Ivan Kopeček Jan Kučera

PROGRAMÁTORSKÉ POKLESKY

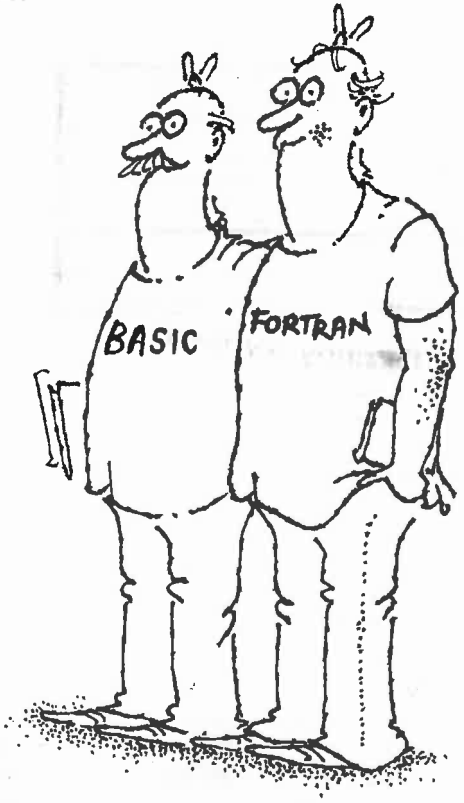
MLADÁ FRONTA

program start

Věnováno všem,
kteří udělali anebo hodlají udělat
alespoň jednu programátorskou chybu.

Ivan Kopeček Jan Kučera

PROGRAMÁTORSKÉ POKLESKY



Odborně lektorovali
ing. Petr Adámek a RNDr. Jiří Hořejš, CSc.

Programování – přívěs
Mikropočítače – přívěs

ISBN 80-204-0068-0

Obsah

Předmluva

Odborník je člověk, který se už dopustil všech obvyklých chyb v určitém oboru.
Niels Bohr

Pár tisíc let poté, co naši prapředci vypěstovali z lesní šelmy milou kočičku s růžovou mašličkou (kočku domácí) a z vyjícího vlka přívětivého jezevčíka (psa domácího), se v domácnostech objevuje nový tvor: počítač domácí, zvaný též mikropočítač. Třebaže se ani v nejmenším nechceme dotknout ušlechtilé záliby v chovu psů a koček, nemůžeme přehlédnout určité přednosti mikropočítačů:
— nekoušou, neškrábou a neštěkají;
— nedělají hromádky a loužičky;
— s dětmi i s dospělými si ochotně zahrají nejrůznější hry;
— na rozdíl od ostatních členů domácnosti se dají kdykoli vypnout;
— můžeme je chovat bez jakýchkoli problémů a třenic se sousedy i ve specifických podmínkách paneláků.

Není proto divu, že mezi nejširší veřejností, zejména mezi mladší částí populace, zájem o mikropočítače stále vzrůstá. Avšak mikropočítače by patrně zůstaly pouze superdokonalými hračkami, nebýt jejich nejpodstatnější vlastnosti: dají se programovat. A to má dalekosáhlé důsledky — programováním se totiž učíme myslet. A nejen to, poznáváme také lépe sami sebe jako tvora nedokonalého a vytrvale chybujícího. O vlivu počítačů na člověka a o významu komputerizace společnosti jsou však již napsány celé knihy a my zde nemíníme toto téma rozebírat. Připomeňme si pouze výrok jednoho na slovo vzatého odborníka:

„Mladí lidé se dnes dělí na dvě skupiny: Jedni tráví svůj volný čas u hracích automatů a do omrzení hrají počítačové hry, druzí se baví programováním. Za deset patnáct let budou ti druzí šéfy těch prvních."

I u nás je již velký, možná ohromující počet (přesná čísla nikdo nezná) počítačových fanoušků a nadšenců, kteří se pokoušejí programovat, a každým dnem jich přibývá. Zvláště jim je určeno naše neformální pojednání o programátorských chybách a o tom, jak jim předcházet. Dalším potenciálním okruhem čtenářů jsou studenti středních a vysokých škol — pro mnohé z nich jsou počítače stejně přitažlivé jako diskotéky a jiné radosti studentského života.

Věříme však, že knížka může být zajímavá a užitečná i pro mnohé profesionální programátory. Proto se také nevyhýbáme příkladům převzatým z prostředí výpočetních středisek a klasických počítačů (ostatně vývoj jde tak rychle kupředu, že u moderních mikropočítačů se setkáváme s mnohým, co

bylo donedávna spojováno pouze s počítači podstatně vyšší výkonové a cenové třídy).

Od čtenáře se očekává povšechná představa o počítačích a programování — tedy to, co drtivá většina třeba i začínajících počítačových fanoušků dávno dobře zná. Ukázky programů jsou psány v nejrozšířenějším mikropočítačovém jazyce Basic. Ten, kdo tento jazyk neovládá, se může naučit základy nutné k pochopení uvedených programů z přílohy *Basic ve zkratce*.

Je milou povinností autorů poděkovat těm, kteří se na knížce spolupodíleli: redaktorce Boženě Fleissigové, bez jejíž obětavé práce by rukopis zůstal odstrašujícím příkladem stylistické neobratnosti a pravopisné nezpůsobilosti;
doc. dr. Jiřímu Hořejšovi, CSc., který autory podnítil k napsání této knížky a bez jehož morální podpory by sotva našli odvahu *Programátorské poklesky* spáchat. Jeho poznámky k rukopisu vnesly koncepci do chaosu, který je pro tvůrčí činnost autorů charakteristický;
ing. Petru Adámkovi, jehož cenné připomínky pomohly odstranit z tohoto pojednání o chybách a omylech v programování řadu (nechtěných) chyb a omylů;
v neposlední řadě pak Vladimíru Renčínovi, kterého neodradila ani první verze rukopisu a rozhodl se knížku zachránit svými ilustracemi.

A pokud jste si koupili (či snad pouze vypůjčili) tuto knížku, děkujeme i vám. Jestliže však právě stojíte v prodejně, listujete knihou a váháte, máme pro vás několik dobře míněných rad:

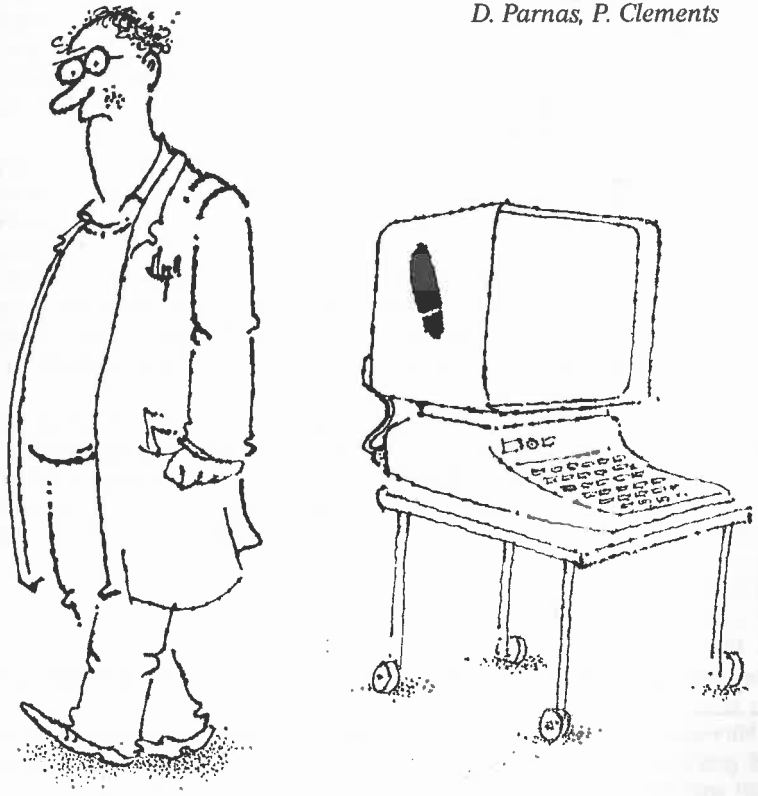
1. Nekupujte tuto knížku, nemáte-li:
   a) rád(a) počítače;
   b) špetku smyslu pro humor;
   c) 15 Kčs.
2. Pokud jste nedal(a) na předchozí radu, potom:
   a) věnujte knihu někomu chytřejšímu (ad 1a);
   b) máte smůlu (ad 1b);
   c) nezapomeňte vrátit vypůjčené peníze (ad 1c).
3. Nečtěte tuto knížku:
   a) při přecházení křižovatky, obzvlášť pokud jste přehlédl(a) červené světlo;
   b) na oslavě narozenin své tchyně;
   c) při pádu s velké výšky; pokuste se raději knížku přebalit, aby při dopadu nedošla úhony.

