přeloženého programu se liší pouze příslušností k různým knihovnám. V okamžiku, kdy překladač zapisoval přeložený program na disk, zjistil, že v knihovně přeložených programů již program daného jména je. To zpravidla znamená, že překládáme novou verzi téhož programu a dosavadní verzi proto operační systém vymaže, protože je již zastaralá. Tentokrát ovšem nebyla vymazána stará verze přeloženého programu, ale pracně vytvořený zdrojový program.

V tomto případě se na zničení souboru podílel operační systém. Daleko snáze si však soubor může smazat programátor sám; stačí například z nepozornosti příkázat, aby se výsledky výpočtu místo do nového souboru zapisovaly do souboru, který již obsahuje užitečná data.

Ten počítač je dneska nějak líný

Jestliže se určitým programem často zpracovávají podobná data, má už každý, kdo s touto úlohou přichází do styku, představu o tom, jak dlouho může takový výpočet trvat. Většinou se odhad od skutečné doby příliš neliší. Pokud program odhadnutý čas překračuje a příslušný pracovník se stále nemůže dočkat výsledků, prohodí něco, co připomíná větu v nadpisu tohoto odstavce, a je si jist, že konec výpočtu nastane každým okamžikem. Obvykle se nemýlí, ale někdy počítač projevuje takovou zálibu v určitém programu, že se s ním zřejmě nehodlá rozloučit. Potom je programátor (uživatel, operátor ...) jako na trní, po nějaké době to již nevydrží a program „típne“. Prohlédne si výsledky zapsané někde ve vnější či vnitřní paměti nebo protokol o průběhu výpočtu a zakleje. Zjistí totiž, že do konce zbývaly dvě sekundy, nebo že program

narázil na skrytu chybu a celou dobu trávil v nekonečném cyklu na samém začátku.

Může se však stát, že se zdržení takto nevysvětlí a výpočet nakonec úspěšně skončí — se správnými výsledky — po desetinásobku nebo stonásobku obvyklé doby. Na pracovišti autorů se to stalo několikrát a zjistit příčinu nebylo jednoduché. Nakonec však podrobný rozbor v kombinaci se šťastným postřehem programátora odhalil, že pomalý výpočet neměla na svědomí lenost počítače, ale — jeho příliš dokonalý operační systém. Ten totiž dovoluje jednak hlídat i takové chybové stavby, které se u jiných počítačů za chybu nepovažují, jednak takové chyby ošetřit provedením opravného podprogramu. Ve zmíněném případě se jednalo o práci s tak malými čísly, že jejich součin již nelze v počítači zobrazit. Při každém pokusu o násobení těchto malých čísel došlo k podtečení (viz slovníček) a ke slovu se tak dostal příslušný modul operačního systému. Ten po rozpoznání charakteru chyby vyvolával speciální podprogram, jehož úkolem bylo pokračovat, jako by se nic nestalo. Tak došlo k tomu, že vždy zhruba po milisekundě byl výpočet přerušen a zpracování tohoto přerušení trvalo sto milisekund. Není divu, že se výpočet stokrát zopamil. Náprava byla jednoduchá — stačilo počítači prostě přikázat, aby podtečení za chybu vůbec nepovažoval.

U mnoha operačních systémů se sice nesetkáte s touto nepříjemností, ale zato narazíte na jiné podobné. A proto dovolte jednu obecně platnou radu: až bude váš počítač líný, podívejte se, zda nespotřebuje většinu svého času opakováním opravování chyb — třeba při čtení ze špatně seřízené disketové mechaniky.

Jinou častou příčinou mimořádné pomalého zpracování je nevhodné využívání virtuální paměti (viz slovníček). Ta sice představuje významnou pomoc pro řešení paměťové náročných problémů, avšak programy se jí musí patřičně přizpůsobit. Ideální je, když program jednotlivé veličiny (např. prvky velkého pole) zpracovává v pořadí, v němž jsou uloženy v paměti. Například matice definovaná popisem

10 DIM M(9, 499)

obsahuje 5000 čísel v pořadí

M(0, 0) M(0, 1) M(0, 2) ... M(0, 499) M(1, 0) M(1, 1) atd. Dejme tomu, že potřebujeme určit největší prvek v této matici. Program

100 LET X=M(0, 0)

110 FOR I=0 TO 9

120 FOR J=0 TO 499

130 IF M(I, J) <=X THEN 150

140 X=M(I, J)

150 NEXT J

160 NEXT I

prohledává prvky matice v pořadí, v němž jsou uloženy v paměti. I na počítači s virtuální pamětí bude pracovat tento program efektivně. Prohodíme-li navzájem vnitřní a vnější cyklus, bude program rovněž pracovat, ale matice se bude probírat napřeskáčku, potřebné prvky zpravidla nebudu připraveny

ve vnitřní paměti a virtuální paměť bude splašeně čist z disku jednu stránku za druhou. Výpočet bude proto velmi pomalý (v praxi tak lze hravě docílit toho, že za hodinu skutečného času bude procesor pracovat pouze několik desítek sekund, zato však bude přeneseno 250 tisíc bloků z disku).

Jak využít porouchaný počítač

*Jestliže něco nemůže dopadnout špatně, stejně to špatně dopadne.
Tzv. Schnatterlyho souhrn logických důsledků*

*Příroda nadřhuje skrytým chybám.
Z tzv. Murphyho zákonů*

V jednom výpočetním středisku se porouchal počítač a bylo nutné čekat několik dní na náhradní díl. Porucha zasáhla pouze část aritmetické jednotky, zajišťující výpočty v pohyblivé čárce, tj. operace s reálnými čísly. Ostatní operace (včetně výpočtů s celými čísly) obstarávají na většině počítačů jiné jeho části. Protože čas velkého počítače je drahý, napadlo pracovníky výpočetního střediska, že by zatím mohli zpracovávat programy, které s reálnými čísly nepracují. Ve výpočetních střediscích zaměřených na zpracování administrativních dat je takových programů celá řada (evidence základních prostředků, zpracování textů, možná i výpočet mezd a mnoha jiných).

Do příchodu očekávaného náhradního dílu tedy středisko vesele pracovalo a jenom několik programů, u nichž se muselo počítat s reálnými čísly, čekalo.

Za pár dní byl počítač opraven. Po několika dalších dnech však začaly docházet reklamace na chybné výsledky. (Výstupní sestavy po nějakou dobu putují k uživateli a další chvíli trvá, než si uživatel dodané výsledky prohlédne — pokud si na to vůbec najde čas.) Pracovníci střediska sice tušili, že to bude souviset s porouchanou jednotkou, ale důvod se zdál záhadný. Vždyť to byly programy, které skutečně s reálnými čísly nepracovaly, a dokonce byly reklamovány i výpočty provedené až po opravě počítače. Tepřve po delším bádání a spoustě přesčasových hodin se zjistilo, že postiženy byly programy, které se v době poruchy překládaly z jazyka Cobol (bez ohledu na to, zda výpočet podle nich byl zpracován při poruše nebo až později). Výsledné programy totiž skutečně jednotku pro práci s reálnými čísly nepoužívaly — zato však její registry (viz slovníček) používal cobolský překladač k uložení některých pracovních dat!

Důvěřuj, ale prověřuj

Příčiny, které mohou zruinovat muže, jsou tři: ženy, hazard a bezmezná důvěra k odborníkům.

Georges Pompidou

Nedůslednost je to jediné, v čem jsou lidé důslední.

John Steinbeck

Už před více než dvaceti lety se jeden z autorů názorně přesvědčil o oprávněnosti této zásady. Zkoušel programovat na jistém (tehdy velmi rozšířeném) počítači v Algolu 60. Při zpracování jednoho programu narazil na velmi zajímavou okolnost: V programu se bezprostředně po sobě vyskytovaly tři téměř stejné příkazy obsahující poměrně složité aritmetické výrazy. Jeden z těchto příkazů dával na první pohled nesmyslné výsledky, ačkoli ostatní dva se zpracovávaly správně. V zápisu příkazu nebyla žádná chyba. Když se do programu vložil ladící tisk, přestávala se chyba do sousedního příkazu. To již napovídalo dosti zřetelně, že chyba tentokrát výjimečně nebude v programu, ale v překladači. A skutečně: výpis přeloženého programu ve strojovém kódu ukázal, že hodnota části aritmetického výrazu se ukládá dočasně na jistou adresu v paměti a o několik instrukcí dále se vybírá z paměti zpět — avšak z adresy o 2 vyšší.

Chyba v překladači byla reklamována u výrobce počítače a program se předělal tak, aby chybu obešel. Asi po roce došla od výrobce opravená verze překladače. (To v té době zdaleka nebylo samozejmé!) Autor vyhrabal ze „spodních geologických vrstev“ na psacím stole původní verzi inkriminovaného programu a ze zvědavosti ji nechal přeložit novým překladačem. S údivem zjistil, že jistý pokrok byl sice patrný, ale oprava byla provedena „poněkud nedůsledně“. Tentokrát se totiž odložená pomocná hodnota vybírala z adresy pouze o 1 vyšší.

Podotkněme, že tento příklad je naprostě netypický. To, že nefungující program má na svědomí operační systém nebo překladač, je zcela mimořádné a ten, kdo na takovou možnost hned zapomene, udělá dobré — ostatně v odstavci *Chyba je v počítači* v následující kapitole to ještě zdůrazníme.

A proč tedy uvádíme tak netypický případ? Za prvé je natolik pozoruhodně unikátní, že by byla škoda se o něm nezmínit. Druhý důvod je však významnější. Aby se z tohoto atypického případu stal typický, stačí v něm nahradit pojmem „operační systém“ pojmem „převzatý softwarový program“. Platí-li pro operační systém „důvěřuj“, platí pro přebírané programy v plné míře „důvěřuj, ale prověřuj“. Příklad toho, jaké pozoruhodné chyby lze najít v převzatých softwarových programech, nalezne čtenář v odstavci *Nerespektování změny obsahu proměnné* v kap. 12.

10. OMYL A JEHO DŮSLEDKY

*„Ti by na takovou vzdálenost netrefili
ani vrata od stod ...“*

*Poslední slova generála
Johna Segdwicka, která
pronesl pohlížeje z palisády
na nepřátelské linie během
bitvy u Spotsylvania r. 1864*

Omylů se dopouští každý z nás, i když naštěstí obvykle nemívají tak chmurné a nenapravitelné následky, jako ten poslední generála Segdwicka. Proti tomu zřejmě moc nenaděláme. V životních omylech nám poslouží psychologické, psychiatrické, právní, manželské a různé jiné poradny, nehledě na stohy příslušné literatury (než ji přečtete, zahojí rány čas) a poměrně dobrou situaci v zásobování pivem. Drobné omyly v programování počítač suše okomentuje a v nejhorším poradí zkušenější přítel.

Zde se však zmíníme o některých omylech, které jsou mezi programátor-skou obcí rozšířenější nebo kterých se často dopouštějí zvláště začínající programátoři. Některé z nich jsou přitom takového druhu, že by snad bylo lépe miuvit o pověrách.

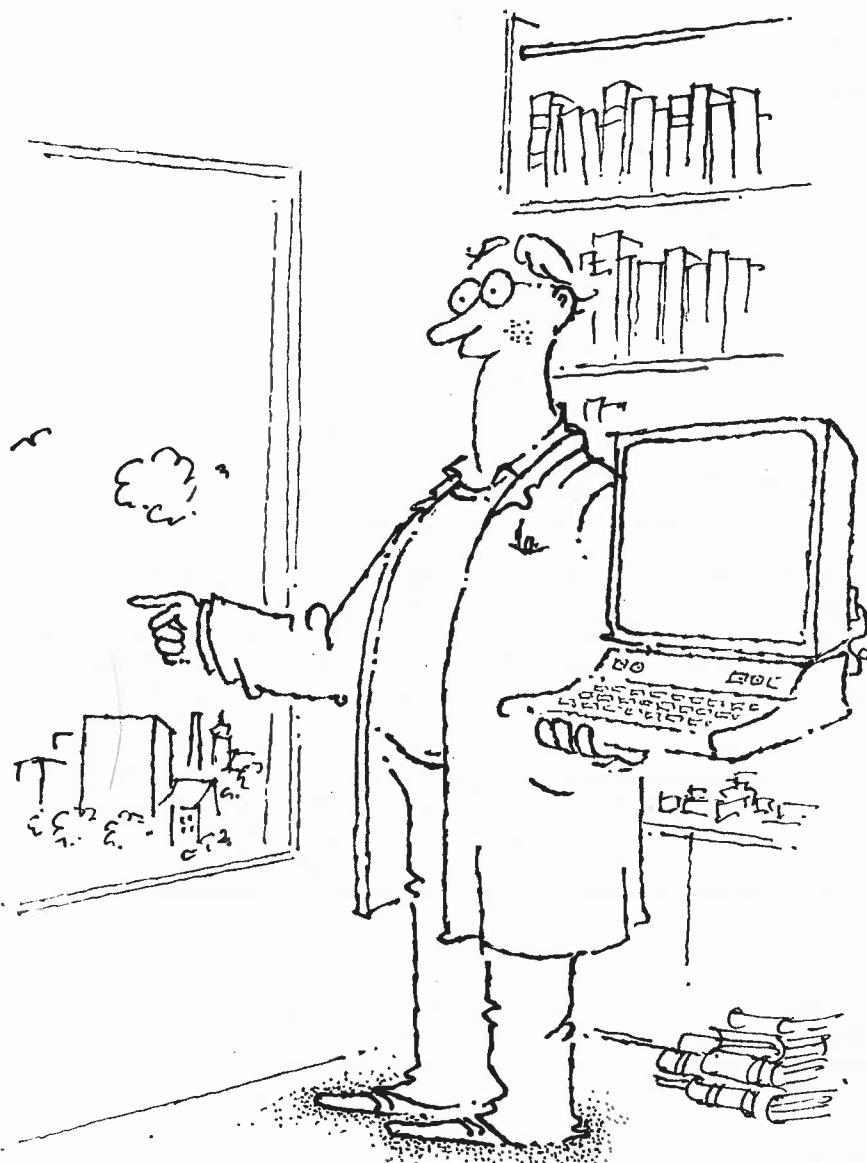
Chyba je v počítači

*U pramene každé chyby počítače jsou
nejméně dvě chyby člověka. Jedna
z nich je ta, že z chyby viní počítač.
Programátorský folklór*

Představa, že za nefungující program může počítač, je mezi programátor-skými omyly a pověrami fantómem. I zkušení programátoři čas od času podléhají tomuto sugestivnímu klamu.

Abychom upřesnili, o čem je řeč — nemáme tu na mysli hardwarové chyby, které mají za následek, že se výpočet přeruší a je zpravidla ztracen („odejde“ procesor, paměť apod.). Chyby tohoto druhu jsou bohužel na některých u nás rozšířených počítačích časté. Jde nám o situaci, kdy program nepočítá, jak by měl, a programátor je skálopevně přesvědčen, že chyba není v programu. Kde jinde by potom ovšem měla být, když ne v počítači nebo v operačním systému.

Takový úsudek — byť podvědomý — je ovšem obvykle důsledkem naprostého zoufalství a ve většině případů mu předchází období střízlivého uvažo-



vání, ve kterém programátor usilovně hledá chybu ve svém vlastním programu. Když však už po sté bezvýsledně prochází svůj původně dvacetřádkový program (ze kterého přidané kontrolní tisky udělaly dvacetřádkovou obluď), když vyčerpal všechny osvědčené triky na polapení proklínáné chyby a když nepomohli ani zkušenější přátelé, začne si nešťastník v důsledku vyčerpání a černé beznaděje pohrávat s hříšnou myšlenkou: Chyba je v počítači. Poté, co definitivně podlehne tomuto klamu, se situace dříve či později vyjasní, ale vždy v intencích tzv. zlatého pravidla programátorských poklesků, které zní:

Programátor, který uvěří, že viník je počítač, si „určíne kšandu“.

Můžete si být stoprocentně jisti, že až se všem svým známým svěříte s tím, jaké neuvěřitelné věci vám počítač provádí, že to je strašné, jak nespolehlivým operačním systémem je vybaven atd., zjistíte například, že jste pracovali s modulem, který neodpovídá výpisu zdrojového programu. Několik dalších let se vás však nikdo ze známých neopomene pozeptat na záhadnou chybu v operačním systému a bude vám znova a znova připomínat, abyste ji reklamovali u příslušné firmy.

To v lepším případě. Pokud v záchravu rozhořčení nad jeho zlomyslnostmi vyhodíte počítač oknem, přijdete na svou chybu zpravidla těsně před jeho dopadem. (Tento způsob vyhledávání chyb jsme však neověřovali a vzhledem k možnému ohrožení chodců ho ani nedoporučujeme.)

Na druhé straně musíme připustit, že i v operačních systémech se chyby vyskytují (viz odstavec *Dívčík ale prověřuj* v předchozí kapitole) — ale pravděpodobnost toho, že na takovou chybu narazíte, je nesmírně malá. Operační systémy jsou totiž ty nejdůkladněji testované a nejpečlivěji vytvořené programy. Rovněž možnost vzniku hardwarové chyby, která by se neohlásila a způsobila např. změnu hodnot výsledků, je tak malá, že o ní prakticky nemusíme uvažovat. Pokud chcete namítout, že se to přesto může stát, musíme přiznat, že tato možnost je nevyvratitelná. Podobně je ovšem nevyvratitelná také např. následující možnost, uveřejněná před lety M. Holubem: *Z nejbližší hvězdy Proxima Centauri k nám neustále vysílají rádiové depeše Morseovkou. Neslyšíme je jenom proto, že tečka u nich trvá 3 miliardy let a tedy právě vysílají mezeru v písmenu H.*

Programátor, který věří na výskyt chyb v operačním systému vzdor jejich velmi malé pravděpodobnosti, by měl být důsledný a neměl by:

- chodit po ulicích (může ho něco přejít);
- bydlet v domě (dům může spadnout);
- dýchat (znečištění vzdúchu může přivodit choroby).

Striktní dodržení těchto dobré (jak jinak) míněných rad by mělo podstatně omezit počet programátorů, kteří ze svých chyb obviňují počítač.

Počítá-li program správně na jednom počítači, bude počítat správně i na druhém počítači

Na každou otázku existuje jednoduchá, snadno pochopitelná nesprávná odpověď.

Z tzv. Murphyho zákonu

V názvu tohoto odstavce se odráží typická, leč klamná představa začátečníků. Nelze jim to vyčítat, protože logicky vzato, mělo by tomu tak opravdu být. Zkušenější programátor však ví, že pravdou je spíše pravý opak.

Nejenže program obvykle nebude počítat správně na jiném počítači, on možná nebude počítat správně ani na též typu počítače vybaveného jiným operačním systémem. Nejenže program nemusí počítat správně dokonce ani na též počítači, pod jiným operačním systémem, on možná nebude počítat správně ani na též počítači s týž operačním systémem, pokud ho přeložíme jiným překladačem. A nejenomže program v takových situacích nemusí počítat správně, on možná nebude počítat vůbec. Kromě toho jej možná vůbec nepřeložíte.

Důvodů pro tato tvrzení je několik. Různé překladače většinou realizují poněkud odlišné verze programovacího jazyka. Zejména u Basiku se jednotlivé verze od sebe často velmi podstatně liší. A i když třeba dva překladače používají přesně stejnou verzi jazyka, bude se lišit způsob, jak se program přeloží do strojového kódu, nebo jak se interpretuje v případě interpretačního překladače. Navíc v různých počítačích se čísla často zobrazují různým způsobem, s různou přesností, s různým rozsahem povolených hodnot apod.

I zcela správný program může proto na jiném počítači havarovat například proto, že tento počítač nemůže pracovat s čísly většími než 10^{19} , která se v průběhu výpočtu objevila, nebo s textovými proměnnými delšími než 255 znaků. V takovém případě alespoň víme, že výpočet havaroval; může se však stát i to, že výpočet proběhne, ale výsledky budou chybné. Původní počítač například počítá s čísly zobrazenými s přesností na 12 dekadických číslic, druhý počítač však pracuje pouze se 7 platnými číslicemi a zaokrouhlovací chyby pak způsobí, že konečný výsledek je zatížen chybou větší než 100 %.

Bez vývojového diagramu nelze programovat

Udělat něco obtížným způsobem je vždy snazší.

Tzv. Murphyho paradox

V dobách, kdy se programovalo ve strojovém kódu počítače, to byla pravda, a také by tehdy asi málokoho napadlo vytvářet složitější program jiným způsobem. Programování ve strojovém kódu bylo ovšem velmi nepohodlné a napsané programy byly nepřehledné. Názor uvedený v titulku odstavce je

však kupodivu — jako jakýsi druh atavismu — poměrně rozšířený i v současnosti.

V dnešní době se však programuje téměř výhradně ve vyšších programovacích jazycích (Basic, Fortran, Pascal, Ada atd.) a při použití zásad strukturovaného programování bývá ve většině případů jednodušší a pohodlnější napsat program přímo v programovacím jazyce. Zápis takto vytvořeného programu přitom nemusí být o nic méně přehledný než příslušný vývojový diagram. Tento fakt vyplývá přinejmenším z vlastní programátorské praxe; dobrí profesionální programátoři dnes používají vývojové diagramy spíše výjimečně.

Ve Spojených státech byl proveden seriózní experimentálně podložený výzkum zaměřený na vhodnost používání vývojových diagramů v programování. Jednalo se nejen o použití vývojových diagramů při vlastním programování, ale i při výuce programovacích jazyků, při ladění a modifikování programů a pro dokumentaci. Výsledky dopadly pro vývojové diagramy vesměs nepříznivě. Přitom zde vývojovým diagramům konkuroval jazyk Fortran založený na staré normě (Fortran 66), který je již zastaralý a není strukturovanému programování dobře přizpůsoben.

Nechceme však vývojové diagramy zcela zatracovat. Podle našeho názoru existují situace, kdy je jejich použití na místě. Jedná se zejména o tvorbu některých programů se složitou logickou strukturou. Otázka, zda je, či není vhodné vytvářet vývojový diagram, závisí i na vyspělosti a stylu samotného programátora. Také ve výuce programování mají vývojové diagramy své místo — mohou například pomoci při vysvětlování řídicích struktur (viz kapitola *O programátorském stylu*) nebo některých programátorských konstrukcí. Navíc jsou dobré srozumitelné i bez znalosti programovacího jazyka, a tím se přirozeným způsobem stávají jakýmsi „referenčním“ algoritmickým jazykem. (Vývojovými diagramy a dalšími grafickými metodami pro znázornění programů a algoritmů se zabýváme v příloze 2.)

Basic a Fortran jsou zastarálé jazyky odsouzené k zániku

Jazyk, v němž se v roce 2000 budou programovat vědeckotechnické výpočty, bude možná vypadat úplně jinak než dnes, ale bude se jmenovat Fortran.

Nejmenovaný člen komise pro normalizaci Fortranu

Názor uvedený v nadpisu se velmi rozšířil s růstem popularity myšlenek strukturovaného programování a pro některé jeho skalní přívřenze jsou dnes Basic a Fortran symboly všeho zla a všech neřestí v programování. Argumenty proti těmto jazykům byly skutečně závažné — jak klasický Fortran tak i klasický Basic například nedisponují ani základním „strukturovaným“

příkazem if — then — else a v mnoha směrech jsou výrazně pozadu za modernějšími jazyky.

Na druhé straně musíme Basiku i Fortranu přiznat určité výhody. Především jsou to jazyky velmi jednoduché, které se i začátečník snadno naučí. Kromě toho velká většina vědeckotechnických programů je naprogramována právě ve Fortranu nebo v Basiku.

A tak se stalo, že prognózy o zániku těchto „mrtvých“ jazyků selhaly, spousta programátorů v nich programuje dál a programů v těchto jazycích stále přibývá. Jeden čas se zdálo, že se programátorský tábor rozdělí na teoretiky, kteří postupně dokáží, že Basic a Fortran dávno zanikly, a na praktiky, kteří v nich budou dál veselé programovat a navíc ještě troufale vykřikovat, že jim je úplně jedno, jak zatraceně hodný jazyk používají, jen když se v něm dobré programuje a když se navíc dá tak výhodně používat té spousty v něm již nařízených programů.

Místo války o Basic a Fortran však nakonec došlo k rozumnému kompromisu, kterým jsou nové normy těchto jazyků. V případě Basiku je to norma, jejiž návrh byl publikován v roce 1985 a která znamená velmi podstatné zdokonalení jazyka. Podobně pro Fortran byla již v roce 1978 schválena nová norma, všeobecně označovaná jako Fortran 77. Obsahuje (až na nepatrné výjimky) celý klasický Fortran a navíc řadu důležitých rozšíření, např. konstrukci if — then — else, textové proměnné, které v klasickém Fortranu neexistovaly, aj. Tato rozšíření přibližují Fortran moderním programovacím jazykům a zároveň si Fortran ponechává svou velkou výhodu — možnost používat velké množství programů, které v něm již byly vytvořeny. (Případně zájemce o tento jazyk odkazujeme na [HŘEB89]). Navíc se již nyní počítá s jeho další modernizací, která má zatím předběžný název Fortran 8x. Tento jazyk bude — při zachování naprosté návaznosti na Fortran 77 — obsahovat natolik zásadní změny, že by měl uspokojit i ty nejnáročnější.

I moderním variantám Fortranu a Basiku lze samozřejmě leccos oprávněně vytknout (např. to, že dnes již sice umožňuje dodržovat dobrý programátorský styl, ale na druhé straně k němu své uživatele nijak zvlášť nevedou). Přesto tyto jazyky, navzdory silné konkurenci a kritice, dál žijí a jejich zánik je v nedohlednu.

Násobení je mnohokrát pomalejší než sčítání

Kořeny tohoto omylu spočívají v historii vývoje počítačů. U některých počítačů to skutečně byla pravda — zejména tam, kde pro sčítání byla k dispozici strojová instrukce, zatímco násobení bylo nutno provést pomocí podprogramu. Ačkolи tyto časy jsou již za námi (snad s výjimkou některých mikropočítačů), je představa o tom, že násobení je podstatně pomalejší než sčítání, dodnes zakořeněna mezi programátory i mezi matematiky. Jedním z důvodů je zřejmě i to, že řada algoritmů byla zvlášť v dřívější době vyvýjena tak, aby počet násobení byl co nejmenší, i na úkor sčítání a operací spojených s orga-

nizací výpočtu. V mnoha případech byla složitost algoritmu charakterizována pouze odhadem počtu násobení a dělení (dělení je o něco pomalejší než násobení a odčítání je prakticky stejně rychlé jako sčítání).

I když je poměr rychlostí aritmetických operací závislý na typu počítače, můžeme si udělat jistý obrázek o časové náročnosti aritmetických operací z následující tabulky, která udává, kolikrát je na některých počítačích násobení reálných čísel pomalejší než jejich sečtení.

Počítač	Jednoduchá přesnost	Dvojnásobná přesnost
EC 1033	3,0	5,4
EC 1040	1,7	4,1
ICL 2950/10	1,9	3,0
PC/AT*	1,03	1,04

Je vidět, že při odhadování časové náročnosti algoritmů musíme brát v úvahu všechny aritmetické operace. Nezanedbatelnou roli mohou však hrát i ostatní příkazy. Proto je dobré při analýze programu provést rozbor algoritmu z hlediska jeho implementace na konkrétním typu počítače. Je do jisté míry pikantní, že i dnes se lze setkat s návody, jak řešit počítačový čas např. tak, že příkaz

100 LET Z=8*X

nahradime příkazem

100 LET Z=X+X+X+X+X+X+X

Přitom je ve skutečnosti druhý příkaz snad na všech počítačích a ve všech programovacích jazycích pomalejší.

Podobného zpomalení výpočtu se může dosáhnout i „rafinovanějšími“ způsoby, které navíc obvykle zatemní použitý algoritmus. Tak například můžeme vynásobit X osmi pomocí následujícího „triku“:

100 LET Z=X+X

110 LET Z=Z+Z

120 LET Z=Z+Z

I toto naprogramování bude však na většině počítačů pomalejší než prosté **LET Z=8*X**.

Podobně se řada omylů vztahuje na odhad doby výpočtu základních matematických funkcí, jako je druhá odmocnina, goniometrické funkce, logaritmus apod. Ty se většinou provádějí pomocí speciálních podprogramů, u některých počítačů jsou pro ně dokonce k dispozici speciální strojové instrukce. Vzhledem k tomu a s ohledem na závislost rychlosti výpočtu na konkrétním komplátoru se tyto doby mohou značně lišit, avšak alespoň hrubé srovnání s dobou provádění aritmetických operací je možno provést.

Přesto se poměrně často vyskytují dva extrémní názory. Jeden, že rychlosť

*) S procesorem 80286 a matematickým koprocessorem 80287.

vyčíslení těchto funkcí je srovnatelná se základními aritmetickými operacemi, a druhý, že jsou nesrovnatelně (řekněme tisíckrát) pomalejší. První názor může mít začátečník, který o tom nikdy nepřemýšel. Kdo se někdy zabýval studiem toho, jak se dají elementární funkce approximovat pomocí řad, řetězových zlomků a jiných matematických triků, bude se asi ve většině případů přiklánět k druhému názoru, neboť pro dosažení dostatečné přesnosti běžnými metodami je zpravidla zapotřebí velkého množství kroků.

Avšak algoritmy, jimiž se obvykle elementární funkce realizují, jsou těmi nejvybranějšími „modrými přízraky“. Počet aritmetických operací, které na výpočet funkční hodnoty potřebují, se pohybuje přibližně mezi deseti až dvaceti. (Pokud je výpočet funkce realizován podprogramem, přistupují ovšem ještě další „režijní“ instrukce spojené se samotným voláním podprogramu.) Uvážíme-li, že tyto algoritmy dávají výsledek s prakticky maximální přesností (jsou voleny většinou tak, že chyba použité matematické metody je srovnatelná s chybou způsobenou tím, že čísla se v počítači zaokrouhlují na určitý počet míst), musí nás jejich dokonalost fascinovat. Avšak porozumět takovým algoritmům — to je záležitost pro profesionály-specialisty.