Budete-li se řídit těmito radami a budete-li knížku skladovat na suchém chladném místě při tlaku 1020 hektopascalů, relativní vlhkosti vzduchu 50 % a v nadmořské výšce 200 metrů, stane se nadlouho vaší milou společnicí. To vám přejí

vaši autoři.

1. NĚCO JAKO ÚVOD (ANEB V KAŽDÉM PROGRAMU JSOU CHYBY)

Chyby působené lidmi je možné vyloučit pouze tehdy, jestliže vyloučíme působení lidí.
D. Parnas, P. Clements



V roce 1980 vyšla kniha Angličana Stephena Pila *The book of heroic failures* (do češtiny bychom její název přeložili nejspíš jako *Kniha velkolepých nezdarů*). Jsou v ní zachyceny historie mnoha slavných omylů, neúspěchů, selhání, chyb, bankrotů, krachů, blamáží a trapasů. Pokud jste schopni přelouskat ji v angličtině (bohužel česky zatím nevyšla) — vřele doporučujeme. Počíst si o cizích nezdarech je vždy povzbudivé.

Co víc, S. Pile vystihl, že neúspěchy a chyby mohou být často mnohem zábavnější, ale i inspirativnější, než obvyklé a fádní dosažení cíle „s jazykem na vestě". Ve své knize po právu konstatuje: „Úspěch je přeceňován. Každý se za ním honí bez ohledu na to, že se denně prokazuje, že génius člověka spočívá v pravém opaku. To, v čem jsme skutečně dobří, je neschopnost: je to vlastnost, která nás odlišuje od zvířat a jíž bychom si měli vážit."

Bohužel, toto pozoruhodné dílo obsahuje pouze jedinou zmínku o nezdaru, k němuž došlo za pomoci počítače (přestože důsledky byly skutečně citelné — meziplanetární sonda Mariner skončila místo na Venuši v Atlantickém oceáně, viz kap. 9). V tomto ohledu musí každý počítačový fanda pociťovat něco jako křivdu. Počítač je přece ideální nástroj na vytváření zmatků a kde jinde než při programování se lidé dopouštějí tak rafinovaných chyb a střetávají se s tak pozoruhodnými nezdary!

To ví každý, kdo zkusil programovat. Mnoho lidí se dnes a denně dopouští nejrůznějších přehmatů, a přesto jsou přesvědčeni o své naprosté neomylnosti. Při programování se to nestává — v tomto smyslu počítače člověka vychovávají. Spatříte-li na lavičce v parku člověka schýleného nad haldou papírů a nevnímajícího okolí, jak čas od času s očima zrudlýma nevyspáním zaúpí „to přece není možné", vyrve si chomáč vlasů, znovu se pohrouží do papírů, aby posléze zajásal „já jsem ale vůl", slušně poklidil papíry i vyrvané chomáče vlasů a důstojně odkráčel — bude to asi programátor, který hledal (a našel) chybu ve svém programu.

Mezi tzv. „programátorský folklór" patří rčení, že v každém programu jsou chyby. Toto tvrzení je prověřeno časem i zkušenostmi a neustále se znovu a znovu potvrzuje jeho pravdivost. Stalo se, že začínající programátor přišel za svým zkušenějším kolegou a pravil: „Říká se, že v každém programu jsou chyby. Jaká je tedy chyba v tomto programu?" S těmito slovy mu předložil následující program:

```
10 REM O. K.
20 LET I=1
30 END
```

„Největší ze všech chyb," zněla pohotová odpověď, „takový program k ničemu není."

Nicméně si přiznejme, že vyčítat Stephenu Pilovi nezájem o programátorské poklesky by od nás bylo zaujaté. Programátorské strasti a radosti jsou stěží sdělitelné těm, kteří této vášni nepropadli. Ne všichni pochopí, že dáte přednost posezení nad displejem svého mikropočítače před shlédnutím detektivního filmu, protože ladění programu je napínavější.

I když chyby a nezdary jsou v jistém smyslu kořením programování, můžeme je stěží považovat za nějakou obzvláštní radost. Způsobují nezdravé zvyšování krevního tlaku se sklerotickými důsledky, což si programátor nemůže dovolit. Zbývá ovšem možnost pobavit se cizími chybami a přitom se vyvarovat vlastních. To měli na mysli autoři, když přemýšleli o koncepci této knížky a o tom, jaké příklady z jejich vlastní programátorské kariéry by mohly být pro čtenáře zajímavé.

Jádrem knížky, kterou držíte v rukou, se tak stalo pojednání o programátorských chybách, omylech a nezdarech — prostě o programátorských poklescích. Na tomto poli mají autoři bohaté a dlouholeté vlastní zkušenosti (což prohlašují ve vší skromnosti — ostatně řada jejich kolegů to jistě ráda potvrdí). V knížce se dočtete i o metodách, jak chybám a nezdarům v programování předcházet. Čtenář nechť již sám rozhodne, zda bude těchto metod využívat, nebo dál sbírat své vlastní zkušenosti metodou pokusů a omylů.

2. ALGORITMY HÝBOU SVĚTEM

Dejte mi pevný bod a pohnu zemí.
Archimédes

Jakmile počítače pronikly do lidské společnosti, začalo se slovo *algoritmy* skloňovat ve všech pádech. Ne že by předtím algoritmy neexistovaly — ale provádět je musel obvykle člověk, a ten cítil, že to není nic pro něj. Například algoritmus pro řešení systému lineárních rovnic známe již po staletí, ale vyřešit takový systém byť i jen o dvaceti neznámých byla dříve práce pro vraha. A to prosím doslova — jeden známý astronom z 18. století dával provádět úmorné výpočty pohybu planet vězňům v místní věznici.

Algoritmus chápeme jako „konečný soubor pravidel, která dávají návod k vyřešení určité třídy úloh". K nejznámějším klasickým knihám obsahujícím algoritmy patří tedy i kuchařka Magdalény Dobromily Rettigové, poskytující návody na řešení důležité třídy úloh, se kterými si dnes mnoho manželek neví rady. Nám však půjde o jiný druh algoritmů, při nichž nejsme ohrožováni nadbytečnými kilojouly a které za nás bude realizovat náš přítel — počítač.

Problémy s problémy

Ten problém sis vymyslel,
řekla mi takticky.
Odpověď na něj je snadná věc,
jen to zvaž logicky.
Paul Simon

Jak vyřešit jakýkoli problém? To je problém, který se dosud vyřešit nepodařilo, a jak si dále řekneme, ani se to podařit nemůže. Lidé měli problémy od nepaměti a nedokáží bez nich žít. Mnozí si je pěstují jako psa nebo kočku a nevěřte těm, co chodí v tričku s nápisem „No problem" nebo „Nema problema". Musíme mít problémy, abychom je mohli řešit, protože řešení problémů je duševní gymnastika, bez níž by náš mozek zakrněl.

Přirozeně, není problém jako problém. Osobní problémy našich přátel jsou únavné svou banalitou a problémy tibetských mnichů zase poněkud odlehlé (s výjimkou jednoho, který si za chvilku uvedeme). My si všimneme těch problémů, při jejichž řešení nám může pomoci počítač. Jsou to problémy

1. logicky nebo matematicky formulovatelné;
2. zahrnující zpravidla nekonečný počet konkrétních případů (v důsledku obecné formulace);
3. vyžadující vymyslet algoritmus, kterým se problém vyřeší.

Problém, o který se zajímáme, může tedy být třeba úloha „seřadit prvky souboru od nejmenšího po největší", ale také „najít algoritmus, který rozhodne, zda v programu je, nebo není chyba". V prvním případě se jedná o problém, jaký řešíme — formou algoritmu — často. Takovým problémům říkáme *algoritmicky řešitelné.* Programátorskými poklesky s nimi spojenými se budeme zabývat ve větší části této knížky.

Ve druhém případě se zřejmě jedná o víc; kdybychom totiž znali příslušný algoritmus, mohli bychom řadu kapitol z naší knížky vypustit. A kdybychom znali algoritmus, který by dokázal opravit všechny chyby v programu, nemělo by smysl tuto knížku vydávat. Naštěstí — vlastně bohužel — tak úžasně užitečné algoritmy neznáme. Nejen to, významná matematická díla nás poučí, že ani neexistují. Problémům takového charakteru říkáme *algoritmicky neřešitelné.*

Abychom byli trochu konkrétnější, uveďme si toto teoreticky dokázané tvrzení:

Neexistuje algoritmus, který by pro kterýkoli daný program rozhodl (na základě zdrojového tvaru programu — viz slovníček — a vstupních dat), zda tento program skončí výpočet. (Obecně není vždy zaručeno, že práce programu skončí, určitá část programu se může do nekonečna opakovat.)

Přitom rozhodnutí, zda program skončí, nebo ne, představuje mnohem jednodušší problém, než zjištění, zda je v něm chyba; vždyť to, že program neskončí, je pouze jedna z ohromného množství chyb, které se v programu mohou vyskytnout.

Co z těchto teoretických výsledků vyplývá? Například to, že vytváření a ověřování programů je tvůrčí činnost a nikoli mechanická záležitost a že na mnohé důmyslné algoritmy lze pohlížet jako na umělecká díla. Ne nadarmo se jedno klasické dílo o programování jmenuje *Umění programovat* [KNUT68].

(P. S. Omlouváme se matematicky fundovaným čtenářům a stoupencům dokonalé přesnosti za poměrně dosti volnou interpretaci pojmů a tvrzení patřících do teorie rozhodnutelnosti, uvedených v tomto odstavci. Ty čtenáře, kteří by se rádi seznámili s touto velmi zajímavou disciplínou a mají potřebné matematické znalosti, odkazujeme na knížku [HOPC78].)

O jednom starém algoritmu

Vymýšlet algoritmy řešící zajímavé problémy patří k oblíbeným kratochvílím. Uvažte jen, kolik času již strávili malí chlapci i důstojní kmeti nad Rubikovou kostkou. Avšak lidé se zabývali algoritmy od nepaměti.

Připomeňme si jeden velmi starý problém (a odpovídající algoritmus), ke kterému se vztahuje zajímavá pověst a který je znám pod názvem „Hanojské věže". Jeho jednodušší varianta je populární už mezi školáky:

Máme tři mince, řekněme korunu, padesátník a desetník, a tři místa na stole, která si označme A, B, C. Na místě A jsou poskládány na sobě postupně koruna, na ní padesátník a na něm desetník. Úkolem je přemístit mince z místa A na místo C tak, aby zde opět ležely na sobě od největší po nejmenší a aby při přemísťování byla dodržena tato pravidla:

1. Přemísťujeme vždy jen jednu minci.
2. Nikdy nesmíme položit větší minci na menší.
3. Mince mohou ležet jen na sobě nebo přímo na místech A, B, C.

Řešení není těžké: desetník položíme na C, potom padesátník na B, desetník na padesátník, korunu na C, desetník na A, padesátník na korunu a nakonec desetník na padesátník. Tak primitivní úlohy jsou ovšem pro malé děti, které si nehrají s mikropočítači.

Vezměme ale všechny existující platné československé mince, tj. pětikorunu, dvoukorunu, korunu, padesátník, dvacetník, desetník a pětník, položme je na pozici A od největší po nejmenší a zkusme je přeskládat podle uvedených pravidel na pozici C. Pokud jste o této úloze dosud neslyšeli a podaří se vám ji vyřešit bez velkého přemýšlení, připište si bod; jistě byste nyní dokázali najít řešení i pro libovolný počet mincí. Klíč k úspěchu spočívá v tom, že si uvědomíme:

1. Umíme-li vyřešit náš problém pro tři mince (což umíme), snadno jej zvládneme i pro čtyři mince: stačí přemístit horní tři mince na B (umíme), nejspodnější minci na C a mince z B na C (rovněž umíme).
2. Zcela analogicky — umíme-li vyřešit problém pro n mincí, umíme jej vyřešit i pro $n+1$ mincí. Vrchních n mincí přesuneme známým postupem na B, spodní minci na C a n mincí z B na C.

Všimněme si, že najít řešení pro sedm mincí není vlastně o nic lehčí než najít řešení pro libovolný počet mincí. Není důležité, že mincí je jenom sedm; najít zkusmo řešení pro sedm mincí je mnohem obtížnější než vyřešit úlohu logicky. (Není bez zajímavosti, že při podezření na poškození mozku nechávají neurofyziologové pacienta řešit problém hanojských věží a jeho postup jim dá cenné informace. Opakované pokusy dokonce pomáhají následky poškození mozku léčit.)

Avšak vraťme se k pověsti, která je s problémem spojena. V jedné tibetské oblasti, v klášteře uprostřed strmých hor, prý existuje náboženská sekta, která ví, kdy bude konec světa. Mnichové této sekty se zabývají naším problémem, jenomže nepracují s mincemi, ale se šedesáti čtyřmi zlatými kroužky různé velikosti, navlečenými na tři tyčky — A, B, C. Abychom nepřišli o jistý prvek romantiky, dodejme, že tyčka A je zlatá, tyčka B železná, tyčka C stříbrná a kroužky jsou osázeny diamanty. Před mnoha sty lety začali mnichové s překládáním kroužků z tyčky A na tyčku C. Až se jim podaří úkol splnit, nastane prý konec světa.

Pokud byste tomu náhodou uvěřili, nemějte žádné obavy. Nemusíte zjišťovat adresu kláštera a psát mnichům, aby nepospíchali, protože si ještě potřebujete dočíst tuto knížku. Nestihli by to v rozumné době, ani kdyby tam pro uspíšení konce světa zřídili komplexní racionalizační brigádu. Snadno lze spočítat, že i kdyby se z mnichů přímo kouřilo a přemístili deset kroužků za sekundu, bude jim to trvat zhruba 58,5 miliardy let.

Třebaže asi neuvěříme legendě o konci světa, nepochybně na nás zapůsobí,

že realizace algoritmu pro čtyřiašedesát kroužků by trvala tak dlouho. To nás nutí k zamyšlení nad tím, které algoritmy vlastně můžeme uskutečnit v rozumném čase.

Složitost algoritmů

Dá se předpokládat, že tibetští mnichové přemísťující zlaté kroužky se příliš nezabývají racionálností a efektivitou své činnosti. Bude jim pravděpodobně jedno, že algoritmus, který provádějí, bude trvat tak dlouho — koneckonců do toho konce světa asi tak moc nespěchají. Naproti tomu my jsme lidé netrpěliví, na provádění algoritmů máme počítače a nejenže se chceme dožít výsledku, chceme ho nejraději hned. Naprogramovat Hanojské věže na počítači jistě není problém — zajímavější je otázka, za jak dlouho výpočet skončí. Odpověď je snadná. Když bude počítač v „přemísťování kroužků" desettisíckrát rychlejší než mnichové (odhadneme-li, že přemístění jednoho kroužku spotřebuje asi 10 strojových instrukcí, vyžaduje to počítač s rychlostí řádově 100 000 operací za sekundu), skončí v 10 000krát kratším čase, tedy „už" za necelých šest miliónů let.