Poznamenejme, že na druhé straně je výpočet funkcí přece jenom natolik pomalý, že na rozdíl od nahradby násobení sčítáním se zpravidla vyplatí nahradit nepříliš vysokou mocninu násobením (jak bylo popsáno v odstavci *Modrý přízrak*, kap. 6).

Je zřejmé, že není nutné zabývat se dobou trvání jednotlivých příkazů, pokud program neobsahuje úseky, které se budou mnohemrát opakovat. „Mnohemrát“ zde znamená, že počet opakování bude u domácího počítače převyšovat řekněme několik set či tisíc, u klasického počítače deset či sto tisíc nebo ještě více. V programech, kde se některé skupiny příkazů tolíkrát opakují (např. proto, že jsou zapsány uvnitř několika cyklů vložených do sebe), mohou však uvedené omyly rozhodnout o použitelnosti programu.

Dodejme, že tyto úvahy ztrácejí do značně míry platnost, pokud pracujeme s interpretačním překladačem (viz slovníček), tj. například v Basiku na většině domácích počítačů. V takovém případě je doba vlastního provádění operace zanedbatelná proti času, potřebnému na analýzu zdrojového textu a hledání proměnných v paměti. Rychlosť provádění příkazu je potom mnohem více určována jeho délkom než konkrétními operacemi, které v něm vystupují.

Kilobajt je tisíc bajtů

Dosti rozšířený omyl, který vyplývá z vcelku logických analogií: kilogram = tisíc gramů, kilometr = tisíc metrů apod. Kilobajt je však 1024 bajtů (viz slovníček). Není to tím, že by lidé od počítačů chtěli být za každou cenu originální, ale z praktických důvodů: dekadické číslo 1024 je rovno 2^{10} , takže se ve dvojkové soustavě zapíše jako 1000000000. Dvojková soustava je základní počítačovou číselnou soustavou a jejím prostřednictvím je také počítačová paměť adresována; proto je kapacita paměťových čipů (viz slovníček)

vždy rovna nějaké mocnině dvou. V důsledku toho se zavedlo používání výrazu jeden kilobajt pro počet bajtů, který je nejbližší tisíci a je přitom mocnou dvou — a to je právě 1024. (Rozdíl proti obvyklé předponě „kilo“, označující 1000, je naznačen i tím, že „počítačové kilo“ se značí velkým písmenem K.)

Analogicky jeden kilobit znamená 1024 bitů (viz slovníček) a podobná situace je i v případě jednotky jeden MB (megabajt). Megabajt opět není milión bajtů, ale 1024 · 1024 bajtů (tj. sumárně 1 048 576 bajtů). Bohužel konvence v používání této jednotky ještě není zcela ustálena a někdy se používá jeden megabajt ve významu 1000 kilobajtů (tedy 1 024 000 bajtů).

Naštěstí důsledky těchto omylů nejsou nijak dramatické, protože malý rozdíl v odhadu paměťového prostoru většinou nehraje roli.

Na stomegabajtový disk se vejde 100 megabajtů dat

Omyl obsažený v názvu tohoto odstavce je sice pochopitelný, ale číselně výrazný — mnohem výraznější než v předchozím případě. Kapacita x -abajtového disku je totiž mnohem menší než x bajtů, a to hned z několika důvodů:

- Udávaná kapacita označuje zpravidla technické maximum při nejvhodnějším naformátování (viz slovníček). Ve skutečném provozu bývá disk naformátován jinak.
- Tato kapacita zahrnuje i pro uživatele nedostupné „služební“ oblasti, jako je adresář souborů uložených na disku, seznam volných oblastí, náhradní stopy používané místo vadných míst na disku, adresy jednotlivých stop (tzv. „home“ adresy), hlavičky bloků a záznamů apod.
- Disk se dělí na fyzické bloky pevné délky. Málodky se podaří zaplnit tento fyzický blok na sto procent, takže na konci bloku zbude obvykle nevyužitý prostor, do něhož se již další záznam nevešel.
- Jsou-li na disku uloženy soubory s nesekvenční organizací (např. indexověný — viz slovníček), zaplňují se takové soubory úmyslně pouze zčásti, aby se usnadnilo a zefektivnilo pozdější vkládání dalších záznamů.

Většina uvedených bodů platí v určitém smyslu pro všechny typy disků; od disketu přes disky typu Winchester až po klasické velkokapacitní disky používané u střediskových počítačů. Můžeme být zpravidla rádi, činí-li u stomegabajtového disku kapacita přístupná uživateli v souhrnu 60 až 80 MB.

K vyměně obsahů dvou proměnných potřebujeme třetí proměnnou

Vyměna obsahu dvou proměnných se většinou skutečně programuje s využitím pomocné proměnné, jak ukazuje následující příklad, který vymění obsah proměnných A a B:

```
10 LET P=A
20 LET A=B
30 LET B=P
```

Ačkoli tento způsob výměny obsahu dvou proměnných je nejpoužívanější, není jediný možný a nemusí být ani nejrychlejší. Navíc vyžaduje další místo v paměti na pomocnou proměnnou P (v uvedeném případě to je sice zanedbatelné, ale pokud bychom vyměňovali obsah dvou textů o délce několika kilobajtů, může to být již citelně znát). A zdálo by se, že bez pomocné proměnné se neobejdeme — vždyť obsah dvou nádob naplněných různými kapalinami také nemůžeme vyměnit bez přelití obsahu jedné z nich do pomocné nádoby.

A přece to jde. S jistými omezeními se nám to podaří třemi příkazy v Basiku:

```
10 LET A=A+B
20 LET B=A-B
30 LET A=A-B
```

Po provedení těchto příkazů jsou skutečně obsahy proměnných A a B navzájem vyměněny, jak vyplývá z následující tabulky:

	Hodnota A	Hodnota B
Původně:	a	b
Po provedení příkazu 10:	$a + b$	b
Po provedení příkazu 20:	$a + b$	$a + b - b = a$
Po provedení příkazu 30:	$a + b - a = b$	a

Uvedené řešení není zcela ideální; má hned tři „vady na kráse“:

- dá se použít jen pro výměnu číselných dat;
- zaokrouhlovací chyby mohou způsobit, že $a + b - a$ není přesně rovno b a že $a + b - b$ není přesně rovno a ;
- vyměňujeme-li velmi velká čísla, může při jejich sčítání a odečítání dojít k přetečení (viz slovníček).

Základní myšlenka zmíněného triku, totiž rekonstrukce původní hodnoty proměnné z kombinace obou proměnných, však může být použita i jinak. Místo sčítání a odečítání použijeme instrukci XOR (tzv. logická nonekvivalence): výsledek operace A XOR B má jedničky v těch bitech, kde se A a B liší a nuly v bitech, kde se A a B shodují. Nahrádime-li ve výše uvedených příkazech operandy + i - operací XOR, obsah obou proměnných se rovněž vymění — a to bez nežádoucích vedlejších efektů; výsledek je přesný, při žádných hodnotách A a B nemůže dojít k havárii, na většině počítačů lze výměnu provést pouhými 3 strojovými instrukcemi, a navíc operaci XOR je možno aplikovat jak na číselné, tak na textové proměnné. I tato finta má však drobný nedostatek — lze ji použít jen v těch jazycích, kde je k dispozici instrukce XOR (na mnoha počítačích je proto dostupná pouze programátorům pracujícím v assembleru).

11. INTERMEZZO O POČÍTAČÍCH A ZLOČINU

Soudím, že pánbůh, tvoře člověka,
přecenil své možnosti.

Oscar Wilde



Pod pojmem zločin si většina z nás představí morbidní událost spojenou s krví a násilím; řekneme-li zločinec, vybaví se nám nejspíš odporně vyhlížející individuum se sveřepým výrazem a zvrhlými pudy. Avšak časy se mění a počítače vnesly novou kvalitu i do oblasti zločinu. Klasický zločinec z 19. století by asi nevěřil svým očím, kdyby mu někdo představil jeho novodobého počítačového kolegu. McNeil ve svém bestselleru *Poradce* (viz [MCNE83])

vykreslil počítačového zločince, kterého bychom mohli charakterizovat asi takto:

- Jedná se o sympatického mladého muže, okouzlujícího nikoli svaly nebo šarmem, ale svým neprekonatelným intelektem.
- Je tak dobrým programátorem, že obyčejné programování pro něj není. Dělá tedy počítačového poradce. Je však tak dobrým poradcem, že ani to pro něj není to pravé. Proto se zabývá počítačovým zločinem.
- Vraždy se dopouští jen velmi nerad a z vážných důvodů (např. pokud se mu někdo nevhodně plete do programování). Když už se vraždě nelze vyhnout, naprogramuje ji na počítači tak, aby byla decentní, bez rizika a aby se na to nemusel koukat.
- Cílem jeho snažení jsou drobné úpravy v komplikovaném systému programů řídících chod velké banky. Teoreticky takové úpravy nelze provést — to je důvod, proč je uskuteční prakticky. Okolnost, že se v důsledku toho stane milionářem, je vedlejší.

Tak tedy vypadá počítačový Fantomas. Jak vypadá realita?

Přiležitosti k počítačovému zločinu vznikají obzvláště tam, kde jsou rozšířeny počítačové a terminálové sítě a kde počítače poskytují účinnou podporu řízení průmyslu a finančnictví. Zejména využití počítačů ve velkých bankách je pro počítačové muže mimo zákon velmi atraktivní. Počítače přinášejí banky miliardové zisky — ale mohou také přinést miliónové zisky chytrému programátorovi. Právě taková úvaha však může v poctivém programátorovi vzbudit pocit Robina Hooda, který bohatým bere a chudým (tj. sobě) dává. I přesto, že počítačové zločiny patří mezi programátorské poklesky, jakých se jistě žádný ze čtenářů této knížky nedopustí, stojí za to všimnout si jich z čisté odborné stránky. Podívejme se na několik typických příkladů.

Poměrně jednoduchou myšlenku realizoval v období počátků počítačového zločinu jeden z programátorů velké banky. Program pro zpracování a kontrolu bankovních kont upravil tak, aby nevidoval přečerpání jeho vlastního konta. Vtip spočíval v tom, že tuto „úpravu“ provedl pouze v přeloženém tvaru programu — zdrojový tvar, který je čitelný a podlehá přísné kontrole, zůstal nepozměněn. Na podvod se příšlo jako obvykle náhodou. Když se pro poruchu počítače provádělo zpracování kont ručně a došlo i na programátorovo konto, zjistilo se, že ho dotyčný přečerpal již o více než 100 tisíc dolarů.

Uvedený případ byl jednou z prvních počítačových loupeží, podobně jako akce jiného nápaditého programáторa, který upravil příslušný program tak, že uměle zvyšoval některé (pečlivě vybrané) fakturované částky; rozdíl šel pochopitelně na programátorovo konto. Na podvod se příšlo, až když autor programu, ukolébaný pocitem bezpečnosti, pozměnil program tak, aby zvyšoval všechny fakturované částky. Dnes by podobné kousky prošly stěží (firmy vynakládají značné prostředky na boj proti počítačovému zločinu) a navíc by se na jejich autory pohlíželo jako na amatéry.

Další slavný počítačový podvod vycházel z jednoduché a vtipné myšlenky. Při počítání bankovních úroků se vypočítané obnosy musí vyjádřit v měno-

vých jednotkách; to určuje rozsah desetinných míst čísla. Např. 154,5259 dolaru musíme vyjádřit v dolarech a centech, tj. jako 154 dolarů 53 centů nebo eventuálně 154 dolarů 52 centů. V prvním případě je rozdíl od přesné hodnoty + 0,41 centu, v druhém případě - 0,59 centu. Takový rozdíl je samozřejmě zanedbatelně malý a banku naprostě neatraktivní. Pokud však budeme systematicky příslušné malé rozdíly střádat a připisovat na své konto (ovšem pouze ty, kde zlomky centů přebývají), dostaneme po miliónu transakcí — a takový počet není pro velkou banku nijaký mimořádný — docela slušnou sumu.

Rafinovaný trik, který údajně dokonce způsobil i jisté problémy britské justici (podle jakých zákonů soudit obviněného?), byl založen na následujícím postřehu: Poskytne-li banka úvěr některé firmě, sjedná s ní zpravidla výši úroku, den splatnosti úvěru a postih za nedodržení tohoto termínu. Mnohé firmy z obavy před tímto postřehem zaplatí poslední částku o něco dřív. Vlastně se tak dobrovolně připravují o určitou (z jejich hlediska samozřejmě velmi malou) částku, kterou představují úroky za tento krátký časový úsek. Tato částka nepatří bance a firma se o ni nestará — shrábne ji tedy pohotový programátor prostřednictvím pečlivě zamaskované úpravy v programu. Protože podobné bankovní operace se provádějí velmi často, částka na kontě utěšeně narůstá.

Malé české počítačové zločince zatím stihá smutný osud outsiderů. Ne že bychom u nás podobné triky nebyli schopni vymyslet — naopak, nápaditost našich lidí v podobných oblastech se stává proslulou. Ale i solidní počítačový zločin vyžaduje kvalitní hardware a software, a jak známo, devizové prostředky jsou omezené... Jak také loupit pomocí počítače, když každou instrukci programu ještě pětkrát přepočítá tým účetních, speciálně zřízený za tímto účelem. Zkuste rafinovaně využívat počítačové a terminálové sítě, ve kterých často nefungují ani základní funkce...

Přesto naši lidé neztrácejí smysl pro humor a nápady. Jak prohlásil neznámý optimista v restauračním zařízení 3. cenové skupiny:

„Elektroniku neelektronika, zlatý český ruce jsou holt zlatý český ruce!“

I když s tímto výrokem nemůžeme zcela souhlasit, neboť ani ty nejzlatější české ruce nezvládnou provést na logaritmickém pravítku deset milionů početních operací za vteřinu, kolují zprávy o tom, jak si naši lidé dokážou poradit i v mnohdy skromných podmínkách:

Obratným způsobem údajně využila skupinka mazaných pracovníků jednoho podniku specificky československé situace: Papír pro počítačové tiskárny ve formě známých skládaček je na okrajích proděrován. Toto děrování však bývá nekvalitní. Některá vyseknutá kolečka zůstávají v otvorech, uvolňují se během tisku a usazují se ve vnitřním prostoru tiskárny. Stává se někdy, že tiskárna některý znak vytiskne místo na papír na uvolněné kolečko, to odpadne — a příslušný znak potom ve výsledcích schází. Zmíněné pracovníky napadlo, že by se rozmištění vyněchaných číslic nemělo ponechat náhodě. Lehce přilepovali kolečka do připravených pozic na výplatních listinách a „omylem“ nevyplacené částky inkasovali společně se zasvěcenými účetními. Inu, zlaté české ručičky...

Lákadlem pro ty, kteří chtějí své programátorské schopnosti zpěněžit mimo meze zákona ovšem nejsou jen počítače, jejichž procesory zpracovávají vysoké finanční částky. Loupit je možno nejenom peníze, ale i data uložená v databankách anebo strojový čas počítače. Dochází ke špiónáži, vydírání, diverzím... (A to nejen v počítačově nejvyspělejších zemích — i u nás se našel operátor, který svou nespokojenosť v zaměstnání projevoval tím, že přejízděl magnetem po magnetických páskách. Způsobená mnohamiliónová škoda byla oceněna více než deseti lety pobytu za mřížemi.)

Počítačoví zločinci jsou nápadití a oblasti, ve kterých se „uplatňují“, jsou velmi rozmanité. Všimněme si činnosti, která je aktuální i u nás: softwarového pirátství.

Mnohé softwarové firmy patří v současné době k nejlépe prosperujícím na světě. Ceny velkých programových systémů jsou značné a není se tedy co divit, když dochází k pokusům obejít výlohy za jejich nákup tím, že se získají neoficiálně, což ovšem obvykle znamená také nelegálně. A takto získané programy se často (zvláště u nás) dále vyměňují nebo volně předávají. To leckdy vede k tomu, že se za autora programu vydává někdo jiný, kdo ve skutečnosti program pouze převzal či dokonce ukradl. Pozoruhodný případ se stal jednomu programátorovi, kterému byl nabídnut ke koupi jeho vlastní program(!).

Zisk a často i další existence softwarových firem však záleží na tom, kolika zájemcům se podaří pracně vytvořené produkty prodat. Proto se pochopitelně snaží proti loupežím a nelegálnímu předávání programů bránit.

Možnosti pro takovou obranu je více. Některé firmy využívají toho, že různé typy osobních počítačů „kompatibilních“ s IBM PC jsou ve skutečnosti kompatibilní jen do určité míry. Program má různé možnosti, jak může ověřit, zda je zpracováván na originálním IBM PC, na kompatibilním PC firmy Olivetti apod. Může například ověřit počet připojitelných přídavných zařízení nebo způsob naprogramování určitých částí operačního systému, které jsou závislé na hardwaru. U některých typů počítačů je do řídicích mikroprogramů zakódováno výrobní číslo procesoru, na které lze přímo vázat použitelnost programu.

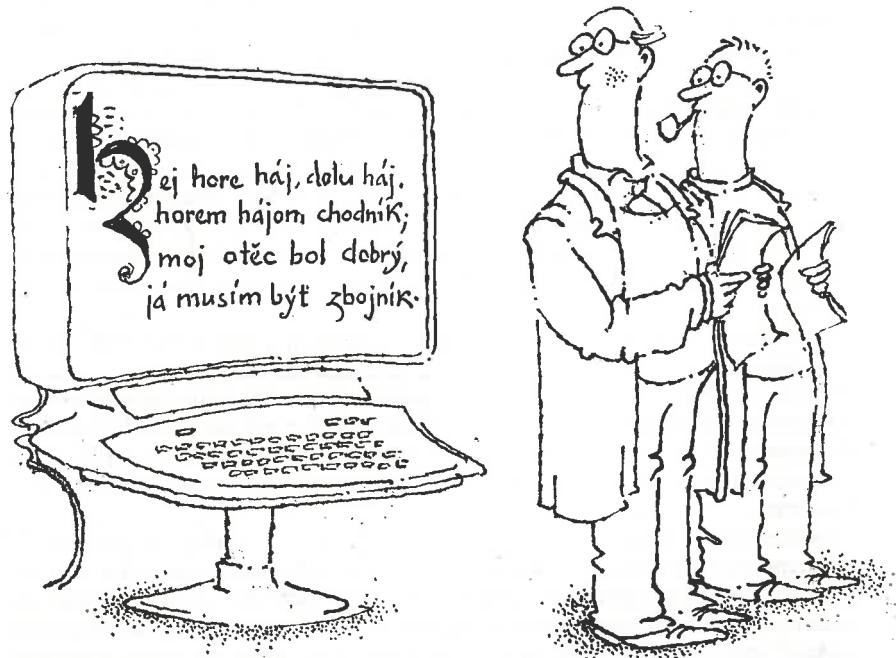
Donedávna se velká většina programů o to, kdo a kde je používá, prostě nestala, a jejich neoprávněným získáním riskoval uživatel pouze výčitky svědomí a případný (velmi nepravděpodobný) soudní spor. V současnosti však přibývají programů, které se jen tak nelegálně používat nenechají. Pokud program zjistí, že je provozován na jiném počítači, než pro který byl vyvinut či prodán, může na to reagovat různě. Některé programy odmítou prostě pracovat.

Existují však i záladné programy, které po takovém zjištění sice pracují, ale nepozorovaně se pomstí — třeba tím, že se v náhodném okamžiku vymaže operační paměť, přepíše část systémového disku nebo že se všechny diskové soubory prodlouží na dvojnásobnou délku. Některé z takových programů „nakazí“ touto svou vlastností všechny programy, které na disku naleznou, a vypukne epidemie. Jakmile se nakažený program okopíruje na jiný disk, infikuje při prvním spuštění opět všechny programy, které na disku najde. Ná-

kaza se tak rychle šíří a často nelze ani zjistit, který program byl jejím původcem. Programy tohoto typu se nazývají *virus* nebo *počítačový AIDS* a stávají se stále větší hrozbou pro softwarové piráty.

Způsoby, jak zneužít počítače, jsou rozmanité a nepřeberné. Naštěstí se daří počítače používat i opačně — v boji proti zločinům, a to nejenom počítačovým. Prakticky ve všech státech světa jsou využívány v evidenci zločinů a zločinců a při jejich vyhledávání. Počítače zpracovávají otisky prstů, katalogy přezdívek, typické pracovní postupy gangů i „vlků samotářů“ apod. Proti vlastnímu zneužití se brání důmyslnou technickou, právní a softwarovou ochranou (viz [BENE79]). Dá se proto očekávat, že počítačovým zločinům udělají nakonec přítrž samy počítače, samozřejmě za vydatné pomoci velké většiny programátorů. Vždyť přece existuje spousta zajímavějších programátorských problémů, než jak ukrást bance milión!

(Jestliže vás nyní napadlo: „samozřejmě, že existuje spousta zajímavějších problémů, než jak ukrást bance milión — například jak ukrást bance dva miliony . . .“, potom dejte na naši radu a nehledejte místo programátora ve finančnictví. Pomněte, že vězeňské cely nejsou vybaveny počítači!)



12. CHYBAMI SE ČLOVĚK UČÍ

Je to horší než zločin, je to chyba.

Policejní ministr Fouché při popravě věvody z Enghienu

V době, kdy nebyly počítače, panovala na světě téměř idylka: prakticky kdokoli si mohl připadat jako ideální, bezchybný jedinec. Zajisté docházelo k politovánihoným a nepředpokládaným situacím, ale za ty mohli vždy manželé nebo manželky, nepřátelé nebo neuznali přátelé, nešťastná doba, objektivní potíže, osud a nezřídka i počasí. Sem tam sice někdo přiznal vlastní chybu, ale pak to vypadalo spíš jako jistý druh exhibicionismu. Tento stav měl jediný nedostatek — nebyly počítače, a tak si lidé museli hledat různé náhrady (víno, ženy, zpěv, šachy atd.).

Dnes nám sice počítače poskytují skvělé rozptýlení, ale zároveň nám nekompromisně připomínají, jak jsme nedokonalí. Začátečníci sice někdy ze sestračnosti obviní z chyby počítač, ale to je obvykle rychle přejde. Později se někteří pokusí přestat dělat chyby — a to se jim nepodaří. Musíme žít s chybami, a to se dá snést jen tehdy, když tuto skutečnost uznáme a když se pokusíme ve vlastních chybách trochu vyznat. Zde se zmíníme o několika typických druzích chyb, které jsou dosti rozšířené a o nichž dosud nebyla blíže řeč.

Syntaktické, sémantické a zavlečené chyby

Chybovat je lidské, ale když se guma opotřebuje dřív než tužka, přeháníte to.

J. Jenkins

Chybovat je lidské, ale člověk se přitom cítí báječně.

Mae Westová

Motta k tomuto odstavci ukazují, že pod chybováním si mnozí představují činnosti zcela rozdílné a že není chyba jako chyba. Chyby, které vznikají při programování, se dělí na **syntaktické** a **sémantické**. Syntaktické chyby jsou prohřešky proti pravidlům zápisu programovacího jazyka a zpravidla je hlídá překladač. Například když vynecháme v klíčovém slově **PRINT** písmeno **I**, ohlási překladač během překladu programu, ve kterém se tento chybný příkaz vyskytne,

UNRECOGNIZED STATEMENT

(nebo něco podobného — tvar zprávy závisí na konkrétním překladači). V překladu do češtiny to znamená „příkaz nebyl poznán“. Všimněme si mimo- chodem, že překladač je (ve smyslu třetí kapitoly) typický sangvinik. Trpělivě znovu a znova hledá syntaktické chyby a ještě nás upozorní tak slušnou formou, jako kdyby si nebyl jist, zda je chyba na naší straně nebo na jeho. Překladač-cholerik by pravděpodobně reagoval:

NAUC SE KONECNE BASIC, NEBO JDI OD TOHO!

Syntaktické chyby vznikají ponejvíce překlepy a špatnou znalostí programovacího jazyka. Jejich odstranění naštěstí nepůsobí velké potíže — pokud jazyk dobře známe nebo máme k dispozici jeho popis, jedná se víceméně o rutinní záležitost. Musíme však dát pozor na *zavlečené chyby*, které se hlásí na jiných místech, než kde k nim ve skutečnosti došlo. Například u správného příkazu

100 GOTO 50

ohlásí většina překladačů chybu, jestliže jsme řádek

50 LET A=B

zapsali omylem

500 LET A=B

takže v programu žádný řádek 50 není.

(Pojem *zavlečená chyba* se používá též k označení chyb, které jsou do programu zaneseny dodatečně nějakou činností, obvykle úpravou programu nebo odstraňováním jiných chyb.)

Sémantické chyby neporušují pravidla zápisu, ale způsobují, že program provádí jinou činnost, než jakou by si programátor přál — často zcela nesmyslnou — nebo havaruje (např. při pokusech dělit nulou apod.). Odstraňování sémantických chyb bývá komplikované a plné omylů, které páchá postřížený programátor ve snaze o nápravu. K obtížné odhalitelnosti sémantických chyb přispívá i to, že se často jedně o chyby zavlečené.

Záměna podobných znaků

*Nejjednodušší chyby se nejhůře hledají.
Programátorský folklór*

Taková záměna je speciálním případem pravděpodobně nejčastější programátorské chyby — překlepů či přepsání. Zatímco „obyčejný“ překlep je zpravidla brzo odhalen, může záměna podobných znaků způsobit značně potíže, protože při kontrole programu snadno unikne pozornosti. Nejčastěji se plétou znaky:

0 (písmeno O)	a	Ø (číslice nula)
I (písmeno I)	a	1 (číslice jedna)
((levá závorka)	a	C (písmeno C)
. (tečka)	a	: (čárka)
" (uvodovky)	a	' (apostrof)

V případě, kdy narazíme na „záhadné“ chyby, je dobré mít možnost podobného omylu na paměti a podezřelé znaky v programu pečlivě zkonto rovat. Jak jsme viděli v odstavci *Nejdražší programátorská chyba na světě* v kap. 9, může se podcenění tohoto typu chyb zle vymstít.

Vynechání operátoru v aritmetických výrazech

To, jakou „paseku“ chyba způsobí, nezávisí na množství intelektu vynaloženého na její vytvoření.

Programátorský folklor

Aritmetické výrazy mají ve většině programovacích jazyků tvar podobný jako v běžném matematickém zápisu. Podobnost však není ani nemůže být úplná. V matematice například někdy značíme násobení tečkou, jindy ji zase vynecháváme — je to záležitost ustáleného úzu a občas i osobních zvyklostí. Přesto nedochází k žádným nejednoznačnostem, protože pro konstanty a proměnné užíváme v matematice většinou pouze samostatná písmena, případně s indexem či jinak graficky upravená. Součin proměnné x s proměnnou y můžeme tedy bez rizika nedorozumění zapsat xy . V případě programových proměnných X a Y tak ale postupovat nemůžeme, protože XY označuje proměnnou se jménem XY a nikoli součin dvou proměnných X a Y. V našem návuku na obvyklý způsob zápisu spočívá proto určité nebezpečí, které se obzvlášť týká právě možnosti vynechání operátoru pro násobení. Zapíšeme-li místo

10 LET X=2*(A+B)

příkaz

10 LET X=2(A+B)

skončí to ještě dobře — dopustíme se syntaktické chyby, kterou ohlási překladač. Zapíšeme-li však místo

20 LET C=A*B

příkaz

20 LET C=AB

neohlási překladač obvykle nic. Většina basikovských překladačů totiž připouští dosť obecný tvar identifikátoru proměnné, takže se AB vezme jako proměnná. Bude se tedy předpokládat, že se do proměnné C má umístit obsah proměnné AB (pokud se proměnná AB nikde jinde v programu nevyskytuje, nebude její obsah definován — viz odstavec *Nepřiřazení hodnoty proměnné*). Odhalit takto vzniklou sémantickou chybu může být dosť obtížné. Překladač, který dovoluje pouze jména proměnných podle staré normy Basiku, by ovšem hlásil syntaktickou chybu.

Přestože operátor násobení se vynechává nejčastěji, dochází ke stejněmu opomenutí i u ostatních operátorů — pochopitelně s analogickými důsledky.

Chybné uzávorkování

se vyskytuje nejčastěji u aritmetických výrazů. Vyznat se ve složitějším aritmetickém výrazu může být dost obtížné; naštěstí máme k dispozici několik jednoduchých kritérií správnosti uzávorkování.

Nejjednodušší kontrolou je ověřit, zda počet závorek ve výrazu je sudý. Tento prostý postup odhalí v praxi většinu chyb v uzávorkování, selže však, vynecháme-li sudý počet závorek — například:

$$(A+B)*(C+D)$$

Další jednoduchá možnost spočívá v tom, že prověříme rovnost počtu levých a pravých závorek. Tím zachytíme všechny případy, kdy počet závorek není sudý, a navíc mnohé další (včetně výše uvedeného nesprávně uzávorkovaného výrazu). Avšak i pak může být výraz nesprávně uzávorkován, například:

$$(A+B))/((C+D))$$

Velmi účinný prostředek pro otestování správného uzávorkování spočívá v tomto postupu: Procházíme výraz zleva doprava, přičemž za levou závorku přičítáme (počínaje od nuly na začátku) jedničku a za pravou závorku jedničku odečítáme. Jestliže jsme při tom nikdy nedostali záporné číslo a na konci obdrželi nulu, je výraz pravděpodobně správně uzávorkován — výjimku tvoří některé málo pravděpodobné případy jako např. A(+).B.

Třebaže nesprávné uzávorkování nepředstavuje pro program velké nebezpečí, protože se téměř vždy jedná o syntaktickou chybu, vyskytuje se natolik často, že se pečlivá kontrola uzávorkovaných výrazů vyplatí.

Nepřiřazení hodnoty proměnné

je sémantická chyba, která často vzniká nesprávným zápisem některé proměnné, vynecháním aritmetického operátoru nebo vynecháním některého příkazu. Například v programu

20 LET Z=X**2

30 PRINT Z

40 END

mohlo být nepřiřazení hodnoty proměnné Xzpůsobeno tím, že jsme vynechali příkaz

10 INPUT X

K takovým situacím dochází nepozorností při zápisu nebo omylem — např. smazáním řádku při editování programu. Chyba tohoto druhu je velmi nebezpečná, neboť proměnná s nepřiřazenou hodnotou obvykle nabývá náhodné hodnoty. Program potom může jednou pracovat (náhodně) správně a jindy chybně, nebo havarovat. Některé překladače umožňují přeložit program tak, že použití proměnné s nepřiřazenou hodnotou je zjištěno a ohlášeno jako chyba. Ve fázi ladění programu je výhodné takových kontrol používat, avšak po odladění programu je lépe zdrojový tvar znova přeložit bez kontrol, které většinou značně zpomalují výpočet.

Nekonečný cyklus

je sémantická chyba, která je často chybou zavlečenou. Říká se tak situaci, kdy se při výpočtu některé části programu do nekonečna opakuje, například, když se skáče z místa A na místo B a z místa B zase na místo A (viz heslo *Nekonečný cyklus* ve Slovníku). Program lze potom ukončit pouze vnějším zásahem. Mezitím ovšem může zbytečně potisknout spoustu papíru nebo vytvořit obrovské, nesmyslné soubory apod.

Následující program má přečíst přirozené číslo n a vypočítat jeho faktoriál, tj. číslo $1 \cdot 2 \cdot 3 \dots (n-1) \cdot n$.

```
10 REM VYPOSET FAKTORIALU
20 INPUT N
30 LET F=N
40 LET N=N-1
50 IF I=0 THEN 80
60 LET F=F*N
70 GOTO 40
80 PRINT "FAKTORIAL ZADANEHO CISLA JE ", F
90 END
```

Program obsahuje sémantickou chybu, která může vést k nekonečnému cyklu. Na řádku 50 je totiž ve výraze $I=0$ místo nuly omylem písmeno 0. Důsledkem je, že pracujeme s proměnnou, které není v programu přiřazena hodnota. Pokud nás překladač nehlídá použití proměnné s nepřiřazenou hodnotou a na začátku výpočtu bude v proměnné 0 náhodou kladné číslo větší než zadaná hodnota n , dojde k nekonečnému cyklu. (Mimořádem, je poučné zamyslet se nad tím, jak bude program reagovat na jiné náhodné hodnoty proměnné 0.)

Jestliže tuto chybu odstraníme, bude program fungovat správně, pokud vstupními daty budou přirozená čísla 1, 2, 3, ... To se sice skutečně předpokládá, ale omylem nebo nedozorem se může stát, že číslo n bude zadáno jako záporné, nulové nebo necelé. V tom případě opět dojde k nekonečnému cyklu (jak lze snadno ověřit). I tento případ lze považovat za sémantickou chybu, protože smysluplnost vstupních dat se má v programu hlídat (správný program má být „hroch“ ve smyslu kapitoly *Malé panoptikum programů*).

Špatně zapsané klíčové slovo příkazu

je obvykle důsledkem obyčejného překlepu. Většina z nás má přirozenou tendenci klíčová slova příliš nekontrolovat — přeletíme je očima a leckdy nám ujde, že některé písmeno vypadlo, nebo je nesprávné. Když se tato chyba navíc zkombinuje se záměnou podobných znaků, dochází k situacím, kdy lokální slepota postihuje i velmi zkušeného programátora. Ten pak nevěří vlastním očím, když překladač ohláší chybu v „evidentně správném“ příkaze

100 GOT0 20

(než si jeho začínající kolega všimne, že druhé „0“ ve slově GOT0 není „0“, ale nula).

Test na rovnost

Všechna stejná čísla v počítači jsou si rovna. Ale některá jsou si rovnější.
Parafráze na klasické téma

je příkazem, který má v důsledku nepřesného zobrazení čísel v počítači a zaokrouhlování často charakter sémantické chyby. Taková chyba je o to zákeřnější, že její následky závisí na použitém programovacím jazyce, překladači i na hardwaru počítače. Níže uvedený program má rozhodnout, zda zadané vektory $x = (a,b)$ a $y = (c,d)$ jsou na sebe kolmé (to je pravda, jestliže $ac + bd = 0$):

```
10 REM PROGRAM ZJISTUJE KOLMОСТ DVOU VEKTORU
20 INPUT A, B, C, D
30 IF A*C+B*D=0 THEN 60
40 PRINT "VEKTORY NEJSOU KOLME"
50 STOP
60 PRINT "VEKTORY JSOU KOLME"
70 END
```

V implementaci (viz slovníček) Basiku, na které jsme tento programek testovali (počítač ICL 2950/10 s operačním systémem VME), se vypsal

VEKTORY NEJSOU KOLME
pro vektory zadané hodnotami $a = 1, b = -2, c = 0,2$ a $d = 0,1$. Ve skutečnosti tyto vektory kolmé jsou, jak snadno spočítáme — chybu má na svědomí nevhodně použitý test na rovnost a zaokrouhlovací chyby, které vznikají při provádění aritmetických operací. Znovu připomínáme, že takovéto chyby jsou implementačně závislé a že náš programek může pro stejně hodnoty na jiném počítači nebo pod jiným překladačem docela dobře vypsat správnou odpověď (např. na mikropočítači Sinclair ZX 81).

Víme-li, v jakém rozmezí hodnot se veličiny v testovaném výraze pohybují, je oprava většinou snadná — místo rovnosti hodnot testujeme, zda je absolutní hodnota jejich rozdílu dostatečně malá; řádek 30 změníme např. takto:

```
30 IF ABS(A*C+B*D)<0.0001 THEN 60
```

Pokud jsou hodnoty všech proměnných celá čísla, většina překladačů Basiku s nimi bude pracovat přesně a k podobným problémům nedojde. To však vždy zaručit nelze.

Použití necelého kroku v příkaze cyklu

může rovněž vést k sémantickým chybám, které jsou implementačně závislé (a tím zrádnější). Ukažme si na příkladě, o co se jedná. Od cyklu

```
10 FOR X=0 TO 1 STEP 0.1
20 PRINT X
30 NEXT X
```

budeme asi očekávat, že vypíše čísla 0, 0.1 atd. až 0.9 a 1. Cyklus

```
10 FOR X=1 T0 0 STEP -0.1
```

```
20 PRINT X
```

```
30 NEXT X
```

by měl analogicky vypsat stejná čísla, ale v obráceném pořadí.

Ve skutečnosti bude výsledek záležet na implementaci; např. na počítači ICL 2950/10 s operačním systémem VME proběhne první cyklus podle našich představ, kdežto druhý nevypíše hodnotu 0 (tj. v prvním případě se provede příkaz PRINT jedenáctkrát, kdežto v druhém jen desetkrát). Na mikropočítači Sinclair ZX 81 to dopadne přesně naopak.

Tento druh sémantické chyby těsně souvisí s chybou popisovanou v předchozím odstavci. Test na konec cyklu se provádí porovnáním řídící proměnné cyklu (v našem případě to je proměnná X) s její mezní hodnotou zadanou v příkaze FOR za slovem TO (v našem případě s jedničkou pro první případ a s nulou pro druhý případ). Zaokrouhlovací chyby však mohou způsobit určitou odchytku od teoretičky správné hodnoty řídící proměnné, a tím i výsledky, které neodpovídají našim představám.

Nerespektování změny obsahu proměnné

je rovněž záludná sémantická chyba. Příklad, který si ukážeme, pochází původně dokonce z profesionálního programového systému (!).

Následující fragment programu má počítat approximovanou hodnotu derivace funkce FNF, definované na jiném místě programu. (Znalost pojmu derivace zde nehraje roli.)

```
50 LET Y1=FNF(X)
60 LET X=X+0.001*X
70 LET Y2=FNF(X)
80 LET X=X-0.001*X
90 LET D=(Y2-Y1)/(0.001*X)
```

Programátor se domníval, že provedením těchto příkazů se hodnota X nezmění (předpokládá se, že se hodnota X nezmění vyuvoláním funkce FNF).

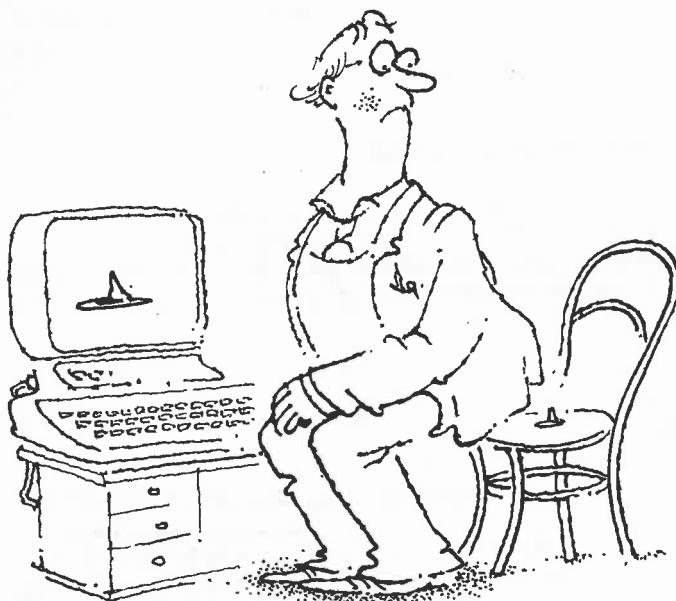
To však není pravda. Jestliže nenulové číslo zvětšíme o jedno promile a to to zvětšené číslo zase o jedno promile zmenšíme, nedostaneme původní číslo. O tom se můžeme snadno přesvědčit. Rozdíl je pochopitelně relativně malý, ale jestliže výpočet mnohokrát zopakujeme, může se hodnota proměnné X cítelně změnit. Pak se třeba stane, že program bude počítat „divně“, ale nebude jasné, zda je to chyba algoritmu, programu, důsledek numerické nepřesnosti, popř. zda jde vůbec o chybu.

Tím způsobem vznikne lahůdková sémantická chyba, která vyžaduje zkušeného programátora silné povahy a svaté trpělivosti. V našem případě se chyba trochu prozrazuje tím, že pro $X = 0$ (což je jediný případ, kdy se hodnota X opravdu nezmění), dojde pro změnu k dělení nulou. (Tento způsob na programování numerického výpočtu derivace je poněkud rozmarný.)

13. JEHLA V KUPCE SENA

Zdá-li se vám, že všechno funguje,
určitě jste něco přehlédli.
Z tzv. Murphyho zákonů

Čím promyšleněji lidé jednají, tím
účinněji je může postihnout náhoda.
Friedrich Dürrenmat



V předcházející kapitole jsme si ukázali, s jakými typy chyb se v programech nejčastěji setkáváme. Teď bychom potřebovali vědět:
a) Jak zjistit, zda v programu jsou chyby.
b) Jak objevit místa, kde se tyto chyby skrývají.
c) Jak chyby odstranit.
Cinnosti a) říkáme *testování*, cinnosti b) a c) označujeme souhrnně jako *la-*

dění. Poznamenejme, že někteří autoři používají termín „testování“ v širším smyslu a zahrnují pod něj všechny tři výše uvedené činnosti; v jejich interpretaci tedy testování v sobě zahrnuje i ladění. I v našem pojetí jsou testování a ladění činnosti navzájem úzce svázané. Pokaždé, když odhalíme v programu chybu (o což usilujeme při testování), musíme ji lokalizovat a odstranit (což je již ladění), takže obě fáze se časově prolínají.

Ladění a testování patří k nejnáročnějším fázím programátorské práce. Je užitečné myslit na to už při psaní programu a vytvářet ho tak, aby se v něm braly v úvahu všechny eventuality a abychom si vytvořili podmínky pro snadné ladění a testování. Tomuto přístupu říkáme *defenzivní programování*. Pokud v programu nepočítáme s tím, že počet měření může být omylem zadán jako nula, že ve neodladěném programu může vzdálenost dvou bodů vycházet záporná, nebo že určitý výsledek může být za jistých okolností neočekávaně větší než tisíc, zjednodušíme si sice poněkud psaní programu, ale ztratíme podstatně víc času a nervů při jeho ladění. Mnohem lépe se nám program bude ladit, když budeme tyto samozřejmě předpoklady v programu kontrolovat a v případě, že nejsou splněny, vytiskneme varování.

V programu jsou chyby — začínáme ladit

Axiomatická teorie programování:
Definice 1: Program je konečná posloupnost příkazů.

Axióm 1: V každém programu je alespoň jedna chyba.

Axióm 2: Každý program, který obsahuje více než jeden příkaz, je možno napsat alespoň o jeden příkaz úsporněji.

Věta 1: Každý program lze zkrátit na jeden příkaz, a ten je chybný.

Programátorský folklór

Hledání chyb v jakémkoli rozsáhlejším díle bývá právem přirovnáváno k hledání příslovečné jehly v kupce sena a programy rozhodně nejsou v tomto směru výjimkou. Pravda, mnohé chyby jsou natolik uznalé, že se přihláší samy, a spíše než jehlu v kupce sena připomínají šídlo v pytlí. Jsou to zvláště chyby syntaktické, které v naprosté většině případů odhalí překladač. Interpretaci překladače (viz slovníček) obvyklé u Basiku hlásí chyby často ihned po napsání chybného příkazu, jindy až při pokusu o jeho provedení. U kompliátorů (viz slovníček) se všechny ohláší obvykle společně po ukončení překladu — například tak, že se vypíše něco jako

COMPIILATION FAILED: 237 ERRORS

za čímž následuje 237 řádků specifikujících jednotlivé chyby. Začátečník pod-

lehne panice — jak mohl proboha udělat ve storádkovém programu tolik chyb? Zkušenější programátor ví, že v programu bude skutečně asi tak 10 chyb a ostatní jsou chyby zavlečené, a má skoro pravdu — až na to, že některé z domněle zavlečených chyb byly chyby skutečné. Ty se po opravě objeví réz dalším pokusu o překlad znova. Po jejich odstranění programátor čeká, že teď už překlad projde bez problémů, a jako vždy se myslí: při opravě jedné chyby se mu podařilo zplodit dvě jiné.

Dříve či později se však program přece jen přeloží bez chyb. Zdá se, že je vyhráno. Zkušenější programátor však ví, že teď to teprve začne být zajímavé; skončila etapa ladění syntaktických chyb (alespoň těch, které odhalil překladač) a ke slovu se začnou dostávat chyby sémantické.

Spustíme-li poprvé úspěšně přeložený program, můžeme se dočkat nejrůznějších překvapení:

1. Program havaruje (tj. operační systém vytiskne zprávu, že při výpočtu došlo k chybě).
2. Program stále počítá; zdá se, že bude počítat věčně.
3. Program neprovede nic (skončí, aniž by cokoli vytiskl), tj. chová se jako černá díra popsaná v kapitole *Malé panoptikum programů*.
4. Program produkuje výsledky, které jsou na první pohled (nebo po prověření) chybné.
5. Program skončí bez havárie a výsledky vypadají důvěryhodně. Tento případ je nejméně pravděpodobný a zkušeného programátora při něm zamrazí v zádech. Ví totiž, že programy bez chyb se vyskytují jen v legendách a bájích, takže ho čeká namáhavé hledání prohnaných a světem proštělých chyb, kterým se podařilo dobře se ukrýt. (Hledáním takových chyb se budeme zabývat v odstavci *Testujeme program*.)

V případech 1 až 4 je přítomnost sémantických chyb (nebo syntaktických chyb, které překladač neodhalil) zřejmá a nám zbývá „pouze“ odhalit, kde jsme je udělali, a opravit je. A to je okamžik, kdy si vzpomeneme na onu pověstnou jehlu v kupce sena.

Sherlock Holmes na stopě aneb ladění pokračuje

Umí-li někdo uvažovat vskutku dokonale (...) a předloží-li se mu jediný fakt se všemi charakteristickými rysy, dokáže z něho vyvodit nejenom řetěz událostí, který k němu vedl, ale i výsledky, jaké z něj budou vyplývat.
A. Conan Doyle

Ladění programu má mnoho podobného s detektivním pátráním:
— Víme, že byl spáchán „zločin“ (mnohý programátor alespoň podvědomě

považuje chybnou funkci programu za něco podobného zločinu a ponechává stranou skutečnost, že viníkem je obvykle on sám).

- Místo po pachateli pátráme sice „jen“ po chybě, ale i ta se také umí až nepríjemně dobře skrývat.
- Podobně jako pachatel i chyba po sobě zanechává stopy, které dobrému detektivovi umožní, aby ji nakonec odhalil.

Z jakých stop můžeme při pátrání po chybě vycházet? Pokud program havaruje, je nejdůležitějším zdrojem informací samo hlášení o chybě. To může mít nejrůznější podoby v závislosti na použitém programovacím jazyku, překladači a operačním systému. Ideální jsou systémy, které ohláší něco jako:

Dělení nulou v 25. řádku podprogramu ALFA.

Podprogram byl vyvolán ze 74. řádku hlavního programu s parametry N=5, X=0.

Ostatní proměnné mají hodnoty A=1. 34815, I=0, K=-12.

(Ano, i s takovým komfortem se u některých systémů setkáváme — i když uvedené informace jsou zpravidla psány anglicky.) Pokud zjistíme, že se v řádku 25 počítá výraz $A^{**}2+1/X$, je nám situace jasná a zbývá zjistit, proč je X rovno nule, přestože má označovat nadmořskou výšku hory, tedy číslo svou podstatou kladné. To bývá obvykle jednoduché. Pokud neuspějeme (protože jsme postiženi lokální slepotou nebo proto, že jde o případ opravdu zapeklitý), použijeme některou z metod popsaných dále.

Většina systémů bohužel zdaleka není k uživateli tak přátelská a často na nás při stejně chybě počítá vyštěkne

Chyba 000B na adresě 12FA7

a vychrlí několik kilogramů papíru potištěného výpisem obsahu paměti počítače v šestnáctkové soustavě nebo zaplaví obrazovku úhlednými sloupcí čísel. V takovém případě máme tyto možnosti:

- Jít se oběsit. (Jedinci s tak malou psychickou odolností by se však do programování nikdy neměli pouštět.)
- Naučit se v té spoustě čísel vyznat. (Upřímnou soustrast.)
- Pořídit si počítací s lepším operačním systémem.
- Postupovat stejně jako v případě komplikované chyby, tj. jít s kanónem na vrabce.
- Přemýšlet a snažit se z toho mála, co o chybě víme, získat aspoň jakous takous informaci — třeba o druhu chyby nebo o přibližném místě jejího výskytu. I drobná stopa může vést k rychlému odhalení pachatele.

Podobně můžeme postupovat i tehdy, když program nehavaruje, ale dává chybné výsledky. Rozbor výsledků často představuje dostatečně zřetelnou stopu, na jejímž základě chybu najdeme.

Jsou-li informace získané předešlým způsobem nedostatečné nebo jestliže chyba nezanechala žádnou patrnou stopu, volíme další postupy převzaté rovněž z arzenálu detektivních románů:

- Metoda induktivní, tj. logický rozbor všech informací, které máme. Jinými slovy, hledáme odpověď na otázku „Co všechno lze vyvodit z toho, co program udělal?“

- Metoda deduktivní, tj. rozbor možných příčin chyby. Jinak řečeno, kladení otázky: „V jakých případech by program mohl dělat právě to, co dělá?“

S řešením kriminální zápletky „od stolu“, pouhým využitím „malých šedých buněk“, se setkáváme často u papírových hrdinů, jakými byli Sherlock Holmes, Hercule Poirot nebo Nero Wolf, ale v kriminalistické praxi se tento přímes uplatní jen stěží. Podobně uvedené metody logické analýzy mají samy o sobě úspěch jen výjimečně, avšak v kombinaci s dalšími metodami mohou být užitečné.

Někteří programátoři (a jsou mezi nimi bohužel někdy i praktici ošlehaní větrem počítačových klimatizací) volí cestu čirého zoufalství: „A nepomohlo by, kdybych to změnil tak a tak? Zkusíme to.“ Zastánce takového přístupu nezbývá než litovat. Možností, jak napsat program chybě je nesrovnatelně víc než možností, jak ho napsat správně, takže metoda náhodného hledání správného řešení je předem odsouzena k nezdaru.

Nejnadějnější a v praxi osvědčený postup je založen na zásadě připomínající přístup detektivů drsné školy: „Když stopy nevedou k pachateli nebo ho neusvědčují, nastavíme mu past. Když nevíme, jak pachatel postupuje, bude me ho sledovat.“ I my můžeme chybě nastavit past a průběh programu sledovat.



Sledování výpočtu

Velký detektiv promluvil. „Dejte mi mé domino! Ty stopy je třeba sledovat.“

*S. Leacock:
Literární poklesky*

Tato metoda ladění je založena na získávání zpráv o postupu výpočtu během jeho průběhu. Zprávy mohou být různého druhu a rozsahu — podle zvolené strategie.

Důležité informace získáme již tím, že do každé větve programu (i do všech jeho podprogramů, pokud nejsou předem odladěny) zařadíme vypsání zprávy o tom, že výpočet tudy prošel. To nám umožní lokalizovat místo, kde program zabloudil, nebo kde došlo k havárii. Navíc můžeme konfrontovat průběh výpočtu s našimi představami o tom, jak by tento výpočet měl pro zadaná data postupovat. Detailnější informace můžeme získat vypsáním hodnot zvolených proměnných na různých místech programu.

Uvedené výpisy jsou zpravidla dosti účinné; pokud je však musíme do programu vkládat ručně, je to pracné a jako každý zásah do programu to představuje jisté riziko zanesení chyby. Navíc je musíme po odladění opět pracně (a opět s jistým rizikem) z programu odstraňovat.

Tuto nevhodou odstraňují *sledovací programové prostředky* (trasovací programy, programy pro analýzu cesty výpočtu, interaktivní testovací prostředky, manipulační prostředky). Jedná se o softwarové „nástroje“, které provádějí automaticky nejen sledovací akce, o nichž jsme mluvili, ale celou řadu dalších; mohou například podat zprávu v okamžiku, kdy se změní hodnota sledované proměnné nebo začne platit určitá podmínka apod. Manipulační prostředky dokonce dovolují ovlivňovat kontrolní výpočet interaktivními zásahy, které mění stav sledovaného výpočtu (např. lze vynutit změnu hodnoty určité proměnné) a které rovněž mohou změnit sledovací režim.

Sledovací prostředky nezasahují do zdrojového textu, a proto jejich použitím neriskujeme zanesení dalších chyb. Sledovacímu programu je ovšem třeba detailně vysvětlit, co po něm vlastně chceme. Obvykle se za tím účelem musíme naučit určitý *sledovací jazyk*, který obsahuje vhodně kódované příkazy pro sledovací program. Význam téhoto příkazu může být např.:

- Vypiš proměnnou X pokaždé, když se změní její hodnota.
- Vypiš proměnnou X pokaždé, když je splněna určitá zadaná podmínka.
- Zastav výpočet na určeném řádku a vypiš vybrané proměnné.
- Zastav výpočet a vypiš hodnoty skutečných parametrů pokaždé, když je volán určitý podprogram, atd.

Sledování může být velmi efektivní, zvláště sáhneme-li po něm s rozmyslem a po náležité přípravě. Chtěli bychom však varovat před živelným a nepromyšleným používáním sledovacích prostředků; snadno se tak dostaneme do postavení, kdy — jak praví známé pořekadlo — vytáhneme kanón na

vrabce. Situace, které bychom vyřešili jednoduchou úvahou, nakonec zvládne v potu tváře po vysilujícím sledování, v němž jsou podrobně prozkoumány stovky možností, které s problémem vůbec nesouvisejí.

Jak ze slepé uličky

Téměř v každém detektivním románu od doby znamenitých povídek Conana Doyla přichází okamžik, kdy ten, kdo případ vyšetřuje, shromázdí všechna fakta potřebná k řešení alespoň nějakého stadia svého problému. Tato fakta se často zdají podivná a zdánlivě spolu nesouvisí. Avšak zkušený detektiv pozná, že v tom okamžiku není zapotřebí dalšího vyšetřování a že jenom čisté uvažování může vést k správnému seřazení nashromážděných faktů. Hraje tedy na housle nebo se pohodlně uvelebí v lenošce a pokuřuje dýmku, až najednou — hrome! už to má!

A. Einstein, L. Infeld: Fyzika jako dobrodružství poznání

I při použití výše popsaných metod se nemusí podařit chybu lokalizovat. V takovém případě mohou pomoci následující rady (volně upravené podle [MYER79]):

- Přemýšlejte. Důkladnou analýzou informací, které o chybě máte, se ji často podaří lokalizovat bez dalšího použití počítače.
- Nepovažujte nic za samozřejmé. Ověřte si, že výpis programu, nad kterým bádáte, opravdu přesně odpovídá programu, který havaroval. Přesvědčte se, že vstupní data byla zadána skutečně tak, jak předpokládáte. Uvědomte si, že výpis programu nemusí přesně odpovídat jeho zápisu v počítači (např. na některých tiskárnách označuje mezera či otazník i znak, který tiskárna neumí vytisknout; překladač potom při zpracování tohoto podivného znaku ohláší „nevysvětlitelnou“ chybu).
- Nevíte-li jak dál, bádání nad chybou přerušte (u jednoduchých programů po půlhodině, u složitých po několika hodinách). Zabývejte se něčím jiným a nechte pracovat podvědomí. K chybě se vraťte, až vás zčistajasna napadne, v čem to může být, nebo aspoň až si trochu odpočinete.
- Nevíte-li stále jak dál, vysvětlete problém někomu jinému. Ujasníte si myšlenky a často najdete řešení i bez jeho pomoci. (Mnohdy pomůže vyložit

situaci někomu, kdo programování nerozumí — pokud je ochoten naslouchat.)

- Případné další sledování provádějte s rozmyslem a až po důkladné analýze nashromážděných informací.
- Neexperimentujte. Jak bylo již řečeno, pravděpodobnost, že chybu odstraníte víceméně náhodně prováděnými úpravami programu, je mizivá a navíc tyto pokusy mohou zanést do programu další chyby.

Jak chybu opravit

Mohlo by se zdát, že když jsme již chybu odhalili a našli způsob, jak ji opravit, nečiní vlastní oprava žádné potíže — stačí program editorem (viz slovníček) pozměnit. Praxe však ukazuje, že je rozumné se na chvíli zastavit a vzít v úvahu následující rady (inspirované opět knihou [MYER79]):

- Prohlédněte si okolí místa, kde jste našli chybu. Je zvýšená pravděpodobnost, že tu budou její „bráškové“ či „sestřičky“. (Chyby stejně jako hříbky mají tendenci se shlukovat; objevíme-li je najednou, ušetříme čas.)
- Opravte chybu a nikoli její následek. Stává se velice často, že příčinu neidentifikujeme správně a místo, abychom opravili podstatu chyby, kompenzujeme pouze její důsledek. Chyba se potom pravděpodobně projeví později znovu, jiným způsobem.
- Nikdy nemáte stoprocentní jistotu, že jste chybu identifikovali správně. Pravděpodobnost je tím menší, čím je program rozsáhlější.
- Dejte si velký pozor, abyste opravou nezanesli do programu další chybu. Opravy vymýslíme obvykle narychlou, často v časové tísni a v době, kdy již pracujeme na něčem jiném; navíc se chyby objevují zejména v komplikovaných místech. Není proto divu, že při opravě uděláme chybu mnohem snáze než při běžném programování. Nová chyba vzniká velmi často tím, že naše oprava má nežádoucí vedlejší efekt.
- Po opravě bývá leckdy nutné vrátit se v práci na programu zpět (revidovat určitou část koncepce, změnit meze polí, přizpůsobit těmito změnám i další místa v programu apod.)
- Opravujte zásadně zdrojový tvar programu. Mnohé operační systémy dovolují provádět zásahy do cílových modulů. Dostatečně poučený programátor — zejména pokud programuje v assembleru — tak sice může ušetřit čas potřebný na nový překlad (s tím, že ve zdrojovém tvaru provede opravu později, až bude více času), avšak za cenu neúměrně velkého rizika. Na tuto opravu se v návalu další práce velmi snadno zapomene a hledání dalších chyb je potom mimořádně ztíženo.

Testujeme program

Kdy máte začít testovat? Nejlepší odpověď na tuto otázku je: Jakmile je co testovat!

G. Gibson, J. Young

Jak moc dobrý má být dobrý program?

Petr Koubský

Poté, co jsme potrápili překladač syntaktickými chybami a sami sebe sémantickými chybami, nastane zpravidla dříve či později slavnostní okamžik, kdy proběhne výpočet, neohlásí se žádná chyba a kupodivu dostaneme předpokládané výsledky. Až uvěříme vlastním očím a vzpamatujeme se z překvapení, podlehнемe asi na chvíli blaženému pocitu vítězství nad záludnostmi programování: program funguje!

Chyba lávky. Program sice fungoval, ale:

1. jen jednou — podruhé může havarovat (například proto, že některá proměnná, které jsme zapomněli přiřadit hodnotu, nabude tentokrát náhodnějinou hodnotu);
2. jen pro ta data, se kterými byl výpočet proveden. Pro jiná data může dát nesprávné výsledky, nebo se může ohlásit chyba.

Pokud tyto skutečnosti nevezmeme na vědomí, vystavujeme se riziku, že po kratší či delší době začne program při výpočtu dělat „psí kusy“. Čím později se tak stane, tím hůře pro nás, protože potom si už obvykle dobré nepamatujeme strukturu programu a hledání záludné chyby je pak o to obtížnější. Jak takové riziko odstranit nebo snížit na minimum?

Předně si musíme připomenout, že se nám vždy nemůže podařit rozhodnout, zda v programu je či není chyba — o tom jsme se zmínili již ve druhé kapitole (může se to podařit pouze u mimořádně jednoduchých programů). Půjde nám tedy spíše o to, abychom program v rozumném čase dostatečně prověřili — otestovali.