Vidíme, že existují algoritmy zadané malou množinou vstupních dat (64 kroužků pro náš případ), které ani ty nejrychlejší počítače neuskuteční v rozumné době. Je jasné, že to musíme vzít v úvahu a vždy si rozmyslet nejenom správnost algoritmu, ale také dobu jeho trvání na počítači. Jestliže dojdeme k závěru, že algoritmus bude příliš pomalý, nezbývá než se pokusit najít jiný, rychlejší. To se ovšem nemusí pokaždé podařit; například náš algoritmus pro Hanojské věže se zrychlit nedá. (Věděli to mnichové, když s jeho skončením svázali příchod konce světa?)

Často se však skutečně můžeme rozhodnout mezi pomalým a rychlým algoritmem — i když to někdy stojí přemýšlení, shánění odborné literatury a konzultace s odborníky. Uvedeme si jeden klasický případ.

V mnoha problémech se setkáváme s řešením soustavy lineárních rovnic pro n neznámých. Na to máme řadu různých metod. Všimneme si dvou nejznámějších — Cramerova pravidla a Gaussovy eliminační metody. Cramerovo pravidlo je možno zapsat formou vzorečků udávajících výsledné hodnoty neznámých. Gaussova eliminační metoda se zapisuje přímo jako algoritmus.

Nám však nejde o popis těchto metod, ale o jejich porovnání z hlediska rychlosti výpočtu. Je známo, že Cramerovo pravidlo (pokud se determinanty, jež v něm vystupují, počítají přímo z definice) potřebuje zhruba $(n + 2)!$, tj $1 . 2 . 3 . \ldots . n . (n + 1) . (n + 2)$ aritmetických operací a Gaussova eliminační metoda zhruba $n^3/3$ aritmetických operací, kde n je počet neznámých. (Protože jde jen o odhad, bývá zvykem brát při srovnávání algoritmů v úvahu pouze operace násobení a dělení.) Lépe než uvedené vzorečky bude rozdíl mezi oběma algoritmy ilustrovat následující tabulka, která udává (pro $n = 2, 3, \ldots, 10$) závislost mezi počtem neznámých a počtem operací:

počet neznámých	odhad počtu operací pro Gaussovu eliminaci	odhad počtu operací pro Cramerovo pravidlo
$[n]$	[celá část z $n^3/3$]	$[(n + 2)!]$
2	2	24
3	9	120
4	21	720
5	40	5040
6	72	40320
7	114	362880
8	170	3628800
9	243	39916800
10	333	479001600

Vidíme, že počet operací narůstá pro Cramerovo pravidlo mnohem rychleji, než pro Gaussovu eliminační metodu. Pro třicet neznámých dává odhad počtu operací pro Gaussovu metodu číslo 9000, které je z hlediska počítače velmi dobře přijatelné, zatímco na provedení výpočtu pomocí Cramerova pravidla bychom potřebovali 32! operací, což je 36místné číslo — výsledku by se nedočkali ani naši vzdálení potomci.

Je zajímavé, že v řadě případů známe pro určitý problém rychlý algoritmus, zatímco opačný problém dokážeme řešit pouze pomocí algoritmů, které jsou nesrovnatelně pomalejší. Máme-li např. určit součin daných prvočísel, jedná se o problém, který je tak jednoduchý, že si ani nezaslouží pojmenování problém. Chceme-li naopak z daného čísla určit jeho prvočinitele (tj. prvočísla, jejichž vynásobením číslo vznikne), máme před sebou úlohu podstatně obtížnější. Lze namítnout, že ani to není příliš těžké — vždyť již ve starověku to uměl mudrc Eratosthenés (odtud se odpovídající algoritmus nazývá Eratosthenovo síto a lze jej najít v mnoha učebnicích matematiky). To je pravda. Obtíž je však v tom, že všechny známé algoritmy pro rozklad čísla na prvočinitele jsou pomalé. Rozložit číslo na prvočinitele pomocí známých algoritmů vyžaduje takový počet operací, že skutečně velká čísla nezvládnou rozložit ani nejrychlejší počítače (toho využívají moderní metody pro šifrování tajných zpráv.)

Jak je vidět, situace kolem rychlosti algoritmů není právě jednoduchá a je třeba se v tom nějak vyznat. Zabývá se tím celá matematická disciplína — *teorie složitosti algoritmů*. Eventuální zájemce odkazujeme na literaturu (např. [MIKL79]); zde si řekneme jen to, že se jedná (mimo jiné) o studium počtu operací potřebných pro provedení algoritmu a o charakterizování algoritmů právě z tohoto hlediska. Důležitým pojmem této teorie jsou tzv. *polynomiální algoritmy*. Tento pojem je dobré znát a bude snadné si jej vysvětlit.

V našich předchozích případech (Hanojské věže, řešení rovnic, rozklad na prvočinitele) se vždy jednalo o problém, jehož složitost závisela na nějakém přirozeném čísle n. Toto n představovalo v případě Hanojských věží počet

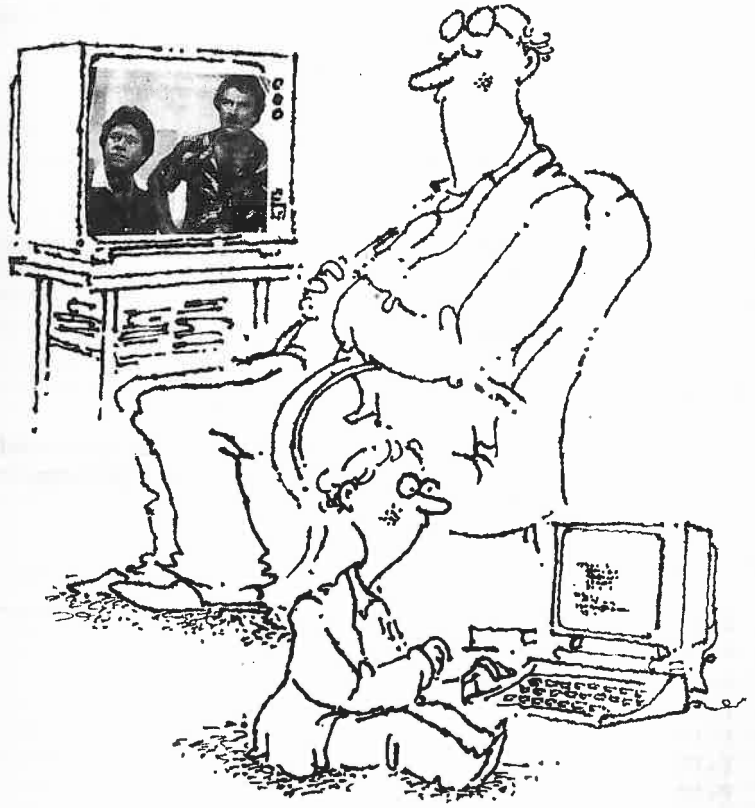
kroužků, které se měly přemístit, v případě řešení soustavy rovnic počet neznámých, v případě rozkladu na prvočinitele číslo, které budeme rozkládat. Pro většinu algoritmů známe nebo umíme odhadnout počet operací, potřebný pro provedení algoritmu (uvedli jsme si např. $n^3/3$ pro Gaussovu eliminační metodu a $(n + 2)!$ pro Cramerovo pravidlo). Je-li možné tento odhad vyjádřit jako polynom (viz slovníček), jedná se o vzpomenuté polynomiální algoritmy. Zhruba se dá říci, že polynomiální algoritmy jsou „slušné" algoritmy, které můžeme vcelku bez obav naprogramovat (pokud ovšem není n příliš velké). Mnohé algoritmy však nejsou polynomiální — např. zmíněný výpočet systému rovnic Cramerovým pravidlem. Setkáte-li se například s algoritmem, který nese označení *exponenciální* — pryč od něj. Váš počítač by vás přestal mít rád, kdybyste na něm chtěli takový algoritmus realizovat.

U exponenciálních algoritmů totiž počet operací roste s číslem n mimořádně rychle. Jako příklad nám může posloužit jeden z nejprimitivnějších algoritmů pro hraní šachů nebo dámy, který v každé situaci probere všechny možné tahy, všechny možné odpovědi soupeře na každý náš tah, všechny naše možné reakce na soupeřovu odpověď atd. Vybere potom takový tah, po němž má soupeř i při nejlepší hře nejhorší vyhlídky. Problém je v tom, že takový algoritmus vyžaduje mimořádně vysoký počet operací. Kdybychom v každé situaci měli jen deset možných tahů (obvykle jich bude více), museli bychom prověřit 100 soupeřových odpovědí, 1000 našich dalších tahů atd.; analýza do hloubky n půltahů tedy dává 10^n možností (odtud plyne i název exponenciálních algoritmů — číslo n se v odhadu složitosti objevuje v exponentu). Je zřejmé, že tímto způsobem je únosné provádět analýzu jen do velmi malé hloubky — řekněme na 4 či 5 půltahů — což pro aspoň trochu dobrou hru nestačí. Kvalitní šachové programy proto používají různé triky, které jim umožní většinu možných tahů předem vyloučit a zaměřit se jen na analýzu těch nejnadějnějších.

Algoritmus a program

Přestože počítače velmi zdůraznily význam algoritmů, jsou algoritmy na počítačích nezávislé; můžeme je formulovat v libovolném přirozeném jazyce, zapsat symbolicky nebo nějakým způsobem kódovat. Současné generace počítačů však vyžadují jejich zápis v některém programovacím jazyce. Kterýkoli programátor musí kromě základních znalostí češtiny (aby se domluvil se svými kolegy) a angličtiny (aby porozuměl hlášením operačního systému) umět ještě aspoň jeden ze záplavy počítačových jazyků: Ada, Algol 60, Algol 68, assembler, Basic, C, Cobol, Forth, Fortran, Fortran 77, LISP, Logo, Modula, Occam, Pascal, PL/1, Prolog, RPG, Simula...

Program, jako zápis algoritmu v nějakém programovacím jazyce, má své specifické aspekty, z nichž některé se vztahují k jazyku samotnému (např. ovlivnění struktury programu charakterem jazyka), jiné vyplývají z implementace programu (viz slovníček).



Vezměme si jednoduchý příklad, který to ilustruje. Je známo, že pro rostoucí číslo n se hodnota výrazu $(1 + 1/n)^n$ stále více blíží číslu $e = 2{,}7182818\ldots$ (základ přirozených logaritmů) — číslo e je tímto způsobem dokonce v matematice definováno. Algoritmus pro výpočet hodnoty čísla e je tedy jednoduchý — abychom dosáhli libovolné přesnosti, stačí použít dostatečně velké číslo n. Čím větší číslo n použijeme, tím přesnější hodnotu čísla e dostaneme.

Vyzkoušejme, pro jak velké n bude hodnota zmíněného výrazu již dostatečně blízká přesné hodnotě čísla e. Výpočet budeme provádět postupně pro $n = 1, 10, 100, 1000$ atd. V Basiku to lze zapsat takto:

```
10 REM VYPOCET CISLA E Z DEFINICE
20 PRINT "N","(1x1/N)**N"
30 FOR I=0 TO 20
40   Nz 10**I
50   PRINT N, (1x1/N)**
60 NEXT I
70 END
```

Po spuštění se na obrazovce začnou objevovat výsledky:

```
N                (1x1/N)**N
 1                2
 10               2.59374
 100              2.70481
 1000             2.71692
 10000            2.71815
 100000           2.71827
 1000000          2.71828
 10000000         2.71828
 100000000        2.71828
```

Pro $n = 10^6$ je tedy výsledek skutečně roven číslu e na pět desetinných míst a dále se nemění. V průběhu dalšího výpočtu však dojde k překvapení:

```
 1000000000      2.71828
 10000000000     2.71828
 100000000000    2.71822
 1.  E+12        2.71792
 1.  E+13        2.71611
 1.  E+14        2.71611
 1.  E+15        2.4307
 1.  E+16        1
 1.  E+17        1
 1.  E+18        1
 1.  E+19        1
 1.  E+20        1
```

Konkrétní hodnoty budou na různých počítačích různé, ale charakter výsledků je vždy týž – zpočátku se postupně blíží číslu e, ale pak se od něj stále více vzdalují, až od určitého n jsou rovny jedné. Na první pohled by se mohlo

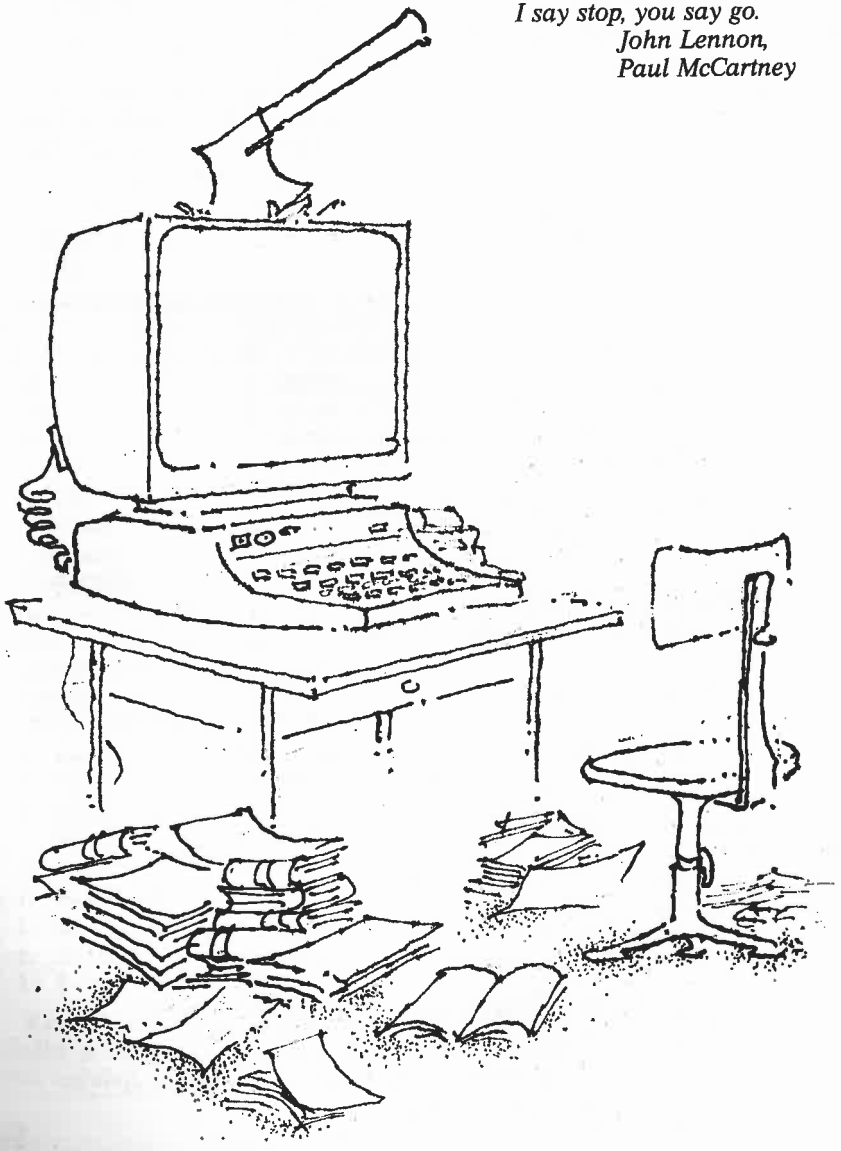
zdát, že se dostáváme do sporu s matematikou nebo že máme chybu v programu.