S testováním je spojen jeden důležitý psychologický moment: měli bychom ho provádět se záměrem odhalit v programu utajené chyby, a ne s úmyslem (byť jen podvědomým) radovat se z toho, že program už funguje. Testování proto považujeme za úspěšné, pokud odhalíme dosud nezjištěné chyby. Testování, které žádnou chybu neodhalilo, musíme považovat za neúspěšné (třebaže jistý význam mělo — ukázalo nám místa, kde chyby hledat nemáme). Známožná nezvykle a tvrdě, ale tvrzení, že v každém programu jsou chyby, je téměř vždy pravdivé (i když jsme ho v jednom mottu v této kapitole dovedli ad absurdum).

Nesmíme ovšem na druhé straně přestřelit a všechn další čas po napsání a odladění programu věnovat jeho testování — vždyť jsme ho psali proto, aby se v praxi používal. To, jak důkladně budeme program testovat, záleží

pochopitelně na jeho rozsahu a struktuře a především na tom, jak vysoké požadavky máme na jeho spolehlivost. V některých speciálních případech se můžeme pokusit správnost programu dokázat — verifikovat ho. Přes velké úsilí, které věnují odborníci v informatice vytvoření metod na verifikaci programů, zůstává však tento postup zatím spíše teorií a jeho praktické použití vzdáleností.

Problémy spojené s testováním programů jsou v centru zájmu počítačových odborníků; o testování je vypracována obsáhlá teorie a existuje řada testovacích metod — od jednoduchých až po velmi důmyslné (a většinou také patřičně komplikované). V této kapitole se zmíníme stručně jen o těch nejjednodušších.

Anomalie

Celá řada chyb v programech je důsledkem obyčejných překlepů. Přesto často není snadné je odhalit. Základní myšlenka metody detekce anomalií spočívá v postřehu, že při takovém přepsání (ale i v důsledku jiných chyb) dochází poměrně často nejen k chybě jako takové, ale i k anomalií — tj. k situaci, kdy část programu, ve které se chyba vyskytuje, bude sice syntakticky správně zapsaná, ale bude vypadat podezřele. Zapíšeme-li např. sekvenci příkazů

...
50 LET I=1

60 LET J=2

...

omylem jako

...
50 LET I=1

60 LET I=2

...

nedopustili jsme se žádné syntaktické chyby, avšak přiřazení dvou různých hodnot proměnné I bezprostředně za sebou je evidentně podezřelé (a pravděpodobně naznačuje sémantickou chybu). Podobnou anomalií je např. použití proměnné, které nebyla přiřazena žádná hodnota, existence nedosažitelných příkazů (tj. příkazů, jež nemohou být pro žádná vstupní data nikdy provedeny), deklarování nepoužitých proměnných (protože proměnné se v Basiku bohužel nedeklarují, platí to v něm jen o polích, ve většině jiných programovacích jazyků však i o proměnných) apod.

Metoda detekce anomalií má své zjevné výhody:

- Program nemusíme provádět.
- Do programu nemusíme zasahovat.
- Můžeme tak testovat i složité programy a dokonce i takové, o kterých vůbec nic nevíme.
- Existují programové prostředky pro vyhledání anomalií (řada překladačů

upozorňuje na běžné anomálie již při překladu programu, existují však i programové prostředky speciálně zaměřené na vyhledávání anomálí.) Na druhé straně musíme mít na paměti, že:

- zdaleka ne všechny chyby lze takto odhalit;
- v některých případech nemusí anomálie indikovat chybu — mohou být důsledkem nepořádného naprogramování (programátor zapomněl zrušit nedosažitelnou část programu), nebo svérázného programátorského stylu (malou úpravou se nedosažitelná část zpřístupní a máme jinou verzi programu).

Metoda testovacích dat

Detekce anomálí patří mezi statické testovací prostředky, tj. takové, které nepředpokládají provádění programu. Metoda testovacích dat je naproti tomu typickým představitelem dynamických testovacích prostředků, které jsou založeny na provedení výpočtu.

Při použití metody testovacích dat volíme vstupní testovací data takovým způsobem, abychom dopředu věděli, jaké výsledky máme dostat (bude-li program správně fungovat). S těmito daty provedeme výpočet a výsledky porovnáme s předpokládanými. Jestliže se předpokládané výsledky liší od vypočítaných, usuzujeme, že je v programu chyba.

I když se takový úsudek se zdá naprosto samozřejmý, je zde třeba jisté obezřetnosti. Výsledky nemusí souhlasit například vinou zaokrouhlovacích chyb, které vznikají při provádění aritmetických operací, tj. v důsledku numerické instability (viz slovníček) zvoleného algoritmu. Použití nevhodného algoritmu nebo nevhodné naprogramování algoritmu můžeme považovat samozřejmě také za chybu; jedná se však o kvalitativně jiný druh chyby, než je překlep nebo běžné sémantické chyby.

Metoda testovacích dat se v praxi s oblibou používá. Přesto má závažné nevýhody. Především může být někdy obtížné najít taková testovací data, k nimž bychom současně znali správné výsledky. To se dá do jisté míry obejít (za cenu snížení účinnosti metody) tím, že výsledná data posuzujeme podle méně náročných kritérií; všimáme si například, leží-li výstupní hodnoty v předpokládaných rozmezích nebo jsou-li pravděpodobně či aspoň smysluplné.

Existují i jiné možnosti. Můžeme například napsat nový program, který nás problém vyřeší, ale pouze pro určitá konkrétní data. To může být samozřejmě mnohem snazší, než vytvoření programu pro původně zadáný obecný problém. Takový kontrolní program bychom však museli znovu vytvářet i pro jiná testovací data, a to je na první pohled nepohodlné. Nadmíru pracná je i jiná varianta — vytvoření „dublujícího programu“ (tj. pokud možno jednoduché a fungující verze za cenu jistého — ne tak dramatického — omezení v univerzálnosti, v rychlosti nebo ve struktuře).

Uvažme nyní, jak je metoda testovacích dat spolehlivá. Je zřejmé, že její

účinnost se projeví, jestliže pro zadaná testovací data budou získané výsledky v rozporu s očekávanými. Ale pokud k tomu nedojde, nemáme vůbec žádnou záruku, že by k chybě nedošlo pro jiná testovací data! Navíc jsme schopni otestovat program na poměrně malé množině testovacích dat — a všech možných vstupních dat (i pokud se omezíme jen na ta smysluplná) je zpravidla příliš mnoho, takže naše sonda může zachytit vždy jen jejich zlomek.

Přes tyto výhrady bychom neměli posuzovat metodu testovacích dat příliš přísně; víme přece, že koneckonců žádná metoda nemůže být stoprocentní. V programátorské praxi je dost běžné, že autor programu získá neotřesitelnou víru ve svůj výtvor hned v okamžiku, kdy ho otestuje na několika exemplářích vstupních dat a náhodou se neobjeví žádná chyba. Pokud se nejedná o velmi jednoduché a průzračné programy, nebývá taková důvěra obvykle opodstatněná. I když program často po měsíce bezchybně funguje, dojde po jednou k havárii nebo se ukáže, že některá z funkcí programu není v pořádku. Takový průběh se vlastně stává zařízenou zvyklostí — uživatel programu s výskytem chyb (ovšem v přiměřené míře) většinou víceméně počítá, a pokud s nimi počítá i tvůrce programu a je schopen chyby rychle odstranit, je všechno OK.

V praxi patří metoda testovacích dat k nejpoužívanějším. Logicky proto vyvstává otázka: Jak zvýšit její spolehlivost vhodným výběrem testovacích dat?

Pochopitelně bychom více důvěrovali takovému otestování, při kterém by kontrolní výpočty prošly každou větví programu. Pro některé jednodušší případy se může podařit takovou množinu testovacích dat sestavit. Bohužel, platí pesimistické, ale dokázané tvrzení, které říká, že obecně to nejde. Přesněji řečeno: Neexistuje algoritmus, který by na základě zdrojového tvaru programu vytvořil takovou množinu testovacích dat. Můžeme si ovšem naši situaci opět trochu zjednodušit a využít z toho, že pokud je program správně napsán, podaří se nám pravděpodobně projít všemi jeho větvemi (nebo aspoň většinou z nich), jestliže naše množina testovacích dat pokryje typické situace přípustných dat a rovněž mezní situace (které mohou být ošetřeny speciálně).

V některých případech se testovací data „vyrábějí“ automaticky prostřednictvím speciálních programů — generátorů testovacích dat. Takové prostředky jsou však vhodné spíše pro komplikovanější a speciální případy.

Testování metodou „shora dolů“

V kapitole *O programátorském stylu* jsme se zmínili o tom, že k dobrým programátorským mravům patří programování metodou „shora dolů“ (top-down). Kromě výhod, o kterých jsme se již zmínili, má tento styl i další přednost — program vytvořený „shora dolů“ můžeme podobným postupem také testovat.

Princip této testovací metody je jednoduchý: Nejdříve budeme testovat modul nejvyšší úrovně, tj. řídící program. Moduly o úrovni nižší, tj. podpro-

gramy, které jsou volány z řídicího modulu, nahradíme pro účely testování „maketami“. Tyto makety budou velmi jednoduché podprogramy, které se budou volat stejným způsobem jako odpovídající skutečné podprogramy, a ne provedou nic jiného, než že vrátí řídicímu podprogramu rozumná výstupní data. Například podprogram pro výpočet daně ze mzdy, který je dost komplikovaný, nahradíme pro účely testování programů vyšší úrovně takovou maketou:

```

100 SUB DAN(H, S, D)
110 REM !!! POZOR - MAKETA !!!
120 REM PRO HRUBOU MZDU H A DANOVOU SKUPINU S
130 REM SPOCITA DAN ZE MZDY D
140 PRINT "VYVOLANA MAKETA PODPROGRAMU DAN"
150 LET D=0. 2*H
160 SUBEND

```

Výstupní hodnota (daň ze mzdy) se nepočítá zatím správně (nezávisí ani na zadávané daňové skupině), ale není příliš vzdálena od skutečnosti. Maketa tedy nevykoná to, co by vykonal odpovídající skutečný podprogram, ale bude jeho činnost jednoduchým způsobem simulovat.

Budeme-li nyní testovat takto upravený program a objeví-li se chyba, můžeme usoudit, že chyba je v nejvyšším modulu (tj. v řídicím programu) — makety jsou příliš jednoduché, takže v nich chyba asi nebude. Tím jsme zredukovali testování celého programu na testování nejvyššího modulu, což je pochopitelně mnohem jednodušší.

Po otestování řídicího modulu můžeme přejít na bezprostředně podřízené moduly, tj. na podprogramy, které jsou z něho přímo volány. Nyní nahradíme makety ty podprogramy, které jsou volány z testovaných modulů, a proveďme testování. Protože řídicí program již považujeme za správný, prověřujeme v této fázi moduly o úrovni nižší. A tak otestujeme po jednotlivých úrovních postupně všechny moduly.

Je zřejmé, že vhodné rozčlenění programu může efektivitu testování výrazně ovlivnit.

Testování a sledování výpočtu

Se sledováním jsme se setkali už při popisu ladících metod. Mnohé prostředky pro sledování můžeme výhodně uplatnit i při testování programu. Pomocí příkazů sledovacího jazyka je možno například změnit průběh výpočtu tak, aby prošel dosud neprověřenou cestou nebo můžeme otestovat reakci programu v případě, ke kterému by teoreticky nemělo dojít (vzpomeňme si na defenzivní programování).

Sledováním se dá testování velmi urychlit, zvláště je-li kombinováno s ostatními metodami a je-li jeho strategie předem dobře promyšlena. Velmi užitečné může být například využití softwarových prostředků pro analýzu cest výpočtu. Tyto programy dovolují vytvořit (pro daná testovací data) sta-

tistiku o tom, kolikrát výpočet prošel jednotlivými větvemi programu. To nám umožní zejména registrovat ty větve, kterými se neprošlo, a ty otestovat s využitím manipulačního programu.

Jak netestovat programy

Kromě testovacích metod, o kterých jsme se v předchozích odstavcích zmínilo, existují další metody a jejich varianty (např. různé metody založené na predikátové logice, symbolické testování, retrospektivní sledování atd.). O nich se čtenář dozví více z publikací uvedených v přehledu literatury na konci této knihy (viz [MYER79], [BROW81], [HOŘE82]).

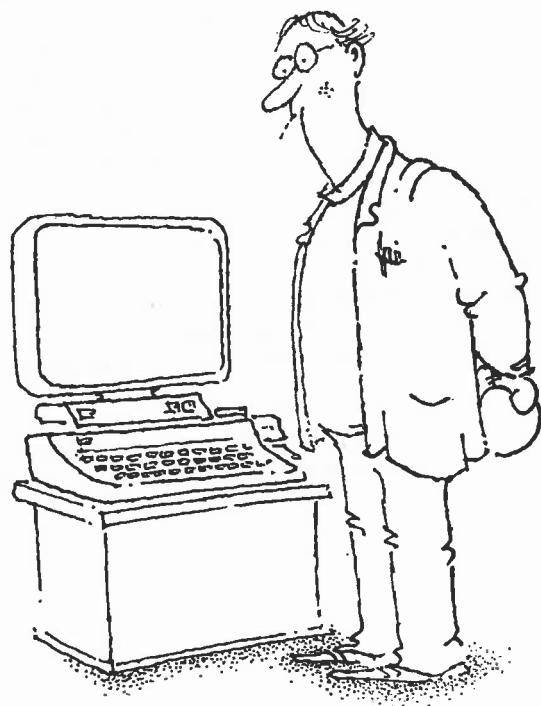
Kromě toho, že by si každý programátor měl osvojit jednotlivé testovací metody (zkušenosti s jejich používáním nasbírá s léty praxe), je důležité, aby každý komplikovanější program tvořil již s ohledem na předem zvolenou strategii jeho testování. Opomíjení této zásady vede k programátorským pokleskům, které jsou výstižně charakterizovány v knize [BROW81]:

- Autor napiše program bez přihlédnutí k testování.
- Program se při prvním zkušebním výpočtu záhadným způsobem zhroutí.
- Programátor si narychlo najde v příručce kapitolu o testování a rozhodne se použít hned první prostředek, na který při čtení narazí.
- Testovací prostředek zařadí do programu.
- Počítač vychrlí obratem neočekávaný stoh potištěného papíru.
- Programátor se cítí podveden: „K čertu, ty testovací pomůcky jsou nanic! V životě už je nepoužiju!“
- Protože je však nutno v práci pokračovat, předloží svůj problém systémovému programátorovi s odůvodněním, že chyba je pravděpodobně v operačním systému, neboť mu špatně pracuje bezchybně přeložený program.

14. HISTORIE JEDNOHO PROGRAMU

Představa, že programování by bylo možno shrnout do výuky receptů, je mylná.

Niklaus Wirth



Theorie programování nás učí, jak metodicky správně analyzovat problém, navrhnut vhodný algoritmus a napsat přehledný, čitelný a strukturovaný program. Praxe nás učí, jak se vyrovnat s tím, že nic z toho pořádně neumíme. Programování není žádná idylka — je to boj. Boj s problémem, překladačem, operačním systémem, počítačem, nečekanými situacemi, které se nenajdou v učebnicích, a především s vlastní hloupostí, neznalostí a netrpělivostí.

Proto nemusí být neužitečné podívat se na věc z této stránky. Mnoho publikací hovoří o tom, jak by programování mělo vypadat, některé (zčásti i naše knížka) o tom, jak by vypadat nemělo. Tato kapitola se pokouší ukázat programování, jak skutečně vypadá.

Pokušení počítače

Mohu odolat všemu kromě pokušení.
Oscar Wilde

Není tomu tak dávno, co se Petr rozhodl pro mikropočítač. Bylo to rozhodnutí poměrně odvážné, uvážíme-li okolnosti — skromný plat inženýra pár let po ukončení školy, nezařízený byt, žena a dvě děti. Ale počítače mají svou magickou přitažlivost a Petr patří k těm, kteří jí neodolali.

Kde Petr sehnal příslušnou nekřesťanskou sumu, ví bůh. Ukázalo se však, že to byl nejmenší z problémů. Získat souhlas vlastní manželky s tím, že se koupí mikropočítač místo mrazničky, se zdá být úkol přesahující lidské možnosti. I to se nakonec podařilo. Sehnat počítač bylo dobrodružství, proti němuž je Odyssea selankou a které dokonale prověřilo Petrovy schopnosti. Když nakonec přece jen uspěl, byl vyčerpán, ale štasten.

Nastalo období, kdy musel přesvědčit svou ženu, že počítač Commodore 64 bude také přínosem pro domácnost a rodinu, jak nasliboval. A tak vznikl i

Projekt „automapa“

„Až si jednou koupíme auto (a to nebude za dlouho, neboť díky racionálnímu využívání našeho počítače ušetříme spoustu peněz), budeme šetřit pohonné hmoty tím, že nám počítač naleze nejkratší cestu k cíli. Zadáme-li počítači místo, odkud vyjíždíme, a koncové místo, kam jedeme, vypíše nejkratší trasu, která tato místa spojuje.“

To byl jeden z argumentů pro mikropočítač. Petr zná jazyk Basic (jehož překladač je na Commodoru k dispozici), takže se mu zdálo vytvoření programu snadné — koneckonců nalézt nejkratší trasu „ručně“ na automapě také není obvykle obtížné. A tak se pustil do práce.

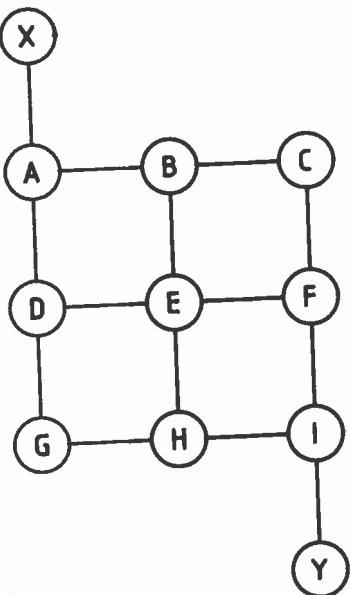
Jak se rodí algoritmus

Na základě Petrova vyprávění se pokusíme o přibližnou rekonstrukci jeho myšlenek:

... s tímhle programkem jsem za dvacet minut hotov ...
... s algoritmem se nebudu párat, probereme všechny možné cesty a hoto-
vo ...
... kolik jich asi tak může vlastně být ...
... tak třeba pro takovouhle jednoduchou „automapku“ ...

Petr si na kousek papíru načrtl „automapku“: (obr. 7)

obr. 7



... kolika způsoby se mohu dostat z jedné „vrstvy“ měst o vrstvu níže ...
... z vrstvy A B C třemi způsoby z vrstvy D E F také třemi, a tak dále ...
... takže celkem je to aspoň tři na třetí možností ...
... ve skutečnosti víc, protože existují i další cesty ...
... pro čtverec deset krát deset měst to bude víc než deset na desátou ...
... kdybych na každou cestu spotřeboval jenom deset operací, budu to při rychlosti sto tisíc operací za vteřinu počítat aspoň milión sekund ...
... no nazzar ... skoro čtrnáct dní ...
... to by mě má drahá vyhodila oknem i s počítačem ještě před ukončením výpočtu ...
... vymyslím lepší algoritmus ...

Zde na chvílku přerušíme tok Petrových myšlenek. Uchýlili jsme se při tom vlastně jen k záznamu útržků myšlenek, protože jinak snad ani nelze zachytit pohyb v mysli člověka pracujícího na vyřešení problému. Dotyčný přestává vznímat své okolí, začíná ohryzávat tužky a jiné psací potřeby (přestože k tomu není nutkán pocity hladu) a je dočasně společensky naprostě nepoužitelný.

Ve skutečnosti právě tak a tehdy přicházejí na svět myšlenky. Rodí se nezpozorovaně a často nás překvapí právě v okamžiku, kdy už ztrácíme naději, že se nám podaří něco rozumného vymyslet.

Heuréka!

*Duševní vlastnosti, které se nazývají analytickými, jsou samy o sobě jen pramálo přístupny analýze.
E. A. Poe*

Jakmile se myšlenka narodí, dá se formulovat, a my se můžeme opět vnořit do Petrovy myсли.

... proboha, to je hodin ...
... vůbec mi to nemyslí ...
... měl bych toho nechat ...
... jak jen na to jít ...
... pomohlo by, kdybych znal vzdálenosti mezi všemi jednotlivými městy? ...
... ! ? ! ...
... ! ! ! ...
... samozřejmě! ...
... pak by to už bylo jednoduché ...
... projdu všechny sousedy počátečního města X a vyberu z nich takové město Z, jehož vzdálenost od X plus vzdálenost do koncového Y bude rovna vzdálenosti z X do Y. V tom případě bude Z ležet na nejkratší cestě. Pak proberu všechny sousedy Z a najdu další město na nejkratší cestě ze Z do Y, a tak dále ...
... ? ? ? ...
... to je sice pěkné, ale vzdáleností všech možných měst je zase strašně moc ...
... ani by se mi to nevešlo do paměti počítače ...
... co s tím? ...
... vlastně bych ani nepotřeboval vzdálenosti všech měst ...
... stačilo by znát vzájemné vzdálenosti měst ležících na nejkratší cestě ...
... jenže nevím, která města to jsou ...
... nebo by stačilo znát jenom vzdálenosti měst od počátečního města X ...
... těch je poměrně málo ...
... to vlastně nejde — neznal bych pak vzdálenost ze Z do Y ...
... ? ? ? ...
... ? ! ? ...
... ! ! ! ...
... ale mohl bych určovat nejkratší cestu počínaje koncovým městem Y! Projdu všechny sousedy Y a vyberu z nich takové Z, jehož vzdálenost od Y plus jeho vzdálenost od X (tu tentokráté znám) je rovna vzdálenosti z X do Y (taky znám). Takové Z leží na nejkratší cestě z X do Y. No a pak buď postupovat analogicky dál, až se dostanu do X ...
... takže jde o to najít algoritmus, jak z tabulky vzdáleností sousedních měst efektivně zjistit vzdálenosti měst od počátečního města X ...

... pro města sousedící s X to je jasné — to jsou přímo vzdálenosti z tabulky vzdáleností sousedních měst ...

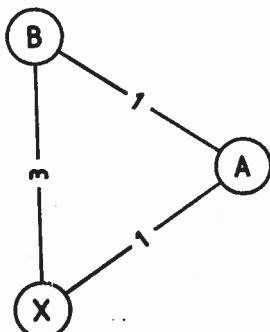
... ? ? ? ...

... vlastně ne ...

... to nemusí být pravda ...

... třeba v takovémto případě: X, A a B sousedí každý s každým, přičemž přímá cesta z X do A je dlouhá 1 km, z X do B 3 km a z A do B rovněž 1 km, takže pojedeme-li z X do B „oklikou“ přes A, ujedeme 2 km, zatímco „přímá“ cesta je o kilometr delší ...

Petr si zde promítl v duchu tuto situaci: (obr. 8)



obr. 8

... také vzdálenosti zahrnuté v tabulce délek silnic spojujících sousední města nemusí být nutně nejkratšími vzdálenostmi mezi těmito městy ...
... to je vlastně jasné — mezi dvěma sousedními městy mohou převést dvě přímé silnice (nebo i více), které nemusí být stejně dlouhé ...

... ? ? ? ...

... ! ! ! ...

... vyberu-li však ze všech silnic přímo spojujících X se sousedními městy tu nejkratší, je zaručené, že aspoň ta se „obejít“ nedá — obcházela by se nutně přes delší silnici ...

... tím určím, které z měst sousedících s X má k němu nejbližše; označím si ho třeba X1 ...

... jak ale určit vzdálenost dalších měst od X? ...

... ? ? ? ...

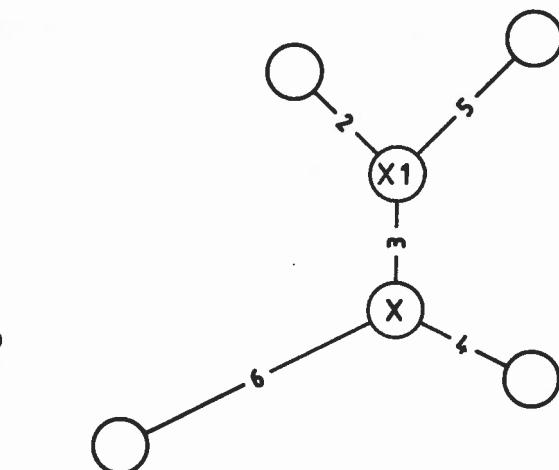
... ! ? ! ...

... Zkusím si to nakreslit ...

Petr si nakreslil tuto mapku: (obr. 9)

... ! ? ! ...

... ! ! ! ...

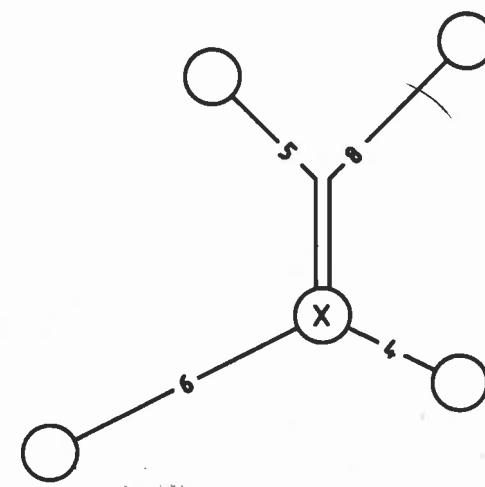


obr. 9

... na X1 zapomenu! Vzdálenost do X1 znám a tím pádem mě už jako město nemusí zatím zajímat. Na chvíli si X1 představím jako patník u cesty. Tím mi sice přibudou „přímé“ silnice, které vedou přes X1, ale to nevadí ...

... vtip spočívá v tom, že snadno najdu vzdálenost některého dalšího města, které teď (poté, co jsem dočasně udělal z X1 „patník“) s výchozím městem sousedí ...

Petr si opět načrtl změněnou situaci: (obr. 10)



obr. 10

... tohle město si pojmenuji X2, udělám z něj „patník“ a budu tak pokračovat pořád dál!...
 ... až se dostanu k městu Y...
 ... ke každému „patníku“ budu znát nejkratší vzdálenost od X, a přitom se bude nejkratší cesta z X do Y skládat ze samých „patníků“...
 ... a to mi vlastně pro nalezení nejkratší cesty stačí...
 ... no a pak zase udělám z patníků města...
 ... docela prosté, milý Watson!...

Algoritmus je na světě

Druhý den dal Petr algoritmu definitivnější podobu. Usoudil, že přidávání „nových“ silničních úseků v procesu „patníkování“ měst není nutné — stačí ke každému městu, které sousedí s X nebo s kterýmkoli „patníkem“, najít nejkratší vzdálenost z X do tohoto města, ze všech takových měst vybrat to, pro něž tato vzdálenost vyjde nejmenší, a z něho udělat „patník“. Vzdálenosti „patníků“ od X si ovšem musíme zapamatovat.

Petr si zapsal algoritmus v bodech:

- (1) Vzdálenost z X do X je nula — uděláme z X „patník“.
- (2) Každému sousedu X přiřadím číslo, které udává, jak dlouhá je nejkratší přímá silnice z X do tohoto souseda. Tomuto číslu budu říkat „dočasná vzdálenost“ (protože to ještě nemusí být skutečná vzdálenost z X).
- (3) Vyberu město, které má nejmenší dočasnou vzdálenost od X, a udělám z něj „patník“. Tato dočasná vzdálenost je ale skutečná, a proto si ji pojmenuji.
- (4) Každému městu, které sousedí s nějakým „patníkem“, přiřadím opět dočasnou vzdálenost: bude to nejmenší ze všech možných čísel: *vzdálenost od X k libovolnému sousednímu „patníku“ + délka kterékoli přímé silnice z patníku do města.*
- (5) Vyberu město s nejmenší dočasnou vzdáleností, udělám z něj patník a zapíši vzdálenost.
- (6) A pak pokračuji zase bodem (4), a tak pořád dál, až se dostanu k městu Y.

Potom Petra napadlo, že v bodě (4) se bude zbytečně znova počítat spousta dočasných vzdáleností; měnit se mohou totiž jenom ty dočasné vzdálenosti, které se týkají sousedů naposledy „zpatníkovaného“ města. Takže bod (4) pozměnil:

(4) Každému městu, které je sousedem naposledy „zpatníkovaného“ města, přiřadím dočasnou vzdálenost, již bude buď předchozí dočasná vzdálenost, anebo nejmenší ze všech vzdáleností k „patníku“ + od „patníku“ do města, a to podle toho, která z těchto dvou vzdáleností bude menší.

Nakonec si Petr ještě všiml, že bod (2) je teď vlastně zahrnut v novém bodě (4), a tak to všechno ještě jednou přepsal:

Algoritmus 1:

- (1) „Zpatníkuj“ X.
- (2) Každému sousedovi naposledy „zpatníkovaného“ města přiřadím dočasnou vzdálenost (jako v předchozí verzi algoritmu v bodu (4)).
- (3) Město s nejmenší dočasnou vzdáleností (a to je již skutečná vzdálenost) „zpatníkuj“ a vzdálenost zaznamenám.
- (4) Je-li toto město Y, pak končím, jinak pokračuji bodem (2).

A pak si ještě zapsal druhý algoritmus, který se provede po algoritmu 1 a který vypíše nejkratší cestu z X do Y (města ležící na této cestě budou však vypisována v opačném pořadí, první tedy bude Y). Tady se už berou „patníky“ zase jako města.

Algoritmus 2:

- (1) Vypíše se Y.
- (2) Ze sousedů naposledy vypsaného města se vybere takové město Z, jehož vzdálenost do posledně vypsaného města plus jeho vzdálenost z X se rovná vzdálenosti z X do posledního vypsaného města; potom se vypíše Z a pokračuje se zase na (2).

Koncepce programu

Beztak to nebude fungovat.

Tzv. Jenkinsonův zákon

Jak uvažuje programátor nad koncepcí programu? Pokusme se opět záchytit Petrovo přemýšlení:

- ... a teď to všechno pěkně naprogramuju ...
- ... také program ...
- ... za prvně načte data ...
- ... za druhé provede algoritmus 1 ...
- ... za třetí provede algoritmus 2 ...
- ... výsledky se ve vhodné formě zobrazí na displej ...
- ... a teď podrobněji ...
- ... jaká data budu načítat ...
- ... počáteční město a koncové město ...
- ... to je vlastně všechno ...
- ... ale budu ještě potřebovat údaje o automapě ...
- ... ty zapíšu přímo do programu příkazem DATA ...
- ... přínejmenším v první variantě programu se omezím na menší fragment automapy, takže data nebudou příliš rozsáhlá ...
- ... jak budu data kódovat ...
- ... jméno výchozího a cílového města načtu jako text — ale v programu je

budu kódovat jako přirozená čísla 0, 1, 2, 3, atd. — s těmi se bude mnohem
 lépe pracovat...
 ... zakódování a dekódování bude hračka...
 ... a automapu budou představovat trojice čísel:
 [město a]; [město b]; [vzdálenost od a do b]
 ... což v programu zapíšu do tří polí, řekněme X , Y , V ...
 ... takže...
 ... jestli z města označeného číslem 2 povede do města označeného číslem 5
 silnice dlouhá 10 km, bude pro některé I ...
 ... $X(I) = 2$, $Y(I) = 5$ a $V(I) = 10$...
 ... budu ovšem potřebovat znát počet silničních úseků...
 ... ten zapíšu do proměnné M ...
 ... a celkový počet měst do proměnné N ...
 ... tak to bychom měli vstupní data...

S algoritmem 1 se Petr ještě trochu potrápil, než přišel na to, jak ho jednoduše a přehledně naprogramovat. Postup, který se nakonec rozhodl použít, je modifikací algoritmu 1, vhodnou pro naprogramování:

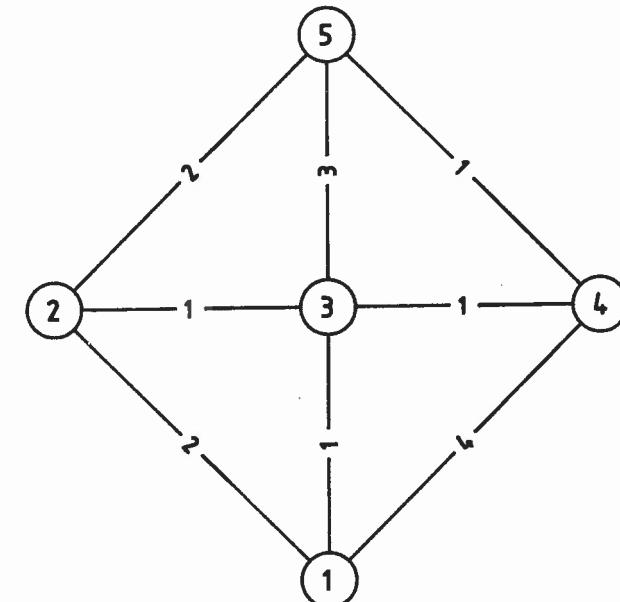
- Každé město má vždy přiřazenu dočasnou nebo skutečnou vzdálenost od výchozího města. U výchozího města je skutečná vzdálenost samozřejmě nula. Na začátku má skutečnou vzdálenost přiřazenu pouze výchozí město. Všechna ostatní města mají na začátku přiřazenu dočasnou vzdálenost, která je větší než jakákoli možná vzdálenost kterýchkoli dvou měst na automapě — řekněme milión.
- Je-li Z naposledy „zpatníkované“ město (tj. takové, kterému byla přiřazena skutečná vzdálenost — prvním takovým městem je výchozí město), Z_1 souřadnice se Z a dočasná vzdálenost Z_1 je větší než nejmenší ze všech čísel $[skutečná vzdálenost Z] + [přímá vzdálenost ze Z do Z_1]$.
potom se toto nejmenší číslo vezme jako nová dočasná vzdálenost Z_1 .
V žádném jiném případě se dočasné vzdálenosti nemění.
- Ze všech měst, jimž dosud nebyla přiřazena skutečná vzdálenost, („nezpatníkovaná“ města), vybereme to, které má nejmenší dočasnou vzdálenost, a tu vezmeme jako skutečnou (tj. toto město „zpatníkujeme“).
- Pokračujeme tak dlouho, až se dostaneme do koncového města.

Takto modifikovaný algoritmus je pro naprogramování jednodušší — umožnuje účinně používat cykly při vyhodnocování skutečných vzdáleností a usnadní i naprogramování algoritmu 2.

Ladění

Petr se rozhodl, že nejprve napiše zjednodušenou variantu programu, ve které se budou města zadávat jako přirozená čísla, a až po odladění této varianty dopíše podprogram, který načte jméno města a přiřadí mu odpovídající

číslo. Program otestuje na malé vymyšlené „automapce“ (obr. 11) — skutečnou automapu zapíše do programu až v jeho definitivním tvaru.



obr. 11

První verze vypisuje nalezenou nejkratší cestu v pořadí, v jakém je určována — to znamená počínaje cílovým městem a konče počátečním městem. Ta to drobná „vada na krásce“ je ale pro odladění programu nepodstatná a odstraní se v definitivní verzi.

A zde je první verze programu:

```

10 REM AUTOMAPA PRVNI VERZE
20 DIM X(200), Y(200), V(200), U(50), L(50)
30 MAT L=ZER
40 READ N
50 DATA 5
60 PRINT "ZADEJ CISLO VYCHOZIHO MESTA"
70 INPUT Ø
80 PRINT "ZADEJ CISLO CILOVEHO MESTA"
90 INPUT I1
100 REM CTENI AUTOMAPY
105 REM DO I-TEHO PRVKU SE UKLADAJI
110 REM UDAJE PRO CESTU Z "X" DO "Y"
130 READ M
140 FOR I=1 TO M
150   READ X(I), Y(I), V(I)
  
```

```

180 NEXT I
190 DATA 8
200 DATA 1, 2, 2, 1, 3, 1, 1, 4, 4, 2, 3, 1, 2, 5, 2
210 DATA 3, 4, 1, 3, 5, 3, 4, 5, 1
220 FOR I=1 TO N
230 LET U(I)=1E6
240 LET L(I)=0
250 NEXT I
260 LET I9=I0
270 LET U(I0)=0
280 REM DOCASNE OHODNOCENI
285 REM BODU SOUSEDICICH S I9
290 FOR I=1 TO M
300 IF X(I)<>I9 THEN 340
310 IF L(Y(I))=1 THEN 340
320 IF U(I9)+V(I)>=U(Y(I)) THEN 340
330 LET U(Y(I))=U(I9)+V(I)
340 NEXT J
350 REM NEJMENSI DOCASNE OHODNOCENI
355 REM SE ZMENI NA TRVALE
360 LET A=1E6
370 FOR I=1 TO N
380 IF U(I)>A THEN 420
390 IF L(I)=1 THEN 420
400 LET I8=I
410 LET A=U(I)
420 NEXT I
430 LET L(I8)=1
440 REM JE DOSAZENO KONCOVEHO BODU?
450 REM JESTLIZE NE, POKRACUJEME:
460 IF I8=I1 THEN 500
470 LET I9=I8
480 GOTO 290
490 REM JESTLIZE ANO,
495 REM VYPISUJEME NEJKRATSI CESTU:
500 PRINT "NEJKRATSI CESTA:"
510 PRINT "CISLO", "KM"
520 LET C=0
530 PRINT I1, C
540 LET I9=I1
550 FOR I=1 TO M
560 IF Y(I)<>I9 THEN 640
570 IF L(X(I))=0 THEN 640
580 IF ABS(U(X(I))+V(I)-U(I9))>=1E-6 THEN 640
590 LET I9=X(I)

```

```

600 LET C=C+VI
610 PRINT I9, C
620 IF I9=I0 THEN 650
630 GOTO 550
640 NEXT I
650 END

```

Petr zapsal program do počítače a spustil příkazem
RUN

Na displej se vypsal:

ZADEJ CISLO VYCHOZIHO MESTA
?

Petr odpověděl:

1

Vzápětí se vypsal:

?SYNTAX ERROR IN 70

Překladač Basiku na mikropočítači Commodore (podobně jako na některých jiných mikropočítačích) hlásí syntaktické chyby až po spuštění programu. V tomto případě překladač sdělil, že k chybě došlo na řádku 70, ale blíž chybu nespecifikoval. Petr vypsal příslušný řádek příkazem

LIST 70

Na displeji se objevilo:

70 INPUT Ø

Syntaktická chyba je vidět na první pohled — místo Ø musí být v příkaze zapsána proměnná, v tomto případě IØ (do které se ukládá číslo počátečního města).

Petr příkaz opravil a spustil program znovu. Další zpráva o chybě zněla:
?ILLEGAL QUANTITY ERROR IN 330

Petr vypsal řádek 330:

330 U(Y(I))=U(I9)+V(I)

V tomto případě je sice chyba komentována (ILLEGAL QUANTITY znamená „nepřípustná veličina“), ale na první pohled není komentář příliš jasný. Zato si brzo všimneme, že proměnná stojící nalevo od rovnítka není dobře uzávorkovaná — chybí poslední pravá závorka. Je tedy skutečně nepřípustně zapsána, což měl překladač na myslí.

Petr opravil příkaz na správný tvar

330 U(Y(I))=U(I9)+V(I)

a znova spustil program. Opět se vypsal chybové hlášení:

?NEXT WITHOUT FOR ERROR IN 340

Doslova přeloženo to není moc srozumitelné („příště bez pro“), ale programátor znalý jazyka Basic ví, že příkazy FOR a NEXT označují začátek a konec cyklu, a že tedy k některému příkazu NEXT nebyl nalezen odpovídající příkaz FOR. A opravdu, na řádku 340 je příkaz

340 NEXT J

ale J není řídicí proměnnou žádného cyklu. Petr snadno zjistil, že se opět jedná o překlep. Správně má příkaz NEXT patřit k příkazu cyklu

290 FOR I=1 TO M
a jeho správný tvar tedy je

340 NEXT I

Petr program znovu opravil a spustil. Tentokrát výpočet proběhl, aniž by se ohlásila chyba:

RUN

ZADEJ CISLO VYCHOZIHO MESTA

? 1

ZADEJ CISLO CILOVEHO MESTA

? 5

NEJKRATSI CESTA:

CISLO KM

5 0

4 0

3 0

1 0

Cesta byla nalezena správně, ale kilometry nesouhlasí — v programu je tedy sémantická chyba, která se týká výpočtu délky nejkratší cesty. Naštěstí se výpočet délky cesty provádí v krátkém úseku mezi řádky 520 až 620 a údaje o délce cesty se ukládají do jediné proměnné C. Když Petr procházel odpovídající příkazy, zarazil se u řádku

600 LET C=C+VI

Nemohl si vzpomenout, jaký význam měla mít proměnná VI. A pak mu došlo, že to nemá být VI, ale V(I). Tím se sémantická chyba vysvětluje: na začátku výpočtu byla v nedefinované proměnné VI nula, která se přičítala místo skutečných vzdáleností silničních úseků k celkové délce C, takže i ta musela na konci vyjít nulová.

Petr příkaz opravil a znovu program spustil:

RUN

ZADEJ CISLO VYCHOZIHO MESTA

? 1

ZADEJ CISLO CILOVEHO MESTA

? 5

NEJKRATSI CESTA:

CISLO KM

5 0

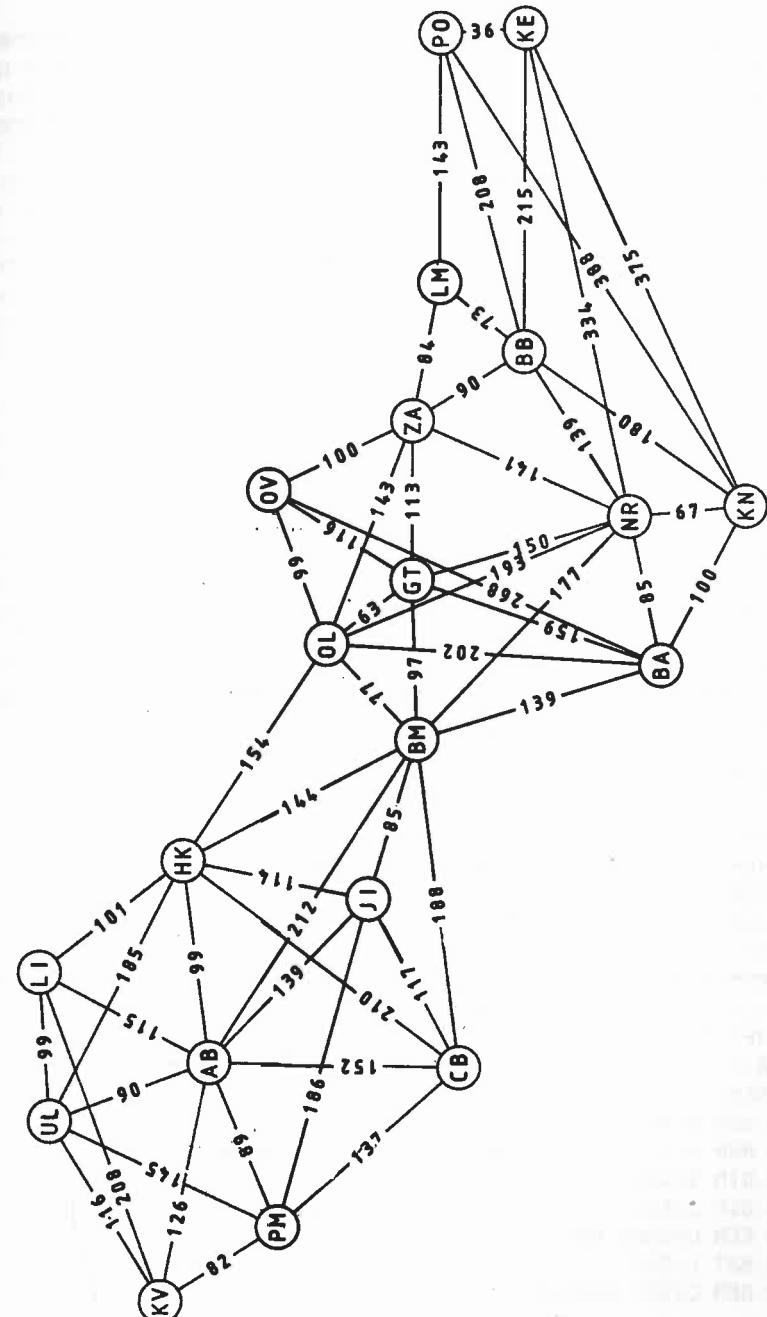
4 1

3 2

1 3

Správný výsledek! Pro jistotu zadal Petr programu ještě nalezení nejkratší cesty z města 2 do města 4. Našla se správná cesta 2 — 3 — 4 a správně se vypsaly i kilometry. Petr usoudil, že tato verze programu je v pořádku, a začal se zabývat další, dokonalejší verzí.

Především se rozhodl uložit do programu údaje o skutečné automapě Československa — i když prozatím ještě podstatně zjednodušené (obr. 12).



obr. 12

V nové verzi programu by se města měla zadávat jménem a nikoli číslem. Pro přečtení jména města, jeho porovnání s tabulkou vybraných měst a nazení pořadového čísla v tabulce napsal Petr krátký podprogram (je umístěn od řádku 1000 v další verzi programu — viz následující odstavec *Program „Automapa“*). To znamená, že vlastní algoritmus bude pracovat opět s čísly měst — budou to jejich pořadová čísla. Petr také upravil příkazy pro vypsání nejkratší cesty tak, aby se vypsala i jména měst a nikoli pouze pořadová čísla.

Nejkratší cesta se zatím vypisovala odzadu — tj. počínaje cílovým a konče výchozím městem. To sice není chyba, ale je to nepraktické a nevypadá to příliš hezký. Je jistě více možností, jak tento nedostatek odstranit; např. zaplatit pořadí měst do pomocného pole a poté je vyspat v obráceném pořadí apod. Petra však napadla tato myšlenka: prostě zamění počáteční a koncové město — jednoduché a elegantní řešení.

Aby se v programu vyznal i po delší době, přidal Petr do programu ještě další komentáře popisující význam použitých proměnných a polí a komentující algoritmus.

Program „Automapa“

Samozřejmě se znova vyskytly syntaktické a sémantické chyby. Nakonec však začal program fungovat podle Petrových představ. Správně nalezl nejkratší cestu z Ústí nad Labem do Bratislavы (přes Prahu a Brno) a z Plzně do Brna (přes Jihlavu). Petr byl spokojen a usoudil, že program je v pořádku. A zde je poslední verze programu:

```

10 REM AUTOMAPA
20 REM VYZNAM HLAVNICH PROMENNYCH A POLI:
30 REM V(I)=DELKA USEKU SPOJUJICIHO
35 REM MESTO X(I) PRIMO S MESTEM Y(I)
40 REM N=POCET MEST
50 REM M=POCET USEKU SPOJUJICICH
55 REM PRIMO SOUSEDNI MESTA
60 REM I1=CISLO VYCHOZIHO MESTA
70 REM I0=CISLO KONCOVEHO MESTA
80 REM M$(I)=NAZEV I-TEHO MESTA
90 REM U(I)=VZDALENOST I-TEHO MESTA
95 REM OD KONCOVEHO MESTA
100 REM L(I)=0: VZDALENOST U(I) JE DOCASNA,
110 REM L(I)=1: VZDALENOST U(I) JE SKUTECNA
120 DIM X(200), Y(200), V(200)
125 DIM U(50), L(50), M$(50)
130 REM UVODNI NAPLNENI PROMENNYCH A POLI
140 MAT L=ZER
150 REM CTENI SEZNAME MEST

```

```

160 READ N
170 FOR I=1 TO N
180 READ M$(I)
190 NEXT I
200 DATA 20
210 DATA BANSKA BYSTRICA, BRATISLAVA, BRNO
215 DATA CESKE BUDEOVICE, GOTTLALDOV
220 DATA HRADEC KRALOVE, JIHLAVA, KARLOVY VARY
225 DATA KOMARNO, KOSICE, LIBERECE
230 DATA LIPTOVSKY MIKULAS, NITRA, OLOMOUC
235 DATA OSTRAVA, PLZEN, PRAHA, PRESOV
240 DATA USTI NAD LABEM, ZILINA
250 PRINT "ZADEJ VYCHOZI MESTO"
260 GOSUB 1000
270 LET I1=I0
280 PRINT "ZADEJ CILOVE MESTO"
290 GOSUB 1000
300 REM CTENI AUTOMAPY.
310 REM DO I-TEHO PRVKU SE UKLADAJI
320 REM UDAJE PRO CESTU Z "X" DO "Y"
330 READ M
340 FOR I=1 TO M
350 READ X(I), Y(I), V(I)
360 NEXT I
370 DATA 55
380 DATA 09, 01, 180, 01, 10, 215, 01, 12, 073
390 DATA 01, 13, 139, 01, 18, 208, 01, 20, 090
400 DATA 02, 03, 139, 02, 05, 159, 09, 02, 100
410 DATA 02, 13, 085, 02, 14, 202, 02, 15, 268
420 DATA 03, 04, 188, 03, 05, 097, 03, 06, 144
430 DATA 03, 07, 085, 03, 13, 177, 03, 14, 077
440 DATA 03, 17, 212, 04, 06, 210, 04, 07, 117
450 DATA 04, 16, 137, 04, 17, 152, 05, 13, 150
460 DATA 05, 14, 063, 05, 15, 116, 05, 20, 113
470 DATA 06, 07, 114, 06, 11, 101, 06, 14, 154
480 DATA 06, 17, 099, 06, 19, 185, 07, 16, 186
490 DATA 07, 17, 139, 08, 11, 208, 08, 16, 082
500 DATA 08, 17, 126, 08, 19, 116, 10, 09, 375
510 DATA 09, 13, 067, 09, 18, 388, 10, 13, 334
520 DATA 10, 18, 036, 11, 17, 115, 11, 19, 099
530 DATA 12, 18, 143, 12, 20, 084, 13, 14, 193
540 DATA 13, 20, 141, 14, 15, 099, 14, 20, 143
550 DATA 15, 20, 100, 16, 17, 089, 16, 19, 145
560 DATA 17, 19, 090
570 REM ZATIM JSOU VSECHNY VZDALENOSTI OD

```

```

550 REM KONCOVEHO MESTA NEKONECNE A DOCASNE:
560 FOR I=1 TO N
570 LET U(I)=1E6
580 LET L(I)=0
590 NEXT I
600 LET I9=I0
610 LET U(I0)=0
615 REM NALEZENI DOCASNYCH VZDALENOSTI
620 REM BODU KTERE SOUSEDI S I9
625 REM (I9 JE NAPOSLEDY ZPATNIKOVANE MESTO)
630 FOR I=1 TO M
640 IF X(I)<>I9 THEN 680
650 IF L(Y(I))=1 THEN 680
660 IF U(I9)+V(I)>=U(Y(I)) THEN 680
670 LET U(Y(I))=U(I9)+V(I)
680 NEXT I
690 REM NEJMENSI DOCASNA VZDALENOST
695 REM SE ZMENI NA TRVALOU
700 LET A=1E6
710 FOR I=1 TO N
720 IF U(I)>A THEN 760
730 IF L(I)=1 THEN 760
740 LET I8=I
750 LET A=U(I)
760 NEXT I
770 LET L(I8)=1
780 REM JE DOSAZENO VYCHOZIHO MESTA?
790 REM JESTLIZE NE, POKRACUJEME:
800 IF I8=I1 THEN 840
810 LET I9=I8
820 GOTO 630
830 REM JESTLIZE ANO, VYPISEME NEJKR. CESTU:
840 PRINT "NEJKRATSI CESTA:"
850 PRINT "CISLO", "KM", "MESTO"
860 LET C=0
870 PRINT I1, C, M$(I1)
880 LET I9=I1
890 FOR I=1 TO M
900 IF Y(I)<>I9 THEN 980
910 IF L(X(I))=0 THEN 980
920 IF ABS(U(X(I))+V(I)-U(I9))>=1E-6 THEN 980
930 LET I9=X(I)
940 LET C=C+V(I)
950 PRINT I9, C, M$(I9)
960 IF I9=I0 THEN 990

```

```

970 GOTO 890
980 NEXT I
990 STOP
1000 REM NALEZENI CISLA MESTA K DANEMU JMENU
1010 INPUT M$
1020 FOR I0=1 TO N
1030 IF M$(I0)=M$ THEN 1120
1040 NEXT I0
1050 PRINT "ZNAM POUZE TATO MESTA:"
1060 FOR I=1 TO N
1070 PRINT M$(I),
1080 NEXT I
1090 PRINT
1100 PRINT "VYBER JEDNO Z NICH"
1110 GOTO 1010,
1120 RETURN
1130 END

```

Na co je dobrá matematika a knihovny podprogramů

*Radost z toho, že někdo objeví něco nového, je omyl starý 6000 let.
Jean Paul*

Vyřešení problému povzbudilo Petrovo programátorské sebevědomí. Program „Automapa“ se přece může hodit i jinde, např. v autodopravě, kde by se jeho využitím daly šetřit pohonné hmoty. Petra začalo zajímat, jestli na takový důležitý algoritmus přišel sám, nebo jestli ho už někdo předešel. Proto se vypravil za svým přítelem Oskarem.

Oskar je matematik a programátor, vyznačuje se vysokým IQ a stejně vysokým stupněm roztržitosti. Kombinace těchto vlastností vyvolávala v Petrovi vždy značný respekt. Petr zastihl Oskara při úklidu jeho bytu (tuto činnost provádí Oskar zcela výjimečně, a to tehdy, když se byt stává neprůchodným a začíná hrozit nebezpečí lavin).

„Snad by sis tu přece jen mohl uklízet častěji,“ poznamenal Petr poté, co se na něj nečekaně zřítil štos odborných časopisů.

„Kdybys znal termodynamické zákony, tak bys věděl, že boj s entropií ve formě nepořádku je v uzavřené soustavě malého bytu předem prohraný,“ odvětil Oskar. „Jaký máš problém?“

Petr mu popsal svůj algoritmus a program. Oskar pravil: „Budu tě muset trochu zklatmat.“

Zároveň v zamýšlení přistoupil k šatníku, zaklepal na jeho dveře, otevřel je a uložil do něj zimník.

„To byly dveře od skříně,“ upozornil ho Petr.

„Aha...“ reagoval nevšímavě Oskar. „Tedy myšlenku jsi měl dobrou, jen ses trochu zpozdil. O tomto problému a problémech s ním souvisejících jsou již napsány celé knihy. Vlastně jsi vytvořil algoritmus a program pro speciální případ takzvaného *problému kritické cesty v ohodnoceném orientovaném grafu*. Mimochodem, metody kritické cesty jsou opravdu velmi důležité a mají významné aplikace nejen v autodopravě, ale i v ekonomii, psychologii, strojírenství, výpočetní technice a dalších oborech. Existuje také celá řada programů a programových systémů pro nejrůznější varianty metod kritické cesty. Není tedy zapotřebí, aby sis takové programy sám vytvářel. Ale věřím, že to mohlo být velmi zajímavé — a potom, člověk v takových případech nejlépe pochopí podstatu problému. A jedná se o skutečně zajímavé problémy,“ pokračoval se zaujetím Oskar, „něco málo ti o tom povím. Vychází se ze základního principu, který se nazývá *princip optimálnosti* a který...“

Následující hodinu poslouchal Petr odbornou přednášku o teorii grafů, metodách CPM a PERT, dynamickém programování a dalších pojmech budících úctu. Zastavit Oskara bylo samozřejmě nemožné.

Z celého výkladu si Petr moc nezapamatoval. Ale už když odcházel, mu bylo jasné, že na příštím programu začne pracovat až po konzultaci s odborníkem a po prozkoumání dostupných knihoven podprogramů.

Epilog

*Člověk se rád směje. Jiným.
S. J. Lec*

Zanedlouho se Petrova rodinka vypravovala na dovolenou do Banské Bystrice (škodovkou, kterou zapůjčili rodiče). „Kudy tam pojedeme?“ otázala se Petrova žena. To byl okamžik, na který Petr čekal.

„Na to přece máme počítač,“ odpověděl hrdě, „a naprosto dokonalý program, který jsem sám vyuvinul. Má zatím jedený drobný nedostatek — budu muset rozšířit data, aby zahrnoval větší množství měst — ale na Banskou Bystrici bude program s přehledem stačit.“

S tím přistoupil ke svému počítači, zapnul ho, nahrál program a spustil výpočet. Na obrazovce se objevilo:

ZADEJ VYCHOZI MESTO

?

Petr odpověděl:

BRNO

Počítač vypsal:

ZADEJ CILOVE MESTO

?

Petr odpověděl:

BANSKA BYSTRICA

Celá rodinka fascinovaně zírala na obrazovku. Po malé chvilce počítač na obrazovce sdělil:

NEJKRATSI CESTA:

CISLO	KM	MESTO
3	Ø	BRNO
2	139	BRATISLAVA
9	239	KOMARNO
10	614	KOSICE
1	829	BANSKA BYSTRICA

Petr nevěřícně zíral a manželka sladce pronesla: „Tak přes Košice? Budeš asi opravdu muset rozšířit ta svá data, miláku. Tuším, že nejkratší cesta do Banské Bystrice povede přes Kamčatku a Nový Zéland.“

Pozorný čtenář si jistě všiml, kde udělal Petr chybu a proč dal počítač chybý výsledek, i když předtím dvakrát počítal správně. Pro toho, kdo tento moment přehlédl, je to drobná detektivní zápletka — všechna fakta jsou k dispozici.

Protože však nechceme riskovat bezesné noci některého ze čtenářů, prozradíme řešení. Pokud jste tedy chybu dosud neodhalili, ale nechcete se připravit o dobrodružství spojené s pátráním po ní, nechte další odstavec.

[Petr si neuvědomil, že informace „z města X vede do města Y silnice dlouhá V kilometrů“ pro program ještě vůbec neznamená, že tato silnice vede také z Y do X. Program (nebo přesněji algoritmus) prostě chápe silnice jako jednosměrné, a pokud je má považovat za obousměrné, musí se do dat zapsat oba směry — což Petr neučinil. Naneštěstí všechny testovací příklady zvolil tak, že se silnice používaly ve „správném“ směru, takže chybu neodhalily. Pokud bychom ovšem chtěli používat program i pro automapy, kde skutečně existují jednosměrné silnice, museli bychom v programu zaměnit pole X a Y. Teprve potom bude trojice X(I) Y(I) V(I) opravdu udávat, že silniční úsek dlouhý V(I) kilometrů vede z města X(I) do města Y(I).]

Závěrem čtenáři doporučujeme, aby se pokusil konfrontovat Petrův postup při programování, ladění a testování se zásadami doporučovanými v této knize a zamyslet se nad tím, do jaké míry by jejich dodržení mohlo zabránit výsledné blamáži.



15. ZÁVĚR

*Naše poslání v životě není mít úspěch,
ale v dobré náladě pokračovat
v chybování.*

R. L. Stevenson

Počítač čeká, a tak nebudeme ztrácat čas obvyklými doporučeními. Pokud se nám podařilo vzbudit ve čtenáři smysl pro krásu počítačové chyby a povzbudit ho, aby nadále chyboval lépe a s větší chutí (tj. aby promyšlenou konцепcí a dodržováním správné metodiky programování omezil banální a typické chyby a věnoval se těm, jejichž odhalování působí oprávněné potěšení), považujeme to za úspěch. A tak jen několik poznámek.

Ani v naší knížce není závěr na konci — následují dvě přílohy, slovníček a odkazy na literaturu. S přílohami a slovníčkem se čtenář pravděpodobně již seznámil. Příloha 1 — *Basic ve zkratce* — je určena k orientačnímu seznámení s jazykem Basic, případně ke stručnému zopakování tohoto jazyka. Příloha 2 — *Znázorňování logiky programu* — ukazuje, jak lze graficky zobrazit strukturu programu. Ve slovníčku naleznete vysvětlení některých odborných termínů z oblasti programování a matematiky, které jsou v knížce použity, nebo které s jejím obsahem souvisí.

Chceme také upozornit na seznam literatury, obsahující nejen díla, ze kterých jsme vycházeli, ale také odkazy na řadu zajímavých a užitečných knih (případně článků) o programování. Snažili jsme se zde uvádět především prameny relativně dostupné, a to i po stránce jazykové. Komentáře, kterými jsou doplněny jednotlivé odkazy, stručně charakterizují jejich obsah a náročnost.