Ve skutečnosti má záhadu na svědomí počítačové prostředí, do kterého jsme algoritmus přenesli. Počítače mohou zobrazit reálná čísla jenom s jistou přesností (např. na sedm, deset nebo šestnáct platných míst) a vypomáhají si přitom zaokrouhlováním; a právě zaokrouhlovací chyby způsobí paradoxní výsledky. Pro dostatečně velké n je totiž $1/n$ natolik malé ve srovnání s jedničkou, že v rámci přesnosti počítače je $1 + 1/n$ rovno jedné. A jedna umocněna na jakýkoli exponent se ovšem rovná zase jedné.

Naštěstí je zpravidla počítačové prostředí pro většinu algoritmů „přátelštější" a nezpůsobuje podobné potíže. Přesto je dobré mít na paměti, že nás na cestě od algoritmu k programu mohou očekávat skryté nástrahy, které mohou vést k těžko odhalitelným selháním funkce programu.

3. O DIALOGU S POČÍTAČEM

I say yes, you say no,
I say stop, you say go.
John Lennon,
Paul McCartney



V dnešní době většina programů s člověkem komunikuje. Některé dávají požadované informace (mnohdy dokonale ukryté v nepožadovaných), jiné vyžadují lidský zásah, když už nevědí kudy kam, další si s vámi zahrají počítačové hry anebo se neúnavně dožadují nekonečné řady vstupních dat. Systémové programy (operační systém, překladače) nebývají příliš sdílné — občas vyštěknou nějakou, obvykle málo srozumitelnou zprávu, nejspíš v angličtině, a dělejte si s tím, co chcete. Nedivte se však — po tom, co všechno musí vydržet od uživatelů, by každého přešla sdílnost.

Komunikování s počítačem je činnost náročná pro psychiku člověka a bezpochyby i pro psychiku programu a počítače. Všimněme si nyní povahových rysů programů právě z této stránky. Povahové rysy člověka ponecháme psychologům, psychiatrům, dr. Plzákovi aj. Nejdříve si však řekneme, co nás vede k tomu, že mluvíme o psychice programu a počítače, čímž naznačujeme, že s jistou nadsázkou považujeme počítač za intelektuálního partnera člověka.

Člověk, počítač a Eliza

Člověk je omyl přírody
Gilbert

Počítač je zařízení, které dokáže za minutu udělat tolik chyb, kolik by tisíc matematiků dělalo deset let.
Programátorský folklór

I když mezi člověkem a počítačem je nepochybně řada podstatných rozdílů, můžeme zde najít také řadu podobností, a to ne nahodilých. Počítač i člověk komunikují. Počítač i člověk dokáží provádět abstraktní činnost — například aritmetické a logické operace (i když některé lidi k tomu nepřinutíte, kdybyste se rozkrájeli). Počítače i lidé se vyvíjejí. Říkáme-li zde, že se počítače vyvíjejí, a nikoli že jsou vyvíjeny, není to příliš velká nadsázka. Při vývoji a konstrukci nových počítačů se totiž používají velkou měrou zase počítače.

Ve výčtu vlastností, které jsou společné nebo blízké lidem i počítačům, bychom mohli pokračovat, ale to není účelem. Jistá „duševní příbuznost" tu nepochybně je. Abychom si uvědomili její rozsah, představme si tento experiment: Člověku Č je předložen záznam dialogu mezi A a B; Č ví, že se jedná o dialog mezi počítačem a člověkem, a má rozhodnout, zda počítač je A anebo B. Podotýkáme, že se jedná o dialog vedený v přirozeném jazyce, a to zcela volně ze strany člověka i počítače. Nejsou tedy stanovena žádná pravidla, jak a o čem se má vést rozhovor, probíhající pochopitelně prostřednictvím klávesnice a obrazovky. Je jasné, že je třeba vytvořit velmi důmyslný program, aby člověk Č nedokázal rozhodnout, zda počítač je A nebo B.

Takové experimenty se skutečně provádějí a nejsou žádnou samoúčelnou hříčkou. Jedná se o jeden ze základních problémů teorie umělé inteligence,

významný například i pro pátou generaci počítačů, která se dnes ve světě vyvíjí.

Slavný program, v němž počítač imituje v dialogu člověka, vznikl v USA a jeho varianty a fragmenty jsou roztroušeny po celém světě. Název programu — Eliza — byl inspirován hlavní postavou hry G. B. Shawa „Pygmalion", ve které se také jedná o problémy jazykové komunikace. Náš experiment můžeme různě modifikovat — Eliza dostane např. za úkol imitovat psychologa a Č má určit, zda se jedná o počítač, nebo o skutečného psychologa. Pro ilustraci se podívejme na úryvek autentického krátkého dialogu s programem typu Eliza-psycholog (program, který jsme pro tuto ukázku použili, komunikuje v angličtině; zde jsme úryvek přeložili).

E = Eliza, A = jeden z autorů této knihy:
E: *Máte nějaký psychologický problém?*
A: *Nemám dost peněz.*
E: *Udělal jste v poslední době něco pro zvýšení svého finančního příjmu?*
A: *Neudělal, protože jsem líný.*
E: *Myslíte si, že je normální být líný?*
A: *Nemyslím, ale mám strach z práce.*
E: *Chtěl byste, aby vám vaši přátelé pomohli překonat strach z práce?*
A: *Nemohly by mi pomoci počítače?*
E: *Snad si nemyslíte, že počítače mohou pomoci lidem?*

Zde Eliza dokázala vcelku úspěšně hrát předepsanou roli. Přesto bychom asi uhádli, že zde psychologa imitoval počítač — není pravděpodobné, že by skutečný psycholog po první odpovědi pokračoval dál ve vlídném tónu. (Podotkněme, že autor, který s Elizou konverzoval, v rozhovoru nevyjadřoval své skutečné názory a problémy.) Tuto Elizu je snadné poplést — jedná se však o malý program, čítající necelých 450 řádků v jazyce Pascal. Dokonalejší varianty jsou však schopny samy se učit a organizovat svou strukturu. Takové programy můžeme považovat v jistém smyslu za inteligentní.

Běžné programy sice nedokáží komunikovat s uživatelem na takové úrovni, ale přesto můžeme najít řadu podobností mezi jejich chováním a chováním inteligentních bytostí.

Jak se domluvit s počítačem

Dialog mezi Elizou-programem a člověkem je zdvořilý, obvykle zábavný a bez jakéhokoli rizika. Takže vcelku o nic nejde. V řadě případů je však dialog člověka s počítačem na hony vzdálen takové idylce. Dokladem toho jsou následující ukázky (P = program, U = uživatel):

P: **ZADEJ VSTUPNI HODNOTY**
U: *(vytřeští oči a začne zoufale uvažovat, jaké vstupní hodnoty má vlastně zadat, v jakém pořadí, v jakých jednotkách a jakým způsobem je má zapsat; v zoufalství se nakonec přece jen rozhodne)*
A=1 B=2 C=3

P: **INPUT ERROR 216126**
nebo
SYSTEMCRASH9459WEW454
nebo
$ END OF JOB BYE $

Po takovém „dialogu" začne zpravidla zdrcený uživatel (kterým klidně může být sám autor programu) pátrat, jak odpovědět, aby se s ním program vůbec dál bavil, a nadává přitom na autora programu anebo provádí sebekritiku.

Ještě ošemetnější situace nastává, když program požaduje data, a přitom nedá uživateli žádné upozornění. A třebaže většina systémů požadavek na zadání vstupních dat nějakým způsobem indikuje, může to uživatel snadno přehlédnout. Pak na sebe začnou člověk s programem vzájemně čekat a než chudák uživatel konečně zjistí, na čem je, uplyne mnohdy spousta času.