Čtenáři přejeme mnoho zdaru, vytrvalosti a optimismu do dalšího, kvalifikovanějšího chybování. Nezapomeňte, že nejdůležitější ze všeho je dát si dobrý pozor zejména na hrubé chibi!

Na závěr jsme sestavili velmi důmyslný program, který měl dát definitivní odpověď na otázku: jak se stát dokonalým programátorem. Ještě předtím, než se z počítače začalo kouřit a jeho procesor umknul na věky, stačil podat zprávu, kterou nyní intenzívne zkoumáme:

TOTAL SYSTEM&SOFTWARE&HARDWARE CRASH
SYSTEM MESSAGE: GOT0569499HELL956655
THE END - GOODBYE

Příloha 1 BASIC VE ZKRATCE

Basic je jedním z nejrozšířenějších programovacích jazyků. Je k dispozici téměř na všech počítačích — od nejmenších mikropočítačů až po velké střediskové počítače. Na některých mikropočítačích je dokonce jediným dostupným jazykem.

V důsledku velkého rozšíření Basiku existuje značné množství jeho variant. Obdobně jako se různým variantám přirozeného jazyka říká nářečí (dialekty), můžeme mluvit o dialektach i u programovacích jazyků. A právě Basic má těchto dialeků mimořádně mnoho, a to navzdory tomu, že byl (dokonce vícekrát) normalizován. Americká norma z roku 1978 (a jí odpovídající světová norma z roku 1982) popisuje natolik primitivní variantu Basiku, že autoři prakticky všechno jeho překladačů považovali za nutné jazyk oproti normě rozšířit — bohužel každý trochu jinak. Basic podle nové americké normy (její návrh pochází z roku 1985) je sice mnohem dokonalejší, avšak autorům této knihy není zatím znám ani jeden překladač, který by odpovídal této nové normě.

S ohledem na zmíněnou nejednotnost nebudeme v tomto stručném přehledu popisovat jazyk v plné šíři. Omezíme se na jeho nejdůležitější prvky, společně většině dialeků, a navíc uvedeme pouze ty „speciality“, které se vyskytly v textu této knihy. Podrobnejší popis Basiku naleznete v [FRAN87] a [KROH88]; přehled rozdílů mezi rozšířenými dialekty Basiku byl publikován v časopise Elektronika v r. 1989.

1. Základní pojmy

Řádky a příkazy

Program v jazyku Basic se skládá z řádků. Na začátku každého řádku se zapisuje jeho číslo. (Čísla jednotlivých řádků nemusí tvořit souvislou řadu, většinou bývá zvykem číslovat řádky s krokem 10.) Na jednom řádku se obvykle zapisuje jeden příkaz jazyka Basic, i když některé dialekty dovolují psát více příkazů na řádek, nebo naopak, jeden příkaz rozdělit do několika řádků. Pro zvýšení přehlednosti je možno jednotlivé části příkazu od sebe oddělovat mezery (v řadě dialeků je tato mezera na určitých místech povinná). Například následující příkazy jsou totožné:

10 FOR I = 1 TO N - 1

10 FOR I=1 TO N-1

a v některých dialektech lze tento příkaz zapsat i ve tvaru:

10FORI=1TON-1

Mezera má však vždy význam uvnitř textových konstant.

Pokud příkazu nepředchází číslo řádku, není částí programu, ale provádí se ihned (Basic tak funguje podobně jako kalkulačka). Číslované řádky tvoří basikovský program a provádějí se, až je zadána direktiva RUN (viz odstavec *Direktivy v této příloze*). Jednotlivé řádky se potom provádějí v pořadí podle svých čísel, dokud výpočet nedojde k příkazu, který výslovně příkazuje pokračovat na jiném řádku.

Většina příkazů předepisuje, jakou činnost má počítač provést. Takovým příkazům se někdy říká výkonné příkazy. Některé příkazy však žádnou činnost nezadávají — jejich úkolem je podat překladači jazyka Basic určité doplňující informace. Takovým příkazům budeme říkat popisy. (Někdy se označují též jako nevýkonné příkazy nebo deklarace.)

Typy dat

V Basiku se setkáváme se dvěma druhy údajů: číselnými a textovými. Hodnotou číselného údaje je číslo — může to být číslo celé i desetinné, kladné i záporné. Hodnotou textového údaje je znak nebo řada znaků (tato řada může mít i nulovou délku, tj. nemusí obsahovat žádný znak). Za znaky považujeme písmena, číslice nebo speciální znaky (znaménko +, mezeru, hvězdičku apod.).

Konstanty

Číselné konstanty zapisujeme podobně jako v matematice (místo desetinné čárky však píšeme desetinnou tečku) — např.

0 125 3. 141592 -1000.

Velmi velká nebo velmi malá čísla je možno výhodněji zapsat v tzv. exponentovém tvaru; např. místo 3000000000 můžeme psát 3E8 (odpovídá matematickému zápisu 3 · 10⁸), místo -0. 0000015 píšeme stručněji -1. 5E-6 (tj. -1.5 · 10⁻⁶).

Textové konstanty se zapisují do uvozovek, např. "ZACATEK VYPOCTU" nebo "— POZOR !!! —". Samotné uvozovky nejsou součástí konstanty; např. první konstanta má délku 15 znaků, z nichž první je Z a poslední U. Konstanta zapsaná jako "" představuje text o nulové délce. Pokud je nutné, aby součástí konstanty byly uvozovky, musíme je zapsat jako dvoje uvozovky po sobě; např. hodnotou konstanty "REKNI ""ANO"" je text REKNI "ANO", skládající se z 11 znaků.

Mezi číselnými a textovými konstantami je jeden zásadní rozdíl: zatímco zápis 1000, +1000, 1E3 a 1000.0 označují tutéž číselnou konstantu (číslo 1000), zápisy "1000", "+1000", "1E3" a "1000. 0" představují čtyři různé textové konstanty (každá z nich má dokonce jiný počet znaků).

Proměnné

V průběhu výpočtu můžeme čísla a texty ukládat do proměnných, které mohou být rovněž číselné a textové. Každá proměnná se označuje názvem. Název číselné proměnné je buď písmeno, nebo písmeno následované jednou číslicí (např. X nebo A1). V mnoha dialektech (i v nové normě) jsou povoleny i delší názvy proměnných, začínající písmenem a pokračující libovolnou kombinací písmen a číslic (např. OMEGA nebo X2NA3).

Názvy textových proměnných jsou obdobné, avšak přidává se za ně znak \$ — dolar (např. A\$ nebo T9\$). Některé dialekty opět připouštějí názvy složitější (např. NADPIS\$ nebo TAB2A\$), v jiných naopak může být před znakem \$ pouze jediné písmeno (takže v celém programu může být nejvýše 26 různých textových proměnných).

Pole

V řadě případů potřebujeme v programu pracovat se skupinami čísel (např. s vektory a maticemi známými z matematiky, s tabulkami apod.) nebo s obdobnými skupinami textů. Takové skupiny dat můžeme ukládat do polí. Celá skupina (pole) má společný název — písmeno, nebo písmeno následované jednou číslicí (v některých dialektech je opět možno používat složitější názvy nebo naopak nazývat pole pouze jediným písmenem.) Jednotlivé prvky pole se rozlišují indexy, které se píší za název do závorky, např. A1(5) označuje 5. číslo z pole A1, J\$(2, 3) označuje text uložený ve 2. řádku a 3. sloupci „tabulky“ textů nazvané J\$. Rozměry polí se určují popisem DIM (viz dále odstavce „tabulky“).

vec „Popisy“). Nejmenší možný index je zpravidla roven nule (u některých dialektů 1, u mnoha dialektů lze zvláštním popisem zvolit, zda indexy mají začínat nulou nebo jedničkou). Maximální hodnota indexu (a současně celkový počet indexů, tj. *počet rozměrů pole*) je dána výše zmíněným popisem **DIM**. (V mnoha dialektech Basiku se příkaz **DIM** může vynechat a maximální hodnota indexu je potom obvykle 10.) Prvku pole se někdy říká *indexovaná proměnná*.

Výrazy

Podobně jako v matematice je možno i v Basiku vytvářet z hodnot proměnných a konstant další hodnoty pomocí sečítání, odečítání, násobení, dělení a umocňování. Tyto operace se zapisují podobně jako v matematice; násobení se však označuje hvězdičkou *, dělení šikmým lomítkem / a umocňování znakem ^ nebo dvěma hvězdičkami za sebou. Např. basikovský výraz

$(X+1)^{**2}/A(2)$

znamená: K hodnotě proměnné X přičti jedničku, výsledek umocni na druhou a děl hodnotou 2. prvku pole A.

Výrazem může být i samotná konstanta nebo proměnná, takže 12 je číselný výraz (a současně konstanta), G\$ je textový výraz (a současně proměnná).

Textové proměnné a konstanty je možno kombinovat do textových výrazů, které spojením několika textů vytvářejí delší text. Takové spojení, zřetězení, označujeme operátorem & (v některých dialektech operátorem +, který zde neoznačuje „sčítání“ textů, ale jejich zřetězení). Například výraz

"JAZYK "&J\$

označuje text začínající slovem JAZYK, za nímž následuje mezera a dále hodnota textové proměnné J\$. Bude-li mít J\$ hodnotu Basic, je hodnotou výše uvedeného výrazu text JAZYK Basic.

Je samozřejmé, že ve výrazech lze kombinovat pouze číselné hodnoty s číselnými a textovými. Výrazy jako X1/A\$ nebo "KAPITOLA"&3&". jsou chybné.

Funkce

Kromě konstant a proměnných (obyčejných i indexovaných) se ve výrazech mohou vyskytovat funkce. (Tento termín je převzat z matematiky, kde slovo funkce označuje pravidlo, kterým se z určité hodnoty počítá hodnota jiná.) Například ABS(X) označuje absolutní hodnotu z X, SIN(X) značí totéž co v matematice sin x, tj. matematickou funkci sinus, SQR(A+1) je druhá odmocnina z čísla (A+1) apod. Existují i funkce, jejichž argumentem (tj. tím, co je zapsáno v závorce za jménem funkce) nebo výsledkem je text. Např. LEN(A\$) je číslo, které udává, kolik znaků obsahuje proměnná A\$, CHR\$(N) je text tvořený jediným znakem, a to n-tým znakem v „abecedě“ příslušného počítače. Obvykle je tato abeceda určena tzv. kódem ASCII (nebo velmi podobným kódem ISO).

Ne každá funkce má právě jeden argument. Existují i funkce s větším počtem argumentů a funkce bez argumentu. Například MOD(A, 3) počítá zbytek, který zůstane, dělímé-li a třemi, hodnotou funkce PI je Ludolfovovo číslo π , funkce RND vytvoří číslo náhodně vybrané z rozmezí od 0 do 1 (bez náhodných čísel by nemohlo existovat mnoho počítačových her), funkce DAT\$ (v některých dialektech DATE\$) dává jako výsledek text udávající momentální datum. V některých dialektech se za funkčemi bez argumentu může psát alespoň závorky — např. TIM\$() — nebo dokonce zapsat jakýkoli argument, na jehož hodnotě nezáleží — např. PI(0).

Tabulka nejdůležitějších funkcí je uvedena ve třetí části této přílohy. Programátor si může definovat rovněž své vlastní funkce — viz dále popis DEF.

2. Nejdůležitější příkazy

Protože tato knížka není učebnicí Basiku (a též z důvodu shrnutých v úvodu této přílohy), popisujeme zde pouze některé nejdůležitější basikovské příkazy. O příkazech uvedených v závorce se zmíňujeme jen velmi stručně, většinou u nich neuvádíme konkrétní tvar jejich zápisu. Ani popis ostatních příkazů často není úplný a omezuje se na nejčastěji používané možnosti.

Příkazy budeme probírat v následujícím pořadí:

- přířazovací příkazy: LET (MAT)
- vstupní a výstupní příkazy: PRINT, INPUT (MATPRINT, MATINPUT, LINPUT, READ, DATA, IMAGE, FILE, OPEN)
- příkazy skoku: GOTO, IF, ON
- příkazy cyklu: FOR, NEXT
- příkazy pro práci s podprogramy: GOSUB, RETURN, SUB, SUBEND, CALL, DEF (FNEND)
- ostatní výkonné příkazy: STOP, END, RANDOMIZE
- ostatní popisy (nevýkonné příkazy): DIM, REM, OPTION

Příkaz LET

LET *proměnná = výraz*

Do proměnné se uloží hodnota uvedeného výrazu. Proměnná může být i indexovaná (tj. může jít o prvek pole). Je-li proměnná číselná, musí být výraz rovněž číselný; textové proměnné lze přiřadit zase pouze textový výraz.

Ve většině dialektů Basiku se slovo LET může vynechat (tj. příkaz, který nezačíná žádným speciálním basikovským slovem, se automaticky považuje za příkaz LET).

Příklady:

- | | |
|-----------------------|---|
| 10 LET X=0 | (slovo LET je vynecháno) |
| 20 A\$="ANO" | (hodnota I se zvětší o jedničku) |
| 30 LET I=I+1 | |
| 40 LET A(0)=SQR(PI()) | (nultý prvek pole A bude roven $\sqrt{\pi}$) |

Příkaz MAT

Tento příkaz přiřazuje hodnotu celému poli. Například umožnuje pole znulovat nebo do prvků ležících na úhlopříčce dvojrozměrného pole poslat jedničky a do ostatních prvků nuly, či násobit všechny prvky pole stejnou konstantou.

Uvedeme pouze několik příkladů použití příkazu MAT (ve všech V označuje jednorozměrné pole čili vektor, A, B dvojrozměrná pole čili matici):

- | | |
|-----------------|---|
| 10 MAT V=ZER | (vynuluje všechny prvky vektoru V) |
| 20 MAT A=INV(B) | (do A zapíše matici inverzní k B) |
| 30 MAT A=A*B | (A se nahradí součinem matic A a B;
tentou součin se počítá tak, jak je v matematice součin matic definován) |

Příkaz PRINT

PRINT výraz;...; výraz

Vytisknou se hodnoty uvedených výrazů. Může jich být zapsán libovolný počet a mohou být promíchány výrazy číselné s textovými. Hodnoty jednotlivých výrazů se tisknou za sebe (na jeden řádek, pokud se něj vejdu). místo středníku je možno výrazy oddělovat čárkami, potom se tiskne každý následující výraz až od určitého sloupce (závislého na konkrétním dialekton Basiku) – tím se usnadňuje tisk tabulek do sloupců. Je-li středník nebo čárka i za posledním výrazem, pokračuje následující příkaz PRINT v tisku na započatý řádek.

Příklady:

10 PRINT "X="; 1/2	(vytiskne X= 0.5)
20 PRINT "A1="; A1	(vytiskne hodnotu proměnné A1 spolu s jejím názvem)
30 PRINT	(vytiskne prázdný řádek)
40 PRINT "Byl pozdní večer -";	
50 PRINT " první maj.";	(tiskne se hned za pomlčku)
60 PRINT RND, RND, RND	(vytiskne 3 různá náhodná čísla)

Příkaz INPUT

INPUT proměnná, ... , proměnná

Vyzve uživatele (obvykle napsáním otazníku), aby zadal určitá čísla nebo texty (vstupní data). Jejich hodnoty uloží do vyjmenovaných proměnných. Je na uživateli, aby do číselních proměnných zadával čísla a do textových proměnných texty. (A je na autorovi programu, aby uživateli vhodně upozornil, zda se od něj očekává zadání čísel nebo textů.) V některých dialektech je za tím účelem možno do příkazu INPUT zapsat textovou konstantu; ta se potom vytiskne jako výzva k zadávání dat.

Příklady:

10 INPUT A, B\$	(žádá zadání dvou čísel)
20 INPUT J\$	(žádá text)
30 INPUT S, Z\$	(žádá číslo a potom text)
40 INPUT "TVE JMENO?", J\$	(před čtením tiskne výzvu; možné jen v některých dialektech)

Příkazy MATPRINT a MATINPUT

Jsou podobné příkazům PRINT a INPUT, ale slouží k tisku nebo ke čtení celých polí.

Příkaz LINPUT

Tento příkaz přečte celý řádek a uloží ho do textové proměnné (číselné proměnné se v něm vyskytovat nesmějí). Je-li v příkazu uvedeno několik textových proměnných, přečte se několik řádků. V některých dialektech Basiku se příkaz nazývá LINE INPUT.

Příkaz READ a popis DATA

Příkaz READ je podobný příkazu INPUT, ale nežádá zadání vstupních údajů z klávesnice. Vstupní data se čtou přímo z programu, kde musí být zapsána ve speciálním popisu DATA; tento popis může být v programu zapsán na libovolném místě. Příklad:

10 READ M, N, NS

20 DATA 5, 10, "TABULKA 1"

Po provedení příkazu 10 je M=5, N=10, NS=TABULKA 1.

Popis IMAGE

Tímto popisem lze přesně definovat tvar, v němž mají být tištěny výsledky (např. počet desetinných míst, způsob tisku znaménka a mnoho dalších detailů). Příkaz PRINT se potom může odvolat na popis IMAGE, kterým se má tisk řídit.

Příkaz FILE nebo OPEN

Příkaz FILE (v některých dialektech příkaz OPEN) nám umožnuje zadat, aby se některé údaje četly ze souboru anebo zapisovaly do souboru uloženého ve vnější paměti (např. na kazetě nebo na disketu). Tímto příkazem přidělíme souboru určité číslo a příslušný příkaz INPUT nebo PRINT ho potom musí citovat.

Příkaz GOTO

GOTO číslo řádku

Příkazem GOTO se předepisuje, že výpočet nemá pokračovat následujícím basikovským příkazem, ale že se má skočit na jiný řádek.

Příklady:

10 GOTO 150	(pokračuje se příkazem s číslem 150)
20 GOTO 20	(příkaz se opakuje stále dokola – tzv. nekonečný cyklus)

Příkaz IF

IF podmínka THEN číslo řádku

Je-li splněna uvedená podmínka, má se skočit na jiný řádek. Pokud podmínka splněna není, bude výpočet pokračovat následujícím příkazem.

Příklady:

10 IF X>0 THEN 150	(skočí na řádek 150, je-li X kladné)
20 IF Z\$="ANO" THEN 200	(skočí, pokud hodnotou proměnné Z\$ je text ANO)
30 IF N<=5 THEN 150	(skočí, je-li N menší nebo rovno 5)
40 IF A\$<>"*" THEN 20	(kombinace <> znamená „není rovno“)

Poznámky:

- V podmínkách se čísla porovnávají podle velikosti, texty podle abecedy. („Abeceda“ je rozšířena i o číslice a speciální znaky, seřazení těchto znaků do „abecedy“ se však u různých počítačů liší. Nejčastěji je definováno tzv. kódem ASCII.) Tedy např. podmínky 1<5 a „ADAM“<„ADOLF“ jsou splněny, podmínka „A“>„5“ na některých počítačích může být splněna a na jiných nemusí.
- Některé dialekty umožňují uvést v tomto příkazu i několik podmínek spojených slovem AND nebo OR. Skok se provede, jsou-li splněny všechny podmínky (použije me-li AND), nebo je-li splněna alespoň jedna z podmínek (při slovu OR).
- V některých dialektech můžeme psát místo slova THEN slovo GOTO.
- V některých dialektech lze napsat za slovo THEN místo čísla řádku basikovský příkaz (nebo i několik příkazů). Příkaz se provede, je-li splněna uvedená podmínka.

— V některých dialektech je možno napsat na konci příkazu ještě slovo ELSE, následované opět číslem řádku nebo příkazem. To, co je napsáno za ELSE, se naopak uplatní tehdy, pokud podmínka není splněna.

Příklady (použitelné jen v některých dialektech):

```
50 IF A=C GOTO 100
60 IF A=C THEN GOTO 100
70 IF X+Z4>10 OR T$="CHYBA" THEN PRINT "CHYBA"
80 IF A$>="A" AND A$<="Z" THEN GOTO 10 ELSE STOP
```

Příkaz ON

ON výraz GOTO číslo řádku, ..., číslo řádku

Za slovem GOTO může být uveden libovolný počet čísel. Výraz uvedený za slovem ON musí být číselný. Pokud je jeho hodnota rovna jedné, skočí se na první z uvedených řádků; je-li rovna dvěma, skočí se na druhý atd. Je-li jeho hodnota menší než jedna nebo větší než počet zapsaných čísel řádků, je to z hlediska Basiku chyba (i když to některé dialekty povolují). Jestliže hodnotou výrazu není celé číslo, zaokrouhlí se (v některých dialektech se nezaokrouhuje, ale desetinná místa se prostě ignorují).

Místo slova GOTO je možno zapsat slovo GOSUB. Potom se místo obyčejného skoku provede tzv. skok do podprogramu (viz dále příkaz GOSUB). V některých dialektech můžeme místo GOTO napsat slovo THEN (funguje stejně jako GOTO).

Příklady:

```
10 ON I GOTO 20, 30, 50
20 ON 1+INT(2*RND) GOSUB 100, 200
```

V prvním příkladu musí mít I hodnotu od 1 do 3. Hodnota výrazu v druhém příkladu je buď 1, nebo 2 podle toho, zda náhodné číslo vytvořené funkcí RND je větší nebo menší než 0,5. Vybere se proto náhodně buď podprogram začínající na řádku 100, nebo na řádku 200.

Příkazy FOR a NEXT

FOR proměnná = výraz TO výraz STEP výraz
libovolný počet příkazů

NEXT proměnná

(Proměnná uvedená v příkazu FOR se nazývá řídící proměnná; nesmí být indexovaná. Výrazy v příkazu FOR musejí být číselné. Za NEXT musí být uvedena tatáž proměnná jako za FOR.)

Příkazy zapsané mezi FOR a NEXT se provádějí opakováně několikrát po sobě. Na začátku má řídící proměnná hodnotu uvedenou v příkazu FOR za rovníkem, při každém opakování k ní příkaz NEXT přičte hodnotu zapsanou v odpovídajícím příkazu FOR za slovem STEP a příkazy mezi FOR a NEXT se provádějí znova tak dlouho, dokud není překročena hodnota zapsaná za TO. Slovo STEP (včetně výrazu za ním) může být vynecháno; v tom případě se hodnota řídící proměnné v každém opakování zvětšuje o jedničku.

Příklad 1 — tabulka funkce sinus pro úhly od 0 do 360 stupňů:

```
10 FOR X=0 TO 360
20 PRINT X, SIN(X*PI/180)
30 NEXT X
```

Příklad 2 — tip do sportky (pokud by se některé číslo ve výsledcích opakovalo, tip ignorujte a spusťte program znovu):

```
10 FOR I=1 TO 6
20 PRINT 1+INT(49*RND())
30 NEXT I
40 PRINT
```

Příkazy vložené mezi FOR a jemu odpovídající NEXT mohou obsahovat i další příkazy FOR a NEXT, ovšem s jinou řídicí proměnnou. Je tedy možno vložit několik cyklů do sebe. Například následující program vytiskne malou násobilku (předpokládá se, že Basic na daném počítači může tisknout 10 čísel na řádku):

```
10 FOR I FROM 1 TO 10
20 FOR J FROM 1 TO 10
30 PRINT I*j,
40 NEXT J
50 PRINT
60 NEXT I
```

Příkazy GOSUB a RETURN

Příkaz GOSUB má tento tvar:

GOSUB číslo řádku

a příkaz RETURN je tvořen pouhým slovem RETURN.

Podobně jako příkaz GOTO skočí GOSUB na řádek s uvedeným číslem, navíc však zapamatuje, odkud se tento skok prováděl. Příkazem RETURN se potom skočí zpět (bezprostředně za poslední provedený příkaz GOSUB). To umožňuje vytvořit tzv. podprogram — úsek programu, do kterého lze z různých míst skočit, provést potřebnou část výpočtu a potom se vrátit zpět.

Příklad (náhodné komentování průběhu hry):

```
50 GOSUB 500          (skok do podprogramu)
...
100 GOSUB 500         (opět skok do podprogramu)
...
500 IF BODY<0 THEN 540 (zde začíná podprogram)
520 IF RND>0. 05 THEN 540 (počítac nevyhrává — nedělej nic)
530 PRINT "ZA CHVILKU TE PORAZIM"
540 RETURN             (návrat z podprogramu)
```

Z jednoho podprogramu můžeme volat další podprogram. Příkaz RETURN se vždy vrátí tam, odkud byl volán poslední podprogram, z něhož se výpočet dosud nevrátil.

Příkazy SUB, SUBEND a CALL

Podprogramy realizované příkazy GOSUB a RETURN mají jednu podstatnou nevýhodu: pokud se v nich pracuje s proměnnými (nebo s poli), je nutno před jejich vyvoláním přiřadit do proměnných, které se v podprogramu používají, potřebné hodnoty a po ukončení podprogramu zase zapamatovat hodnoty spočítané v podprogramu, neboť by se dalším vyvoláním přepsaly. To je dosti pracné a snadno se zde udělá chyba nebo se něco přehléde. Tuto nevýhodu odstraňuje nový druh podprogramů, který je k dispozici v modernějších dialektech Basiku. Tyto podprogramy začínají popisem SUB a končí popisem SUBEND (v některých dialektech END SUB). Volají se příkazem CALL. V příkaze CALL je přitom možno zadat názvy proměnných, s nimiž má podprogram pracovat (tzv. skutečné parametry), a uvnitř podprogramu je možno s těmito proměnnými pracovat

pod jinými názvy, uvedenými v popisu **SUB** (tzw. formální parametry). Všechny akce související s navázáním formálních a skutečných parametrů se provedou automaticky. Tento způsob předávání informací do podprogramu a z něho je mnohem jednodušší a pohodlnější.

Stejně jako u podprogramů typu **GOSUB** je možno z jednoho podprogramu volat druhý.

Uvedené příkazy mají tvar:

SUB název podprogramu (parametr, ..., parametr)

SUBEND nebo **END SUB**

CALL název podprogramu (parametr, ..., parametr)

Příklad (výpočet délky trvání časového intervalu):

10 PRINT "CAS ZACATKU: "	
20 CALL CTICAS(Z)	(podprogramem čteme čas začátku)
30 PRINT "CAS KONCE: "	
40 CALL CTICAS(K)	(týmž podprogramem čteme čas konce)
50 PRINT "TRVALO"; K-Z; "SEC"	
60 GOTO 10	(vracíme se — počítáme další interval)
70 END	(konec hlavního programu)
100 SUB CTICAS(C)	(zde začíná podprogram)
110 PRINT "HODIN?"	
120 CALL CTI1(H, 24)	(vnitřním podprogramem čteme hodiny)
130 PRINT "MINUT?"	
140 CALL CTI1(M, 60)	(týmž podprogramem čteme minuty)
150 PRINT "SEKUND?"	
160 CALL CTI1(S, 60)	(týmž podprogramem čteme sekundy)
170 LET C=3600*H+60*M+S	(C je počet sekund od půlnoci)
180 SUBEND	(zde končí podprogram)
200 SUB CTI1(I, N)	(zde začíná vnitřní podprogram)
210 INPUT I	
220 IF INT(I) <> I THEN 210	(není celé číslo — opakuj)
230 IF I < 0 OR I >= N THEN 210	(chybná hodnota — opakuj)
240 SUBEND	(zde končí vnitřní podprogram)

Popisy **DEF** a **FNEND**

DEF název (argument, ..., argument) = výraz

Tímto popisem může programátor definovat svou vlastní funkci a rozšířit tak sortiment dostupných funkcí. Ve většině dialektů může mít název funkce pouze tvar **FNx** nebo **FNx\$**, kde *x* je libovolné písmeno; v programu lze proto definovat 26 číselných a 26 textových funkcí. Takto definované funkce se volají stejně jako základní funkce, které jsou součástí jazyka Basic. Funkce může mít libovolný počet argumentů (je možno definovat i funkce bez argumentů, podobně např. standardní funkci **RND**).

Příklad:

Program pro tip do sportky, který jsme uvedli u příkazu **FOR** a **NEXT**, můžeme zapsat i takto:

10 DEF FNR(N)=1+INT(N*RND)
20 PRINT FNR(49); FNR(49); FNR(49);
30 PRINT FNR(49); FNR(49); FNR(49);
40 PRINT FNR(35); FNR(35); FNR(35); FNR(35); FNR(35)

(Rádek 40 přidá navíc tip do Matesa.)

V některých dialektech je popis **DEF** rozšířen tak, že je možno jím definovat funkce zadané složitěji než pouze jedním výrazem. V tom případě se v popisu **DEF** vynechá = výraz a za tímto popisem následují další řádky, kde se spočítá hodnota funkce a zapíše se do proměnné **FNx** či **FNx\$**. Definice takové funkce potom končí popisem **FNEND** (tvořeným pouze slovem **FNEND**). V jiných dialektech popis takových „vícezádkových“ funkcí nezačíná slovem **DEF**, ale slovem **FUNCTION** a končí popisem **END FUNCTION**.

Příkazy **STOP** a **END**

Příkazem **STOP** se ukončuje výpočet. Může být zapsán kdekoli v programu. Příkaz **END** rovněž ukončuje výpočet, avšak smí být zapsán jen na konci programu (ve většině dialektů zde být zapsán musí). Za příkazem **END** mohou následovat pouze definice podprogramů, začínající popisem **SUB**. Oba příkazy jsou tvořeny pouze slovem **STOP**, respektive **END**.

Příkaz **RANDOMIZE**

RANDOMIZE

nebo

RANDOMIZE číselný výraz

Příkaz **RANDOMIZE** ovlivňuje tvorbu náhodných čísel generovaných funkcí **RND**. Pokud v programu nebyl proveden příkaz **RANDOMIZE**, vytváří se každým vyvoláním funkce **RND** sice jiné náhodné číslo, avšak při každém spuštění programu se vytváří stejná řada náhodných čísel. Použijeme-li příkaz **RANDOMIZE** v prvním uvedeném tvaru (bez následujícího výrazu), budou se v každém běhu programu vytvářet jiná náhodná čísla. Při použití příkazu **RANDOMIZE** s následujícím výrazem bude řada náhodných čísel záviset na hodnotě uvedeného výrazu (tj. pro stejný výraz bude vytvářena stejná řada náhodných čísel).

Popis **DIM**

DIM název pole(výraz)

nebo

DIM název pole(výraz, výraz)

Tímto popisem se definují pole. V závorce může být uveden 1 nebo 2 výrazy, které zádají maximální hodnoty jednotlivých indexů pole. Je-li výraz pouze jeden, mluvíme o jednorozměrném poli (vektoru), při dvou výrazech o poli dvourozměrném (matici); takové pole je uspořádáno do sloupců a řádků. První výraz udává maximální číslo řádku, druhý výraz maximální číslo sloupce. V některých dialektech je možno definovat i pole s více než dvěma rozměry.

Příklady:

10 DIM V\$(15)

20 DIM M(2, 5)

V\$ je vektor tvořený 16 texty (číslovanými od 0 do 15), **M** je číselná matice se 3 řádky (číslovanými od 0 do 2) a 6 sloupcí (číslovanými od 0 do 5).

Popis **REM**

REM jakýkoli text

Popis **REM** neprovádí vůbec nic – je zcela ignorován. Jeho jediným smyslem je to, že

umožňuje zapsat do programu vysvětlující poznámky. U dobrých programátorů je zvykem, že poznámkami v programu nešetří (pokud se nedostanou do potíží kvůli omezenému rozsahu paměti).

Příklady:

```
10 REM PROGRAM PRO EVIDENCI MAGNETOFONOVYCH KAZET
20 REM COPYRIGHT (C) 1989 MLADA FRONTA
30 REM NASLEDUJE CTENI A KONTROLA VSTUPNICH DAT
...
150 REM !!!NASLEDUJICI PRIKAZ JE NUTNO
155 REM V NEKTERYCH DIALEKTECH BASIKU ZMENIT!!!
```

V některých dialektech je možno psát poznámky nejenom na samostatný řádek do popisu **REM**, ale i na konec kteréhokoli jiného řádku. Před takovou poznámkou se potom píše speciální znak – obvykle vykřičník nebo apostrof.

Příklady:

```
10 LET H0=1 ! POCATECNI VYSKA
20 FOR I=1 TO 100 'ZMENOU HORNÍ MEZE MENIME RYCHLOST
```

Popis OPTION

```
OPTION BASE 0
OPTION BASE 1
```

Pokud chceme, aby se prvky pole nečíslovaly od nuly, ale od jedné, uvedeme v programu popis **OPTION BASE 1**. Varianta **OPTION BASE 0** potvrzuje standardní číslování od nuly.

Příklad:

Doplňme-li do příkladu uvedeného výše u popisu **DIM** popis:

```
5 OPTION BASE 1
bude mít vektor V$ pouze 15 prvků a matice M bude mít 2 řádky a 5 sloupců. Všechny prvky budou číslovány od jedné.
```

V některých dialektech je možno popisem **OPTION** měnit i jiné konvence Basiku; např. **OPTION ANGLE DEGREES** udává, že úhly se pro funkce **SIN**, **COS** apod. měří ve stupních a ne v radiánech.

3. Standardní funkce

Basic obsahuje určitý počet funkcí, které jsou přímo součástí jazyka a není nutno je definovat příkazem **DEF**. Sortiment těchto *standardních* či *vestavěných* funkcí se u jednotlivých dialektů podstatně liší. Následující tabulka uvádí ty z nich, se kterými se setkáváme nejčastěji (tím není řečeno, že jsou k dispozici v každém dialektru Basiku).

Císelné argumenty se v tabulce označují X, Y, argumenty tvořené celými čísly I, J, textové argumenty A\$, B\$. Funkce, jejichž název končí znakem \$, dávají jako výsledek text, výsledkem ostatních funkcí je číslo.

Funkce pracující s čísly

ABS(X)	absolutní hodnota z X
EPS()	nejmenší číslo, pro něž ještě $1 + \text{EPS} > 1$
INF() ^{2/}	a $1 - \text{EPS} < 1$ (tzv. strojová nula) ^{1/}
	největší číslo, s nímž může počítač pracovat (tzv. strojové nekonečno)

INT(X)

LOG(X)
MAX(X, Y)
MIN(X, Y)
MOD(X, Y)
PI()^{3/}
RND()
SGN(X)
SQR(X)

největší celé číslo, které není větší než X
např. $\text{INT}(1.8) = 1$, $\text{INT}(-2.5) = -3$, $\text{INT}(5) = 5$
přirozený logaritmus X
větší z obou čísel
menší z obou čísel
zbytek po dělení X/Y
Ludolfovovo číslo π
náhodné číslo mezi 0 a 1
1, je-li X kladné; -1, je-li X záporné; 0, je-li X = 0
druhá odmocnina z X

Funkce pracující s úhly^{4/}

ACS(X)^{5/}
ASN(X)^{6/}
ATN(X)
COS(X)
SIN(X)
TAN(X)

$\arccos X$, tj. úhel, jehož kosinus je roven X
 $\arcsin X$, tj. úhel, jehož sinus je roven X
 $\arctg X$, tj. úhel, jehož tangens je roven X
kosinus úhlu X
sinus úhlu X
tangens úhlu X

Funkce s textovým argumentem a číselným výsledkem

CHR(A\$)^{7/}
LEN(A\$)
POS(A\$, B\$)

pořadové číslo znaku A\$ ve vnitřním strojovém zobrazení (A\$ musí být text skládající se z jediného znaku)
délka textu A\$, tj. počet znaků
pokud v textu A\$ je obsažen text B\$, udává výsledek, počínaje kolikátým znakem A\$ začíná text totožný s B\$;
pokud v A\$ není B\$ obsažen, je výsledek 0;
Například:
POS("ONEMOCNEL OBRNOU", "BRNO") = 12
POS("KAVAL", "A") = 2
POS("A", "B") = 0
text A\$ musí mít tvar číselné konstanty, výsledkem je potom hodnota této konstanty^{8/}

VAL(A\$)

Funkce s textovým výsledkem

CHR\$(I)
CLK\$()^{9/}
DAT\$(I)^{10/}
GAP\$(I)^{11/}
SEG\$(A\$, I, J)
STR\$(X)
SUB\$(A\$, I, J)

výsledkem je text tvořený 1 znakem, totiž I-tým znakem z kódu ASCII, případně z kódu, v němž daný počítač zobrazuje texty momentální čas; např. v poledne bude výsledek 12:00:00 momentální datum, např. 87/07/31 text tvoření I mezerami část textu A\$ tvořená I-tým až J-tým znakem včetně^{12/} hodnota výrazu X vyjádřená jako text, tj. text, který by vytiskl příkaz **PRINT(X)** Část textu A\$ tvořená J znaky počínaje I-tým znakem^{13/}

Poznámky:

- 1/ Počítáč pracuje s čísly pouze s jistou přesností (zaokrouhuje čísla na určitý počet platných číslic). Proto pro dostatečně malé X je $1 + X$ rovno 1.
- 2/ V některých dialektech se tato funkce místo INF nazývá MAXNUM.
- 3/ V některých dialektech se tato funkce místo PI nazývá CPI.
- 4/ Úhly se obvykle vyjadřují v radiánech ($180^\circ = \pi$ radiánů). V některých dialektech je však možno popisem OPTION ANGLE DEGREES určit, že se má pracovat s úhly měřenými ve stupních.
- 5/ V některých dialektech se tato funkce místo ACS nazývá ACOS.
- 6/ V některých dialektech se tato funkce místo ASN nazývá ASIN.
- 7/ V některých dialektech se tato funkce místo CHR nazývá ORD. V některých dialektech může být argumentem buď jeden znak, nebo symbolické označení speciálního znaku; např. LCA označuje malé písmeno „a“, FF přechod na novou stránku (vymazání obrazovky), BEL akustický signál (pípnutí).
- 8/ V některých dialektech může být argumentem funkce VAL i výraz; např. lze psát VAL("PI()"/2) a výsledkem této funkce je potom text 1.570785.
- 9/ V některých dialektech se tato funkce místo CLK\$ nazývá TIME\$.
- 10/ V některých dialektech se tato funkce místo DAT\$ nazývá DATE\$.
- 11/ V některých dialektech se tato funkce místo GAP\$ nazývá SPACE\$.
- 12/ V některých dialektech se část textu nevybírá pomocí standardní funkce, ale speciálním zápisem. Např. výběr 2. až 5. znaku z textu A\$ se zapisuje A\$(2:5) nebo A\$(2 TO 5).
- 13/ SUB\$(A\$, I, J) označuje totéž jako SEG\$(A\$, I, I+J-1).

4. Direktivy

Chceme-li používat Basic, musíme nejenom umět psát v něm programy, ale rovněž musíme vědět, jak ovládat jeho překladač. Tomu slouží ovládací příkazy (direktivy), které nejsou součástí programu (a proto se před ně nepíší čísla řádků). Na rozdíl od číselovaných příkazů se direktivy provádějí ihned. Sortiment a tvar těchto direktiv se v jednotlivých dialektech liší ještě více než vlastní příkazy jazyka Basic, i zde však nalezeme některé společné rysy.

Většina dialektaček pracuje (mimo jiné) s následujícími direktivami:

Direktiva NEW

Vymaže z operační paměti (zapomeně) dosavadní basikovský program i jeho proměnné, pole apod. V některých dialektech je direktiva tvořena pouze slovem NEW, v jiných se za slovo NEW píše jméno programu, který bude následovat.

Direktivy OLD a LOAD

Tyto direktivy přečtou z vnější paměti (např. kazety nebo diskety) program. V některých dialektech k tomuto slouží direktiva OLD ve tvaru:

OLD název programu

v jiných direktiva LOAD, která má tvar:

LOAD textový výraz

(Hodnota výrazu udává název programu; pokud má tento výraz nulovou délku, přečte se první program, na který se při čtení narazí.)

Direktiva RUN

Spustí provádění basikovského programu. Bývá tvořena buď pouze slovem RUN (potom spouští program, který je právě uložen v paměti počítače, tj. byl právě napsán nebo byl přečten direktivou OLD či LOAD), nebo za slovem RUN následuje název programu (v tom případě se program přečte z vnější paměti a potom spustí).

Direktiva SAVE

Zapíše program z paměti počítače do vnější paměti. Je tvořena slovem SAVE, za nímž následuje buď přímo název, pod nímž se program má do vnější paměti nahrát, nebo textový výraz, jehož hodnota udává tento název.

Direktiva LIST

Vypíše na obrazovku program momentálně uložený v paměti počítače. V případě potřeby je možno vypisovat jenom vybrané řádky z programu, např. LIST 100, 200 (v některých dialektech LIST 100 TO 200 nebo LIST 100-200) vypíše pouze řádky s čísly od 100 do 200.

Jak bylo řečeno již v první části této přílohy, je obvykle možné psát i basikovské příkazy bez čísla řádku. Tím dostanou charakter direktiv, neukládají se tedy do programu a provádějí se ihned. Naopak v řadě dialektaček je možno i direktivy opatřit číslem řádku a zapsat je tak do programu.

Intermezzo o poklesích v "Poklescích" aneb opravenka

Pravidlo, že v každém programu zůstala alespoň jedna chyba, neporušily ani programy otisklé v této knížce. Tiskařskému šotkovi se znelíbilo zejména rovnítko a nahradil nám je na str. 20 malým písmenem "z" (o řádek níže na konci navíc spolklo písmeno "N") a na str. 143 místo rovnítku zase dvakrát vpašoval slůvko "FROM".

Drobné chybíčky jako "dopověď" místo odpověď v mottu na str. 81 nebo citace literatury [KROH88] zařazená jinde, než bychom ji podle abecedy hledali, nám snad čtenář promine.

Mezi zrodem a vydáním této knížky nezahálely dějiny a napsaly populární opravenku s datem 17.11.1989. Proto bychom asi dnes na Hamletovu otázku "Kolik lidí je ochotno snášet bič a posměch doby a kopance, jež od neschopných musí strpět schopný" odpověděli v kapitole 8 optimisticky: U nás už zdaleka ne všichni. Věříme, že i v dalších vydáních bude tato oprava platit.

Příloha 2 ZNÁZORŇOVÁNÍ LOGIKY PROGRAMU

Text programu zapsaný v programovacím jazyce je určen především pro počítač. I když může být (a měl by být) formálně upraven tak, aby byl co nejsrozumitelnější i člověku, existují způsoby, jak logiku algoritmu či programu znázornit pro naše chápání přehledněji. Tyto způsoby jsou založeny hlavně na skutečnosti, že obrázek je pro člověka většinou názornější a srozumitelnější než text.

Logiku programu je možno graficky zobrazit mnoha způsoby. Historicky nejstarší jsou *vývojové diagramy* (dříve se pro ně užíval název „bloková schémata“). Jednotlivé akce, které program provádí, se zapisují do značek (bloků), jejichž geometrický tvar napovídá, o jaký druh akce se jedná. Způsob kreslení jednotlivých značek a jejich spojování do vývojového diagramu je předepsán normou ČSN 36 9030. Norma definuje celkem 30 značek; nejdůležitější z nich jsou shrnutы v obr. 13. Jakým způsobem se značky kombinují do vývojových diagramů jsme již poznali z obrázků v intermezzu *O češtině a „počítačstíne“*.

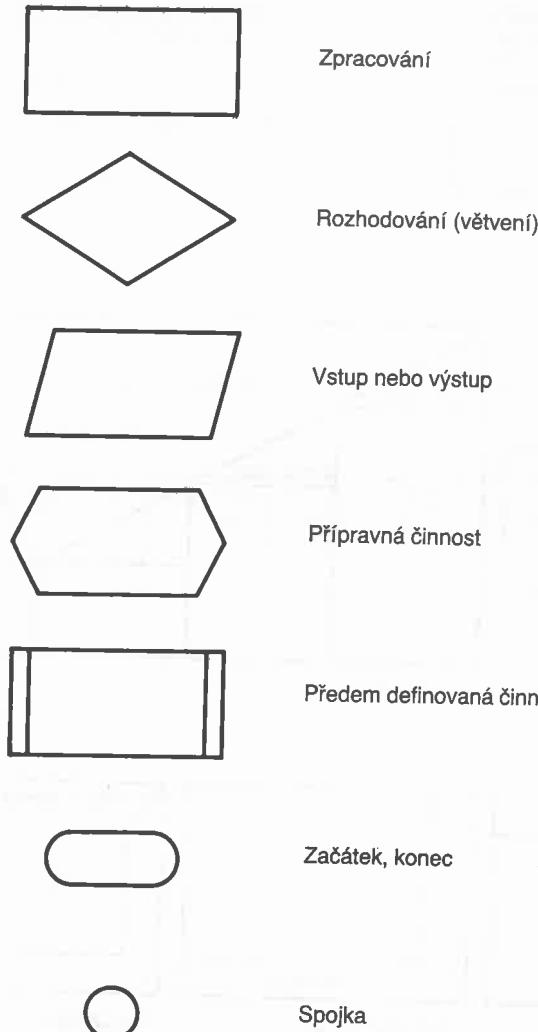
Kromě názornosti mají vývojové diagramy ještě další výhody:

- Mohou mít různou úroveň podrobnosti (např. hrubý vývojový diagram obsahuje pouze základní strukturu programu a jednotlivé akce jsou v něm popsány pouze slovně; podrobný vývojový diagram přímo odpovídá programu, používají se v něm konkrétní proměnné, matematické vzorce apod.).
- Jsou v podstatě nezávislé na programovacím jazyce, i když některé prvky podrobného vývojového diagramu (názvy proměnných, způsob zápisu některých akcí nebo terminologie) již mohou odrážet vlastnosti použitého jazyka.

Na druhé straně mají vývojové diagramy i své nevýhody:

- Nevedou k dodržování zásad strukturovaného programování, i když na druhé straně strukturovanému programování nijak nebrání. (Dobře strukturované programy lze znázorňovat vývojovými diagramy stejně dobře jako programy nestrukturované.) V tom smyslu mají vlastnosti analogické programovacím jazykům typu Basic, v nichž lze psát jak v duchu zásad strukturovaného programování, tak i v duchu zcela opačném.
- Některé prvky běžných programovacích jazyků se ve vývojových diagramech znázorňují dosti těžkopádně. Například cyklus typu **for — do** (realizovaný v Basiku dvojící příkazů FOR a NEXT, v Pascalu nebo ve Fortranu příkazem jediným) je nutno rozkreslit pomocí tří značek, nepočítaje v to samotné tělo cyklu).

Mezi modernější metody znázorňování logiky programu, vyvinuté v novější době, patří *kopenogramy* a *N-S diagramy*. (Cizokrajně znějící názvy jsou v obou případech odvozeny od jmen autorů — kopenogramy vymysleli Kofránek, Pecinovský a Novák, N-S diagramy Nassi a Schneiderman.) Základní prvky těchto diagramů ukazuje obr. 14 a 15. Jako ukázkou jejich praktického použití jsme zvolili jednoduchý program: v poli *A*, které má *n* prvků, máme najít největší prvek *max*. Současně má program do proměnné *index* zapsat, na kterém místě v poli byl tento největší prvek nalezen, a do proměnné *opak* zaznamenat, zda této maximální hodnoty dosáhl ještě některý další prvek. Obr. 16 a 17 ukazují, jak se zapíše uvedený program pomocí kopenogramu a pomocí N-S diagramu.

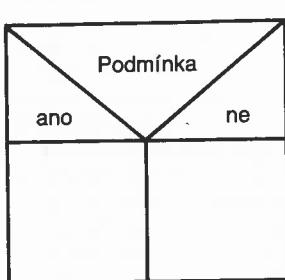


obr. 13

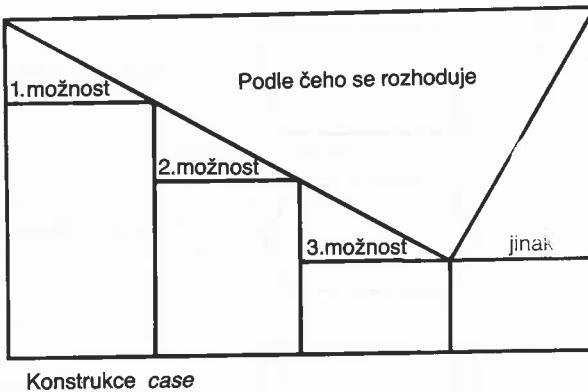
ZPRACOVÁNÍ:



ROZHODOVÁNÍ:

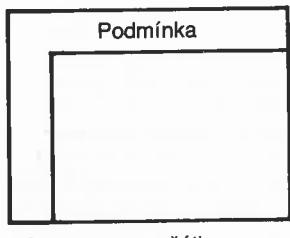


Konstrukce *if – then*
nebo *if – then – else*

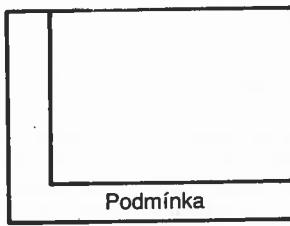


Konstrukce *case*

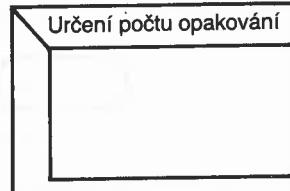
CYKLY:



S testem na začátku
(*while – do*)



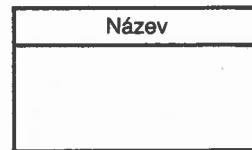
S testem na konci
(*repeat – until*)



Se zadaným počtem
opakování
(*for – do*)

obr. 14

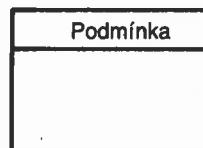
ZÁHLAVÍ KOPENOGRAMU:
(horní pruh vybarven žlutě)



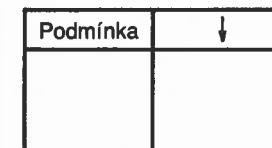
ZPRACOVÁNÍ:
(celý blok vybarven červeně;
jde-li o rekurzívní volání,
je však vybarven žlutě)



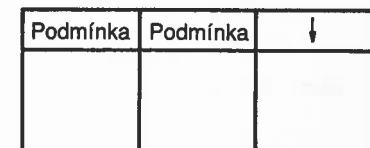
ROZHODOVÁNÍ:
(horní pruh vybarven modře)



Konstrukce
if – then

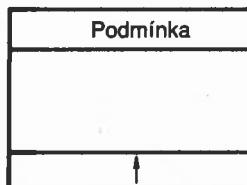


Konstrukce
if – then – else

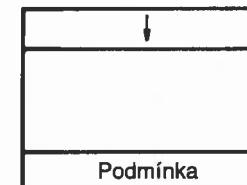


Rozhodování podle více podmínek
(např. konstrukce *case*)

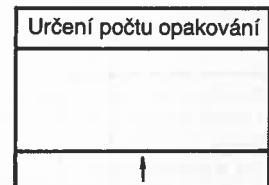
CYKLY:
(horní i dolní pruh vybarven zeleně)



S testem na začátku
(*while – do*)

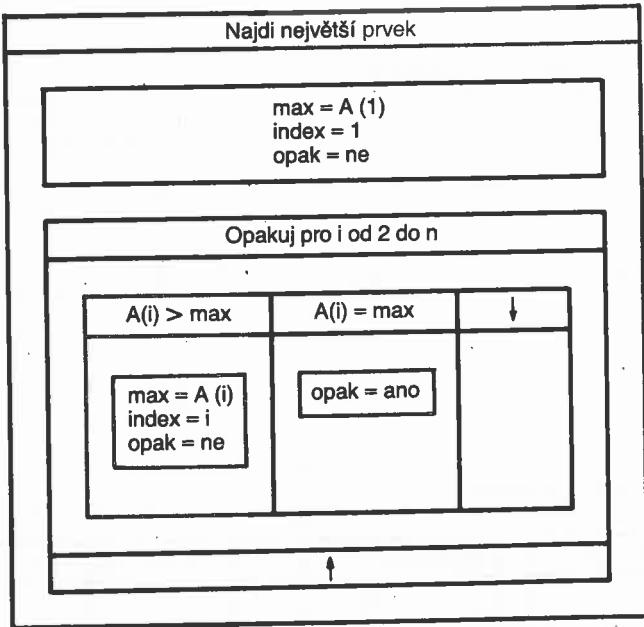


S testem na konci
(*repeat – until*)

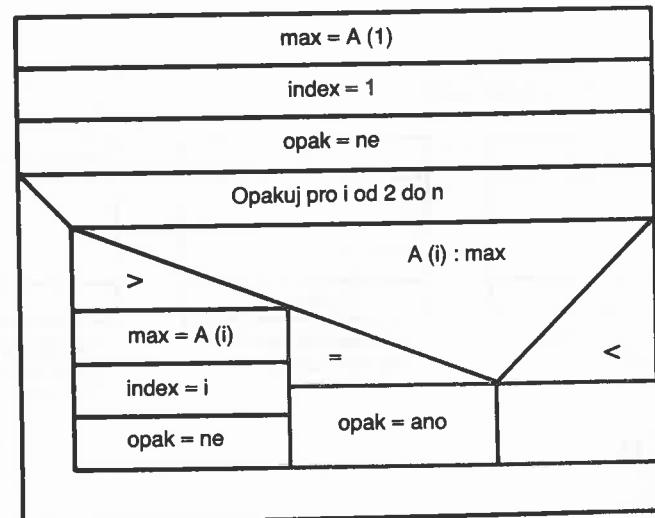


Se zadaným počtem
opakování
(*for – do*)

obr. 15



obr. 16



obr. 17

Možnosti kopenogramů a N-S diagramů jsou podobné. Podle našeho názoru jsou kopenogramy poněkud přehlednější (zejména tehdy, jestliže si dáme práci s jejich vybarvením), N-S diagramy jsou naproti tomu kompaktnější, takže dovolí na téže ploše znázornit složitější algoritmus. Kromě toho jsou kopenogramy československou specialitou (navíc používanou hlavně pro speciální výukový jazyk Karel), kdežto N-S diagramy byly publikovány v předním světovém časopise.

Poslední technikou, o níž bychom se chtěli zmínit, jsou *rozhodovací tabulky*, tvořící jakýsi přechod mezi textovým a grafickým znázorněním. Pro ilustraci uvádíme rozhodovací tabulku popisující dopravní předpisy platné při jízdě přes křižovatku řízenou světelnými signály (situaci poněkud zjednodušujeme, například nebereme v úvahu červené světlo doplněné zelenou šipkou nebo možnost vjet do křižovatky na žluté světlo).

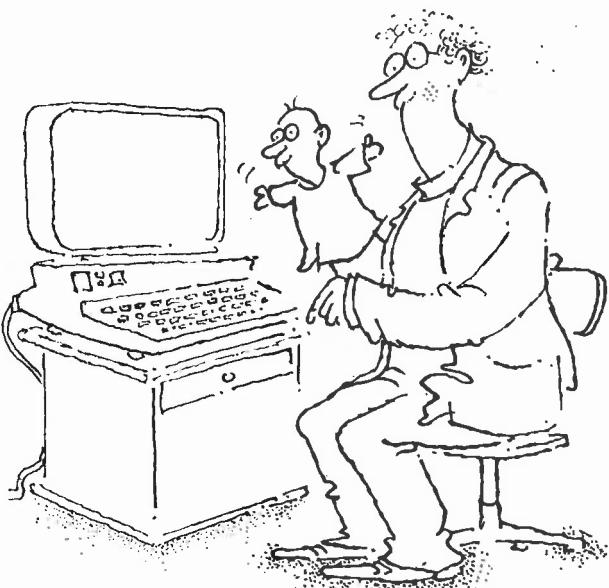
Svíti kruhové zelené světlo? Svíti zelená šipka? Jedeš rovně? Odbočuješ doprava? Odbočuješ doleva? Odbočuje souběžně vpravo tramvaj? Jede vozidlo proti?	NNNNNNNN - - AANNNNN - - NNANNNN - - NNNAAAA - - NA- NNAA - - - - NANA
Čekej na zelenou Jed. Neohroz chodce na přechodu. Dej přednost tramvaji. Dej přednost protijedoucímu.	X XX XX XX X XX X X

Jak je z příkladu zřejmé, skládá se rozhodovací tabulka ze čtyř částí (kvadrantů): výčtu podmínek, výčtu činností, kombinací podmínek (A znamená „ano“, N „ne“, -podmínce v dané kombinaci nepodstatnou) a výběru činností pro dané kombinace podmínek (X označuje činnosti, které se pro danou kombinaci mají provést). V některých případech se používají i poněkud odlišné typy rozhodovacích tabulek, zájemci najdou podrobnejší informace v [CHVA84].

Oproti dříve zmiňovaným metodám znázorňování logiky programu mají rozhodovací tabulky určité specifické vlastnosti:

- Nutí programátory, aby si promysleli všechny možnosti (kombinace podmínek), které mohou nastat.
- Lze je snadno zakódovat do tvaru textu vloženého přímo do programu. To dává možnost vytvořit překladače (preprocesory) rozhodovacích tabulek, které automaticky převедou rozhodovací tabulku do určitého programovacího jazyka. Řada takových překladačů se v praxi používá. (Programy obsahující rozhodovací tabulky se nejprve přeloží preprocesorem do některého z obvyklých programovacích jazyků a teprve výsledek tohoto přepracování se překládá obvyklým překladačem.)
- Jsou použitelné jen v určitých případech. Pro ty situace, které lze touto formou popsat, jsou však rozhodovací tabulky cenným nástrojem.

Slovniček



Tento malý výkladový slovníček obsahuje počítačové výrazy a zkratky, které souvisejí s textem knížky, a dále některé pojmy z příbuzných oborů (zejména matematiky), pokud jsou v textu použity. Nejsou do něj zahrnuty všeobecně známé pojmy (jako např. počítač nebo hardware), pokud v jejich chápání nemohou být nejasnosti.

Výklad k jednotlivým heslům si klade za cíl uvedené pojmy stručně a srozumitelně vysvětlit — nejde tedy v žádném případě o přesné a obecně platné definice. Pro snazší orientaci v anglicky psaných textech o počítačích (nebo v publikacích používajících slangové termíny) uvádíme u českých výrazů většinou též anglické ekvivalenty, u zkrátek plné znění a jeho překlad.

Některá hesla jsou vysvětlována pomocí pojmu, které jsou rovněž zahrnutы do tohoto slovníčku. Tyto pojmy jsou tištěny *kurzívou*.

Ada (název byl zvolen na počest Lady Ady Lovelaceové, která v minulém století rozpracovala metodiku řešení úloh na nikdy nedokončeném mechanickém počítači Ch. Babbage a stala se tak první programátorkou na světě) — moderní programovací jazyk, vycházející zejména z Pascalu. Mnoha odborníky je považován za hlavní programovací jazyk budoucnosti.

Adresa (address) — číselné označení místa v *operační paměti*. Adresovat lze malé paměťové celky (*bajty* nebo *slova*, nikoli však jednotlivé *bity*). Podobně jako operační paměť lze adresovat též *přídavná zařízení*, *registry*, místa na disku apod.

Algol 60 (ALGOrithmic Language, tj. algoritický jazyk) — jeden z prvních programovacích jazyků. Zavedl do programování mnoho nových prvků a ovlivnil tak další programovací jazyky.

Algol 68 — následovník *Algolu 60*. Obsahuje řadu velmi moderních prvků, ale příliš se nerozšířil. Jeho hlavní význam spočívá v tom, že podobně jako Algol 60 výrazně ovlivnil jiné programovací jazyky.

Algoritmus (algorithm) — přesně formulovaný postup řešení určité skupiny úloh (např. algoritmus pro sčítání zlomků nebo algoritmus pro dělení slov v češtině).

Assembler — 1. (angl. assembly language) jazyk blízký strojovému kódu počítače; jednotlivé instrukce tohoto jazyka obvykle přímo odpovídají strojovým instrukcím, jsou však kódovány způsobem pro člověka srozumitelnějším.

2. (angl. assembler) překladač z výše zmíněného jazyka do strojového kódu.