Na první pohled slušněji, ale o to záludněji, se chová jiný program v následujícím dialogu:

P: **ZADEJ VSTUPNI HODNOTY PARAMETRU A, B, C**
U: **1 2 3**
P: **CHYBNY TVAR VSTUPNICH DAT - ZOPAKUJ ZADANI**
U: **1, 2, 3**
P: **CHYBNY TVAR VSTUPNICH DAT - ZOPAKUJ ZADANI**
U: **A=1 B=2 C=3**
P: **CHYBNY TVAR VSTUPNICH DAT - ZOPAKUJ ZADANI**
U: *(krevní tlak roste z původních 120/90 na 130/100, puls vzrůstá ze 70/min na 80/min)*
A=1, B=2, C=3
P: **CHYBNY TVAR VSTUPNICH DAT - ZOPAKUJ ZADANI**
U: *(krevní tlak 150/120, puls 100)*
1.Ø, 2.Ø, 3.Ø
P: **CHYBNY TVAR VSTUPNICH DAT - ZOPAKUJ ZADANI**
U: *(krevní tlak 180/150, puls 130)*
A=1.Ø B=2.Ø C=3.Ø
P: **CHYBNY TVAR VSTUPNICH DAT - ZOPAKUJ ZADANI**
U: **JDI K CERTU!!!**
(hroutí se a je posléze odvážen záchrankou)
P: **CHYBNY TVAR VSTUPNICH DAT - ZOPAKUJ ZADANI**
U: *(po přeložení z jednotky intenzívní péče se od autora programu dovídá, že správný tvar vstupních dat měl být:*
1;2;3
Jeho zdravotní stav se poté opět prudce zhorší a vrací se zpátky na jednotku intenzívní péče).

Méně časté jsou případy, kdy na nedorozumění má jednoznačně vinu člověk. Obvykle se jedná o osobu, která se k počítači dostane nedopatřením, proti své vůli nebo omylem. Konverzace pak vypadá např. takto:

P: **BUDOU SE VYSLEDKY TISKNOUT? (A=ANO N=NE)**

U: NEVIM
P: ODPOVEZ A (ANO) NEBO N (NE)
U: NA CO MAM ODPOVEDET?
P: ODPOVEZ A (ANO) NEBO N (NE)
U: A (ANO) NEBO N (NE)
P: ODPOVEZ A (ANO) NEBO N (NE)
U: *(odchází hluboce uražen tím, že si z něho počítač dělá legraci a že mu dokonce tyká).*

Uživatel v tomto příkladě vlastně ani není uživatelem. Je to nešťastná oběť počítačové éry. Může to být v jádru dobrý člověk, může rozumět poezii a mít rád zvířata, hrát na ukulele nebo ovládat svahilštinu, ale nikdy si nebude rozumět s počítači.

Případ unikátního nedorozumění v dialogu počítač — člověk se udál ve Francii a byl před jistou dobou publikován v tisku. Protože nemáme k dispozici solidní údaje a fakta, pokusíme se o volnou rekonstrukci této události.

Uklízečka, která pracovala ve výpočetním středisku jisté firmy, byla náhle vyrušena akustickým signálem jednoho terminálu. Netušila, že příčinou byla několikerá nahodilá kolize jejího smetáku s klávesnicí zmíněného terminálu, což mělo za následek vyvolání jednoho ze servisních programů firemní databanky a poté akustický signál podobný pípnutí kuřete. Na displeji terminálu stálo:

```
 * UKLIDOVY PROGRAM FIRMY XX PRO DATABAZI YY *
POUZITI POVOLENO POUZE ZODPOVEDNYM PRACOVNIKUM
PREJETE SI PROVEST AKCE V PLNEM ROZSAHU?
(ODPOVEZTE ANO NEBO NE)
```

Když se vzpamatovala z prvního překvapení, začala uklízečka logicky uvažovat: za úklid je zodpovědná ona, je tedy zřejmě zodpovědný pracovník. Od počítače je nepochybně hezké, že jí chce pomoci s úklidem, ale domluví se s tou elektronickou obludou? Zeptat se však nebylo koho (bylo po směně), a tak to zkusila s počítačem podobně, jako když jí onehdá programátoři předváděli jednu počítačovou hru; vyťukala **ANO** (když dělat pořádek, tak pořádně) a zmáčkla červené tlačítko v pravém rohu klávesnice. Počítač vzápětí odpověděl:

```
PREJETE SI VYCISTIT
A) DATOVE POLOZKY
B) ODKAZOVE TABULKY
C) VSECHNO
(ODPOVEZTE A NEBO B NEBO C)
```

Uklízečka neviděla žádný důvod pro to, aby se nevyčistilo všechno, a odpověděla **C**. Další dotaz byl méně srozumitelný:

```
JAKA AKCE SE MA PROVEST?
A) AKTUALIZACE
B) KONDENZACE
C) ZRUSENI DAT A UVOLNENI PRISLUSNEHO PROSTORU
(ODPOVEZTE A NEBO B NEBO C)
```


Odpovědi **A** i **B** byly zcela nesrozumitelné. Odpověď **C** však vypadala slibně. Všude po středisku se v zaprášených stozích listingů povalovalo nekonečné množství dat a uklízečka se již několikrát marně pokoušela přimět programátory, aby si v nich udělali pořádek a aspoň trochu prostoru uvolnili. A tak se uklízečka po krátkém váhání rozhodla pro **C**. Počítač ohlásil

```
ZAHAJENO PLNENI POZADOVANYCH AKCI
```

a následujících několik hodin strávil tím, že systematicky likvidoval životně důležitou databazi — a tím i firmu.

Příležitostí ke zmatení partnera v dialogu člověk — program je dost a dost. (Některými dalšími problémy se budeme ještě zabývat v intermezzu *O češtině a „pocitacstine"*.) Na jakých zásadách by se tedy měla stavět komunikace mezi počítačem a uživatelem?

1. Zprávy a dotazy musí být formulovány jasně a jednoznačně.
2. Požaduje-li se odpověď, je třeba přesně a srozumitelně uvést její tvar.
3. Je nezbytné zabezpečit, aby po případně nesprávné odpovědi program nehavaroval, ale aby se vypsala zpráva upozorňující na nesprávnost a podávající znovu, pokud možno podrobněji, informaci o správném tvaru odpovědi, možném číselném rozsahu apod.
4. Je třeba věnovat pozornost dobrému rozvržení textu na obrazovce tak, aby byl přehledný a názorný. Je vhodné používat „dotazníkové" rozvržení textu, kdy se zadává větší počet údajů (celá strana) současně.
5. Je velmi užitečné povolit, aby co největší počet zadávaných položek mohl uživatel vynechat. V takovém případě bude automaticky dosazena nejčastější hodnota. Uživateli však musí být zcela jasně sděleno, jakou hodnotou se každý vynechaný údaj nahradí.

Sangvinik, flegmatik, cholerik, melancholik

Binární radosti a digitální povzdechy
jsou jak milenky
jsem jeden z mnoha a kradu ti
jenom myšlenky
jsem k tobě user-friendly
Ian Anderson: User-friendly
(z alba Walk into Light)

I korektně komunikující programy se od sebe často výrazně liší. Podívejme se, v čem se tyto odlišnosti mohou projevovat.