Bajt (byte; česky též slabika) — nejmenší adresovatelná jednotka paměti. Skládá se z 8 bitů, takže jeho obsah může nabývat 2^8 , tj. 256 různých hodnot. V jednom bajtu může být uložen jeden textový znak; pro uložení číselné hodnoty je zpravidla zapotřebí několika bajtů.

Basic (Beginner's All-purpose Symbolic Instruction Code, tj. univerzální jazyk symbolických instrukcí pro začátečníky) — jednoduchý programovací jazyk, velmi rozšířený zejména na *mikropočítačích*.

Bit (vyslov [bit]; zkratka z Binary digit, tj. dvojková číslice) — nejmenší jednotka paměti, do níž lze uložit jednu dvojkovou číslici (tj. 0 nebo 1).

C — moderní jazyk vhodný zejména pro profesionální programátory.

Cílový modul (object module) — výsledek práce *kompilátoru*, tj. program v přeloženém tvaru (blízkém strojovému kódu).

Cobol (COmmon Business Oriented Language, tj. obecný obchodně orientovaný jazyk) — nejrozšířenější jazyk pro programování úloh z oblasti hromadného zpracování dat.

Computer science viz *Matematická informatika*.

Cyklus nekonečný viz *Nekonečný cyklus*.

Cíp (chip) — integrovaný obvod vytvořený na jednom krystalu polovodiče.

Databanka (data bank) — způsob uložení (obvykle velkého množství) dat, zajišťující snadný přístup různým uživatelům. Tatáž data se přitom podle potřeby mohou jevit každému uživateli jinak (logický tvar dat se liší od jejich fyzické podoby). Databanka se skládá z báze dat (vlastní data) a ze systému řízení báze dat (programy pro práci s daty a pro obsluhu databanky).

Datová struktura (data structure) — seskupení datových položek (čísel, textů) do většího objektu, s nímž lze pracovat buď jako s celkem nebo individuálně po položkách. Datovou strukturu je např. *záznam* nebo *pole*.

Editor — program pro provádění změn v textových souborech (např. ve *zdrojových tvarech programů*). Jednoduché editory zobrazují opravovaný text postupně na obrazovku a dovolují do něj provádět zásahy, složitější editory jsou řízeny příkazy speciálního editovacího jazyka a dovolují provádět v textu systematické změny.

EPROM viz ROM.

Expertní systém — programový systém, který dovoluje ukládat do počítače vhodným způsobem zakódované znalosti a zkušenosti odborníků a používat jich k řešení složitých problémů. Skládá se z báze znalostí a obslužných programů, z nichž nejdůležitější realizuje tzv. inferenční mechanismus (ten umožňuje ze zapsaných znalostí vyvozovat závěry). Expertní systémy se uplatňují v medicíně, geologickém průzkumu, právnictví aj.

Fortran (FORmula TRANslator, tj. překladač vzorců) — nejstarší programovací jazyk vyšší úrovně. Je velmi rozšířen v oblasti vědeckotechnických výpočtů; jsou v něm zapsány rozsáhlé knihovny programů.

Fortran 77 — novější verze jazyka Fortran, odstraňující řadu nedostatků, jimiž trpěly starší verze.

Hornerovo schéma — algoritmus pro rychlý a jednoduchý výpočet hodnoty polynomu, při němž se nemusí výslově počítat jednotlivé mocniny nezávisle proměnné. Např.

$$5x^3 + 2x^2 - x + 2$$

se počítá podle matematicky ekvivalentního výrazu

$$((5 \cdot x + 2) \cdot x - 1) \cdot x + 2.$$

Implementace programu (program implementation) — realizace programu (například překladače) v určitém výpočetním prostředí, tj. na určitém počítači a pod určitým operačním systémem.

Indexsekvenční organizace (index-sequential organization) — způsob organizace souborů na magnetických discích. Každý záznam má svůj klíč, který jej jednoznačně identifikuje (např. při zpracování vkladu je klíčem číslo vkladní knížky, při evidenci motorových vozidel jím může být státní poznávací značka) a záznamy jsou uloženy v pořadí daném hodnotou tohoto klíče. U takto organizovaného souboru lze jednotlivé záznamy číst a zapisovat jak sekvenčně (postupně), tak i „na přeskáčku“.

Interpretační překladač (interpreter) — překladač, který analyzuje postupně jednotlivé příkazy ze zdrojového tvaru programu a ihned je provádí. Tímto způsobem pracuje většina překladačů Basiku.

Jazyk pro řízení úloh (Job Control Language, JCL) — speciální jazyk, pomocí něhož se operačnímu systému zadávají příkazy řídící zpracování jednotlivých úloh. Na rozdíl od programovacích jazyků, které jsou většinou strojově nezávislé, má každý operační systém svůj vlastní jazyk pro řízení úloh.

JCL viz jazyk pro řízení úloh.

JSEP — jednotný systém elektronických počítačů. Řada navzájem kompatibilních počítačů, vyráběná v zemích RVHP. Jednotlivé počítače této řady (i jejich přídavná zařízení) se označují symbolem EC nebo ES, za nímž následuje čtyřmístné číslo. Počítače JSEP jsou programově kompatibilní též s počítači IBM/360, IBM/370 a příbuznými typy.

Knihovna (library; partitioned data set) — skupina souborů; zpravidla jde o soubory s podobnými vlastnostmi, uložené na též disku. Velmi často jsou do knihoven ukládány zdrojové tvary programů a člověkové moduly; viz modulová knihovna a zdrojová knihovna.

Kompatibilita (compatibility; česky též slučitelnost) — míra podobnosti počítače s jiným počítačem, umožňující vzájemnost nebo spolupráci. Rozeznáváme kompatibilitu datovou (zaměnitelnost záznamových médií — např. disket), programovou (použitelnost programů přenesených z jiného počítače na úrovni zdrojového tvaru nebo člověkového modulu) a technickou (zaměnitelnost jednotlivých částí počítače).

Kompilátor — překladač, který na základě analýzy zdrojového tvaru programu vytváří člověkový modul ve strojovém kódu počítače (případně v jiném jazyce).

Matematická informatika (computer science) — teoretický základ výpočetní techniky a programování. Patří do ní např. teorie formálních jazyků a automatů, teorie algoritmů aj.

Mikropočítač (microcomputer) — počítač, jehož procesorem je mikroprocesor. Někdy je celý mikropočítač (tj. mikroprocesor, paměť a další podpůrné obvody) tvořen jediným integrovaným obvodem (čipem) a mluvíme potom o „jednočipovém mikropočítači“.

Mikroprocesor (microprocessor) — procesor tvořený jediným integrovaným obvodem. Podle počtu současně zpracovávaných bitů rozeznáváme mikroprocesory 8bitové, 16bitové a 32bitové.

Minipočítač (minicomputer) — počítač, který cenou a rozměry (původně i výkonem) stojí mezi klasickými střediskovými počítači a osobními počítači.

Modulová knihovna (object module library) — knihovna, do níž se ukládají člověkové moduly.

Multiprogramování (multiprogramming) — současné zpracování několika úloh na témže procesoru (čas procesoru se přiděluje střídavě jednotlivým úlohám). To umožňuje využít čas, kdy některá úloha čeká (např. na provedení operace s přídavným zařízením), pro řešení jiné úlohy.

Naformátování disku (disc formatting) — zápis služebních informací na disk. U mnoha typů disků musí být disk před použitím naformátován.

Nekonečný cyklus viz Cyklus nekonečný.

Numerická nestabilita — citlivost výsledků výpočtu na drobné nepřesnosti způsobené zaokrouhlováním. Může být způsobena povahou problému (nestabilní problém) nebo použitím nevhodného algoritmu či jeho nevhodným naprogramováním (nestabilní metoda).

Operační paměť (main store) — paměť přímo dostupná procesoru. V současné době je realizována obvykle jako paměť RAM s integrovanými obvody velké integrace. Může být doplněna též pamětí ROM.

Operační systém (operating system) — soubor programů, které řídí činnost počítače. U moderního počítače tvoří jeho nezbytný doplněk.

Operand — veličina (např. konstanta nebo proměnná), s níž se provádí matematická nebo jiná operace.

Operátor (operator) — 1. Znak nebo skupina znaků udávající, jaká (matematická či jiná) operace se má provést s operandy. Např. / (lomítko) je operátor dělení. Každý operátor působí na určitý počet operandů (zmíněný operátor dělení má dva operandy). 2. Pracovník výpočetního střediska obsluhující počítač.

Osobní počítač (personal computer; PC) — výkonný mikropočítač určený pro profesionální použití. Většina osobních počítačů je kompatibilní s osobními počítači firmy IBM. Někteří výrobci označují z reklamních důvodů za osobní počítače i počítače určené pro použití v domácnostech; ty se však častěji označují jako „domácí počítače“ nebo „hobby-počítače“.

Parametr (parameter; argument) — prostředek, který umožňuje pracovat v podprogramu s daty volajícího programu. V záhlaví podprogramu se uvádějí formální parametry, které se jistým způsobem ztotožní se skutečnými parametry uvedenými v příkaze, jímž se podprogram vyvolává.

Pascal (název byl zvolen na počest francouzského matematika Blaise Pascala, který zkonztruoval první mechanickou kalkulačku) — moderní programovací jazyk vhodný zejména pro výuku základů programování. Obsahuje konstrukce účinně podporující strukturované programování, má však též nedostatky, které omezují jeho použitelnost v běžné praxi.

Podtečení (underflow) — stav, který nastane, když je výsledek aritmetické operace s reálnými čísly v absolutní hodnotě tak malý, že ho nelze v počítači zobrazit. U mnoha počítačů se v tomto případě výsledek nahradí nulou, u některých se však tento stav považuje za chybu.

Polynom (polynomial; česky též mnohočlen) — součet mocnin určité proměnné, násobených určitými koeficienty. Např. $5x^2 - 3x + 2$ je polynom; protože obsahuje nejvýše druhou mocninu x , nazývá se polynomem druhého stupně.

Portabilita (portability, česky též přenositelnost) — vlastnost programu vyjadřující snadnost jeho převedení na jiný počítač. Má vztah k programové kompatibilitě, ta je však vlastnosti počítače a nikoli programu. Programy by měly být psány tak, aby byly portabilní; v některých případech se však tento požadavek dostává do rozporu s efektivností.

Procesor — základní část počítače, dekódující a provádějící strojové instrukce.

Prolog — programovací jazyk vysoké úrovně zaměřený na řešení logických problémů. Je jedním z moderních jazyků podporujících tzv. umělou inteligenci; hráje proto významnou úlohu v připravované 5. generaci počítačů.

PROM viz ROM.

Přetečení (overflow) — stav, který nastane, je-li absolutní hodnota výsledku aritmetické operace větší než největší číslo, které lze v počítači zobrazit. Zpravidla má za následek přerušení výpočtu.

Přídavné zařízení (peripheral device) — jednotka připojená k počítači a sloužící jako vstupní či výstupní zařízení nebo jako vnější paměť (např. disketová jednotka, tiskárna, terminál).

RAM (Random Access Memory, tj. paměť s libovolným přístupem; též RWM = Read/Write Memory, tj. paměť pro čtení i zápis) — adresovatelná paměť, z níž lze číst i do ní zapisovat; vypnutím počítače se její obsah obvykle vymaže. Musí být součástí každého počítače; kromě ní může mít počítač i jiné druhy pamětí (např. ROM a vnější paměti).

Registr (register) — rychlá paměť o malé kapacitě (např. 1 slovo). Součástí procesoru jsou specializované registry (např. strádač, indexregister, registr masek přerušení), popř. univerzální registry, které mohou plnit více funkcí.

ROM (Read Only Memory, tj. paměť jen pro čtení) — paměť, z níž lze číst, ale nelze do ní zapisovat; její obsah je určen již při výrobě. Pokud je do ní obsah zapsán až dodatečně po jejím vyrobění (zvláštním zařízením), označujeme takovou paměť jako PROM (Programmable ROM, tj. programovatelná ROM); pokud lze tento obsah dodatečně vymazat (např. ultrafialovým zářením) a přepsat, mluvíme o paměti EPROM (Erasable PROM, tj. vymazatelná PROM). U mikropočítačů bývá v paměti ROM uložena základní část operačního systému, popř. překladač basiku.

Sekvenční organizace (serial organization) — organizace souboru, při níž je možno do souboru zapisovat a číst z něho pouze postupně. Na většině přídavných zařízení je to jediná možná organizace, pouze na discích (a některých podobných zařízeních) jsou možné i jiné organizace (např. indexsekvenční).

Slovo (word) — elementární část paměti větší než bajt. Na počítačích s pamětí organizovanou po bajtech je slovo tvořeno obvykle 2 nebo 4 bajty, na některých počítačích je paměť organizována přímo po slovech a slovo je nejmenší adresovatelnou paměťovou jednotkou.

SMEP — systém malých elektronických počítačů. Řada minipočítačů a mikropočítačů vyráběná v některých státech RVHP. Většinou se jedná o počítače kompatibilní s počítači firmy DEC (PDP 11, VAX) nebo s IBM PC.

Soubor (file) — ucelená množina dat zapsaná na vnějším médiu a skládající se ze

záznamů. Soubory lze seskupovat do knihoven. Souborem může být např. zdrojový tvar programu, týž program přeložený do cílového modulu, balíček děrných štítků obsahujících vstupní data pro jedno zpracování programu, seznam pracovníků podniku s jejich osobními daty apod.

Strukturované programování (structured programming) — metodika tvorby programů zajišťující jejich přehlednost, čitelnost a snižující počet chyb. Strukturované programy se snáze ladí, testují a upravují.

Terminál — přídavné zařízení, jehož pomocí může uživatel zadávat počítači data nebo i řídicí příkazy. Obvykle je tvořen klávesnicí a obrazovkou; potom umožňuje dialogovou (konverzační) práci. Jako levnější konverzační terminál může sloužit i dálkový terminál mohou být připojeny k počítači jako místní (kabelem) nebo dálkové (telefonní linkou nebo prostřednictvím jiné přenosové cesty) — v tom případě mohou být od počítače značně vzdáleny. Existují i dálkové terminály, které nepracují konverzačně a jejich posíláním je dálkové zadávání úloh ke zpracování.

Virtuální paměť (virtual store) — paměť, s níž může program pracovat, jako kdyby šlo o operační paměť, i když fyzicky není k dispozici. Pokud je počítač vybaven virtuální pamětí, udržuje v operační paměti pouze ta data, která jsou ke zpracování bezprostředně zapotřebí, a zbývající část virtuální paměti je fyzicky umístěna v rychlé vnější paměti (na disku). V případě potřeby se data, na něž se program obrátí, velmi rychle přesunou z disku do operační paměti. Tento přesun provede operační systém počítače automaticky. To umožňuje zpracovávat úlohy, na které nestačí kapacita operační paměti, bez nutnosti organizovat v programu využívání vnějších pamětí.

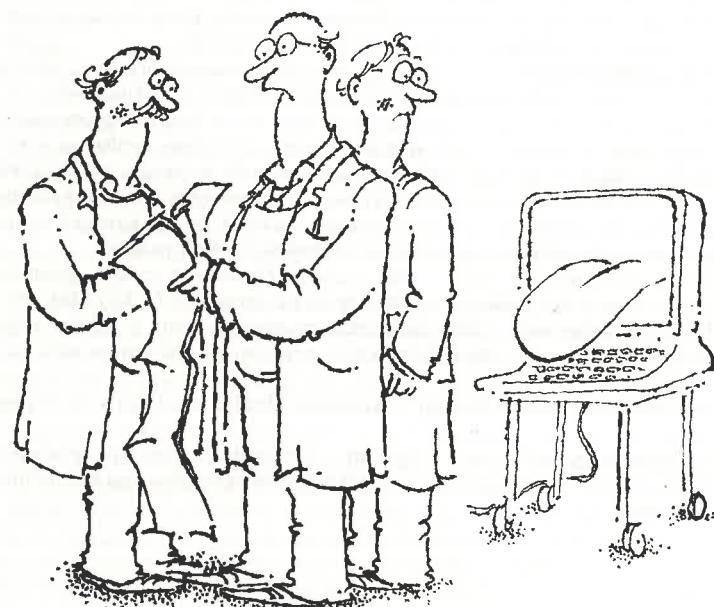
Záznam (record; česky též věta) — část souboru. Záznamem je např. jeden řádek programu, jeden děrný štítek nebo jeden řádek textu na obrazovce (někdy však jím může být i celý obsah obrazovky). Velmi často záznam obsahuje data o jednom objektu (při evidenci knih tvoří záznam data o jedné knize, v osobní agendě data o jednom pracovníkovi).

Zdrojová knihovna (source library) — knihovna obsahující zdrojové tvary programů.

Zdrojový tvar programu (source program) — program zapsaný v programovacím jazyce. Může být zpracován interpretacním překladačem nebo přeložen kompilátorem do cílového modulu.

Literatura

Druhá nejlepší věc po znalosti něčeho je vědět, kde informaci o znalosti najít.
S. Johnson



[BENE79] M. Benešovský, J. Hořejš: Počítače a zločin aneb k otázkám ochrany počítačů, programů a dat. SOFSEM '79, Zborník referátov, Labská bouda 1979.

Příspěvek ve sborníku známého softwarového semináře si všímá otázek souvisejících s počítačem v boji proti zločinu, s počítačem jako nástrojem potenciálního zločinu a s ochranou počítače a dat. Větší část příspěvku může být srozumitelná i začátečníkovi, část vyžaduje znalost základů programování a vyšší matematiky.

[BROW81] A.R. Brown, W.A. Sampson: Ledenie programov. Alfa, Bratislava 1981. Kniha vznikla na základě kursu ladění programů. Autoři sami ji charakterizují takto: „Účelem knihy je hlavně poradit, jak předcházet chybám, a když se vyskytnou (čemuž se nedá zabránit), jak je co nejrychleji najít a odstranit.“ Kniha je psána srozumitelně a vyžaduje jen nejzákladnější znalosti o počítačích a programování.

[DIJK76] E. W. Dijkstra: A discipline of programming. Prentice-Hall, Englewood Cliffs 1976.

Autor je jedním z nejvýznamnějších odborníků v matematické informatice. Kniha se zabývá základními otázkami konstrukce programů a jejich korektnosti; patří mezi klasická díla matematické informatiky. Látka je vysvětlována většinou na řešení konkrétních problémů. Doporučujeme všem čtenářům se znalostmi základů matematické informatiky. V ruštině vyšla tato knižka v roce 1978 pod názvem „Disciplina programowania“.

[FRAN87] J. Franěk, R. Kryl, P. Štěpánek, O. Štěpánková: Domluvte se s počítačem. Magazín VTM 1987/2, Mladá fronta, Praha 1987.

V této sympathetické brožurce najdete stručně, přehledně a srozumitelně popsány základy jazyků Basic, Pascal a Prolog. U čtenáře se předpokládá pouze chuť do programování.

[HONZ87] J. Honzík a kolektiv: Programovací techniky. VUT Brno, 1987.

Skripta určená pro třetí ročník studia oboru elektronické počítače na elektrotechnické fakultě VUT v Brně. Obsahuje nejmodernější poznatky disciplíny programování. Látka je vykládána s důrazem na srozumitelnost, je však poměrně náročná a předpokládá znalost základů matematické informatiky.

[HOPC78] J. E. Hopcroft, J. D. Ullman: Formálne jazyky a automaty. Alfa, Bratislava 1978.

Publikace obsahuje základy matematické teorie formálních jazyků a automatů. Teorie formálních jazyků a automatů patří k matematickým základům informatiky. K praktickému programování není znalost této teorie nutná, dává nám však k dispozici zajímavé matematické modely programovacích jazyků a na strukturách automatů přiblížuje i některé vlastnosti počítačů a programů. Knižka obsahuje také základní výsledky z teorie výčíslitelnosti, např. zajímavá tvrzení o rozhodnutelnosti problémů, která mají přinejmenším filozofický význam pro programování. Knižka vyžaduje jistou matematickou záběhlost a znalost matematiky asi na úrovni prvního až druhého ročníku vysokých škol.

[HOŘE80] J. Hořejš, J. Brodský, J. Staudek: Struktura počítačů a jejich programového vybavení. SNTL, ALFA, Praha 1980.

Systematický úvod do studia programového vybavení počítačů. V knize se čtenář seznámí s podstatnými rysy počítačů a v návaznosti na to se strukturou programů, překladačů a operačních systémů. Výklad je ilustrován na systému JSEP (viz slovníček). Hovoří se zde rovněž o některých konkrétních jazycích: je popsán strojový kód a assembler JSEP, vyšší jazyk typu PASCAL i jazyk JCL (viz slovníček) pro řešení úloh v rámci operačního systému OS/MVT JSEP. Od čtenáře se předpokládá základní rozhled po problematice programování a provozu počítače.

[HOŘE82] J. Hořejš: Analýza a syntéza programů. Sborník semináře „Moderní programování“, díl 2, str. 7 až 69. Pezinok 1982.

Příspěvek se zabývá obecnými pohledy na vývoj programů, jejich analýzu, syntézu a testování. Problematika je velmi přehledně zpracována – pro dobré pochopení však vyžaduje znalost základů matematické informatiky.

[HŘEB89] J. Hřebíček, I. Kopeček, J. Kučera, P. Polcar: Programovací jazyk Fortran 77 a vědeckotechnické výpočty. Academia, Praha 1989.

Kniha je učebnicí a detailním popisem jazyka Fortran 77. Obsahuje kapitolu určenou pro začátečníky, kde se mimořádně dbá na srozumitelnost textu a jednoduchost výkladu a kde se čtenář seznámí se základními příkazy Fortranu 77. Kniha kromě toho obsahuje detailní popis jazyka, který svým rozsahem odpovídá jeho normě, příklady programů, kapitolu týkající se programování vědeckotechnických problémů a řadu příloh. Je vhodná jak pro začínající, tak i pro pokročilé programátory.

[CHVA84] V. Chvalovský: Rozhodovací tabulky. SNTL, Praha 1984.

Podrobný popis různých typů rozhodovacích tabulek a jejich použití.

[KNUT68] E. Knuth: The art of computer programming; Volume 1, Fundamental Algorithms. Addison—Wesley Publishing Company, Ontario 1968.

První díl jedné z nejobjemnějších klasických knih o programování (kniha má mít celkem sedm dílů). Autor je jeden z nejvýznamnějších amerických specialistů na systémové programování. První díl této knihy se detailně zabývá problémy algoritmů a informačních struktur. Kniha je dosti náročná, některá místá vyžadují znalost základů vysší matematiky a určitý přehled o problematice programování. V roce 1976 byla vydána v roce 1976 pod názvem „Iskusstvo programirovanija dlja EVM“.

[LEAC71] S. Leacock: Literární poklesky. Mladá fronta, Praha 1971.

Aby byla patřičně vychutnána, vyžaduje tato slavná knížka u čtenáře především inteligenci a smysl pro humor. Těmito vlastnostmi se vyznačuje drtivá většina programátorů — je tedy právě pro ně ideálním rozptýlením. Pro přetileté programátory jsou však vhodnější pohádky Boženy Němcové nebo Einsteinova teorie relativity.

[KROH88] P. Kroha, P. Slavík: Basic pro začátečníky. SNTL, 1988.

Učebnice a popis nejrozšířenějšího mikropočítáčového jazyka. Při výkladu se dbá na srozumitelnost, text je doplněn řadou příkladů. Doporučujeme všem zájemcům o Basic.

✓ [MCNE83] J. McNeil: Poradce. Svoboda, Praha 1983.

Napínavý thriller o počítáčovém zločinu, ve kterém se o programování můžete dovédat možná víc, než z některých naučných pojednání. Hlavní hrdina však může sloužit jako vzor pouze z hlediska odborné stránky své činnosti.

✓ [MIKL79] J. Mikloško: Syntéza a analýza efektívnych numerických algoritmov. Veda, Bratislava 1979.

Tématem knihy jsou některé současné problémy syntézy a analýzy numerických algoritmů. Výklad klade do popředí algoritmické a implementační hledisko. Látka je ilustrována řadou numerických experimentů realizovaných na počítači. Studium předpokládá znalost základů vysší matematiky a programování.

[MOLN87] L. Molnár: Programovanie v jazyku Pascal. Alfa, SNTL, Bratislava/Praha, 1987.

Učebnice a popis jednoho z nejrozšířenějších programovacích jazyků. Text je doplněn řadou příkladů a cvičeními. V přílohách obsahuje rovněž formální syntax jazyka a jeho syntaktické diagramy.

[MYER79] G. J. Myers: The art of software testing. J. Wiley and sons, 1979.

Jedna ze základních monografií o testování programů. Kniha je psána velmi srozumitelně. Je v ní rozebrána metodologie testování a ladění programů i otázky psychologie a ekonomiky testování. V roce 1982 vyšla kniha pod názvem „Iskusstvo testirovaniija programov“.

[NEŠE79] J. Nešetřil: Teorie grafů. Matematický seminář SNTL, Praha 1979.

Monografie je věnována základům teorie grafů. Je psána matematickou formou, vyžaduje středoškolské znalosti matematiky a určitou záběhlost v matematickém uvažování. Obsahuje popis řady velmi zajímavých algoritmů na řešení problémů formulovatelných v teorii grafů.

[PILE79] S. Pile: The book of heroic failures. Futura publications, London 1979.

Autor této úspěšné knížky je zakládajícím členem „Klubu ne právě úspěšných mužů Velké Británie“. K získání členství je zapotřebí být v nějakém oboru „ne právě úspěšný, pokud možno totálně neschopný“. Knižka obsahuje přihlášku do klubu (zájemce musíme zklamat, u nás podobný klub zřejmě nebude založen z organizačních důvodů). Knižka vyžaduje dobrou znalost angličtiny, smysl pro humor a pro nadsázkou. Nic z toho by programátorem nemělo chybět.

✓ [SOKO77] J. Sokol: Jak počítá počítač. SNTL, Praha 1977.

Knižka názorně a srozumitelně seznamuje čtenáře se základními principy práce počítačů a rovněž ho uvádí do základů programování. Doporučujeme všem, kteří se zajímají o počítače a nemají potřebné základní vědomosti.

[WIRT88] N. Wirth: Algoritmy a štruktury údajov. Alfa, Bratislava 1988.

Kniha významného odborníka v oblasti programování a informatiky, patřící ke klasice literatury tohoto druhu. Autor fundovaně a na vysoké metodické úrovni vysvětuje pojmy, se kterými se setká každý, kdo chce důkladně pochopit principy moderního programování. Příklady uváděné v knize jsou programovány v jazyku Pascal. Výklad je srozumitelný a jasný, pro dobré pochopení však vyžaduje určitou matematickou záběhlost (asi v rozsahu znalostí základů vysší matematiky) a základní přehled o počítačích a programování. Originál knihy vyšel v roce 1975 pod názvem „Algorithms + Data Structures = Programs“.

Ivan Kopeček

Jan Kučera

PROGRAMÁTOŘSKÉ POKLESKY



Odbornou revizi provedl ing. Petr Adámek. Odborně lektoroval doc. RNDr. Jiří Hořejš, CSc. Ilustrovali Vladimír Renčín, Jaroslav Turek a Jan Kučera. Obálku navrhl Miroslav Pechánek. Graficky upravili Jana Vysoká a Jan Kučera. Vydala Mladá fronta jako svou 5 110. publikaci. Odpo- vědná redaktorka Božena Fleissigová. Výtvarný redaktor Josef Velčovský. Technická redaktorka Jana Vysoká. Na základě rukopisu dodaného autory na magnetické pásmo vysadila Svoboda, n. p., závod 4, Karlovo nám. 15, Praha 2. Vytiskl Mír, novinářské závody, sdr. p., závod 1, Václavské n. 15, Praha 1. 11,29 AA. 13,78 VA. 168 stran. Náklad 36 000 výtisků. 510/21/82.6 Vydaní 1. Praha 1989. 23-081-89 03/2 Cena brož. 15 Kčs

MLADÁ F.

program

Star

A73714

23-081-89 03/2

Cena brož. výt.

15 Kčs

V době, kdy hromadně sdělovací prostředky představují programování jako procházkou růžovou zahradou a na druhé straně vycházejí (byť poskrovnu) pro laickou veřejnost těžko stravitelné a specializované učebnice programovacích jazyků, je posuzované dílko vítaným přínosem. Ač dělá na první pohled dojem lehkého humoristického čtení, představuje ve skutečnosti dosti rozsáhlý soubor cenných informací ...

Po odborné stránce lze knížku hodnotit jako fundovaného průvodce úskalimi, která jsou specializovanými učebnicemi notoricky opomíjena. Humorně laděný ton nikterak nesnižuje význam výkladu chyb a omylů, jejichž repertoár je vybrán skutečně reprezentativně.

S potěšením jsem si přečetl rukopis, v němž si zkušení programátoři otestovali své literární vlohy, a jsem si jist, že podobným způsobem se rádi pobavi a poučí i další čtenáři.

Publikace je svým způsobem unikátem v tom smyslu, že podává souhrnný přehled chyb a nevhodných představ, tak jak je přinášel a přináší programátorský život a jež byly často draze vykoupeny.

(z lektorských posudků)

Knížka je určena širokému okruhu čtenářů, kteří se zajímají o programování a počítače, ale nepostradatelná se zdá být zejména pro šťastné majitele vlastních mikropočítačů, fanoušky z řad studentů středních a vysokých škol a začínající programátoři. Řadu cenných informací zde jistě najezou i mnozí profesionální programátoři a jejich vlastní programátorské zážitky jim umožní plně vychutnat nekonvenční způsob výkladu, v němž nadsázka a humor tvoří kontrast k vlastnímu odbornému obsahu knížky.

O čtenáři se předpokládá, že má základní všeobecné představy o počítačích a programování. Ukázky programů a jejich části jsou psány v jazyce Basic. Tomu, kdo tento jazyk ovládá jen málo nebo ho neovládá vůbec, je určena příloha, v níž jsou základy tohoto programovacího jazyka stručně a srozumitelně popsány.

Knížka, která slibuje všech pět pé — příjemné počtení, pobavení a poučení o programování!!!