Program typu *sangvinik*
je přátelský, zdvořilý, trpělivý a na uživatelovy přehmaty reaguje s nadhledem, v krajním případě s jemnou ironií. Takové programy se často označují anglickým termínem *user-friendly* (což znamená „přátelský k uživateli"). V následujícím dialogu je S sangvinický program a U uživatel:

```
S:      ==      PREJI VAM MNOHO USPECHU      ==
   A PRIJEMNOU ZABAVU PRI POUZITI TOHOTO PROGRAMU
    PRO NUMERICKOU INTEGRACI ROMBERGOVOU METODOU

   ZADEJTE PROSIM NEJPRVE INTEGRACNI INTERVAL (A, B)
   VE TVARU A B (TJ. A MEZERA B)
   (A, B - REALNA CISLA V BASIKOVSKEM ZAPISU,
   A MUSI BYT MENSI NEZ B)
U: YGUKYGYRF
S: OBAVAM SE, ZE JSTE SE DOPUSTIL DROBNE CHYBICKY.
   ZADEJTE PROSIM ZNOVU INTEGRACNI INTERVAL...
   (opakuje znovu požadovaný tvar údajů)
U: RTYR5YU9
S: ANI TENTOKRAT TO NEBYLO ONO. ZKUSTE SE UVOLNIT
   A KONCENTROVAT.
   ZADEJTE PROSIM ZNOVU INTEGRACNI INTERVAL...
   (opakuje znovu požadovaný tvar údajů)
U: WERWRE56CFT
S: NE. JE TO NA VAS MOC TEZKE. ZKUSTE NAJIT VE SVEM
   OKOLI OSOBU S UKONCENYM ZAKLADNIM VZDELANIM A POZADAT
   JI O POMOC.
   ZADEJTE PROSIM ZNOVU INTEGRACNI INTERVAL...
   (opakuje znovu požadovaný tvar údajů)
U: FTYDFUWE
S: PATRNE JSTE JIZ DUSEVNE VYCERPAN, PROTO RADEJI
   SKONCIME. NIC JSME SICE NESPOCITALI, ALE BYLO TO
   DOCELA ZABAVNE, NENI-LIZ PRAVDA?
   KONEC
```

Podotkněme, že zde uživatel úmyslně odpovídal chybně, aby tím víc vynikly příjemné povahové rysy programu typu sangvinik.

Program typu *flegmatik*

Flegmatický program se na rozdíl od sangvinického vůbec nesnaží být na uživatele příjemný a jeho zdvořilost je spíše chladná (pokud ovšem flegmatický program vůbec nějaké známky zdvořilosti vykazuje). V případě, že se uživatel soustavně dopouští omylů v zadání vstupních dat, bude bez zájmu neustále opakovat svou stručnou žádost, dokud nevypnou proud nebo to uživatele nepřestane bavit. Flegmatikovy zprávy jsou strohé a jejich grafická úprava je minimální.

Program typu *cholerik*

Takové programy reagují na uživatelovy chyby bouřlivě a přímočaře. Vědouce, že jim nikdo fyzicky neublíží (zkuste program inzultovat), ani nebudou kádrově postiženy, dopouštějí se výroků, kterých by se často rádi dopustili jejich autoři. V takových situacích jde u nich zdvořilost stranou a používají ta-

ková slova, která teprve v poslední době pronikají do umělecké literatury. Mezi oblíbené výroky cholerických programů (z těch, které se zde osmělíme citovat) patří:

PUJCTE MI NEKDO OCI, AT SE NA TO INDIVIDUUM MUZU PODIVAT

nebo

PROSIM KOLEMJDOUCI, ABY SE O TOHO UBOZACKA POSTARALI

Při komunikaci s cholerickými programy se zpravidla nenudíte, avšak tato pozitivní stránka věci obvykle ustoupí do pozadí, pokud s vaším programem komunikuje váš nadřízený nebo vaše tchyně (nemají-li ovšem smysl pro humor — ten však nadřízeným a tchyním obvykle chybí).

Program typu *melancholik*

Melancholikové jsou, jak známo, lidé citliví. Mají zasmušilý pohled a při zprávách o počasí se rozpláčou, protože hodnocení povětrnostní situace v Čechách a na Moravě v nich budí asociace na Stříbrný vítr či probouzí zaváté vzpomínky. Zdálo by se, že v tomto případě analogii ve světě počítačů nenajdeme.

Skutečně, případy melancholických programů jsou vzácné — ale i takové programy existují. Jeden pochází dokonce od profesionální softwarové firmy (z taktnosti nebudeme jmenovat) a jeho posláním je hrát s uživatelem šachy. Náš přítel-šachista líčil průběh utkání s tímto programem asi takto:

Zápas probíhal na mikropočítači ZX Spectrum. Program hrál s chutí a radostně až do chvíle, kdy byl postaven před dvoutahový mat. Když zjistil tuto skutečnost, zesmutněl a obrazovka se zamlžila. Neodpověděl ani po dvou minutách (což je doba, do které — podle záruk autorské firmy — by měl vždy reagovat), ani po dvaceti. Ve zbědovaném stavu musel být nakonec ukončen vytažením síťové zástrčky. (Možná jsme líčení trochu přehnali — ve skutečnosti si nejsme úplně jisti, zda zápas probíhal na mikropočítači ZX Spectrum nebo Sharp.)

4. INTERMEZZO O ČEŠTINĚ A „POCITACSTINE“

Programovat se učíme podle pana Jacksona, Dijkstry, Dahla, Wirtha a dalších. Prostředky k tomu nám dodávají všemocné firmy IBM, ICL, DEC atd. Ale tu češtinu, tu si budeme muset vyřešit sami.

V. Novák, S. Zdebski

„Ten sort s apdejtovanejma opšnama mi vypliv' mraky vórningů, tak jsem džob po čekpointu típnul, Svičnul jsem se pod sysmana a zkolektoval júzrhuky, ale hodilo to systym eror.“

„Zkus dylítnout vérkfajly a vyendovat se. Pak to exekni z meku do malýho beče s trasováním na olejblovanou pásku a autput esajnuj na flopáč. Jestli to krešne, vylistuj žurnál s dampem, kilni spůlfajly a před odlogováním vytajduj modulovku. Oukej?“

Při poslechu tohoto fragmentu rozhovoru mezi dvěma ostřílenými programátory by pracovníkům Ústavu pro jazyk český pravděpodobně běhal mráz po zádech a laická veřejnost by se zřejmě domnívala, že se jedná o esperanto, protože některá slova v té divné mluvě jsou přece česká.

Existence počítačového slangu, založeného téměř výlučně na anglických termínech, je důsledkem toho, že většina rozšířených počítačů vznikla v místech, kde se mluví anglicky, nebo aspoň byla z této jazykové oblasti inspirována. Kromě toho se angličtina stala v počítačové oblasti referenčním jazykem, podobně jako latina v medicíně. Naneštěstí není počítačová anglická terminologie ustálena — liší se zejména v závislosti na výrobci počítače, ale i v závislosti na jednotlivých systémech. (Například pro spojovací program používá firma ICL ve třech operačních systémech tři různé názvy — composer, consolidator, collector — kdežto IBM a většina ostatních firem dává přednost čtvrtému názvu — linkage editor.) V důsledku toho se i počítačový slang více či méně liší v závislosti na výrobci počítače nebo na operačním systému. Výše uvedený dialog proto nemusí být ani pro zkušené programátory od jiných počítačů příliš srozumitelný.

Nedorozumění vzniklé neustáleností počítačového slangu se však obvykle rychle vysvětlí, i když mezitím může vést k humorným či trapným situacím. Obětí slangu se tak stal i návštěvník jednoho střediska, který obcházel kanceláře vyptávaje se na ing. Sysmana; jako programátor zcela jiného počítače nemohl ovšem tušit, že odkaz „s touhle chybou zajděte radši za *sysmanem*“ znamená v místním slangu „obraťte se na *systémového programátora*“ (SYSMAN = SYStem MANager).

Počítačový slang však není pouze českou záležitostí. I v jiných jazycích, nevyjímaje angličtinu, můžeme nalézt slang přímo ukázkový. Například v jistém velmi rozšířeném operačním systému se vyskytuje chybové hlášení, které musí potěšit každého milovníka angličtiny: „Disc XYZ is non-IPLable". (Chce se tím říci, že z disku nelze provést tzv. Initial Program Load, tj. spustit operační systém.)

Projevem historické svázanosti počítačů s angličtinou je i skutečnost, že s málokterým počítačem lze komunikovat skutečně česky — většinou jsme přinejmenším omezeni na anglickou abecedu. Na české texty bez háčků a čárek jsme si naštěstí díky telegramům a dálnopisu zvykli dávno před nástupem počítačové éry, a tak nás chybějící diakritická znaménka příliš nezaskočí. V některých případech však může dojít k nejednoznačnosti významu slova (RADY = *rady* nebo *řady* nebo *řády* nebo dokonce *rády*?) a potom musíme být opatrní. Ostatně — nejednoznačnost nemusí mít na svědomí „POCITACSTINA"; například ve zprávě

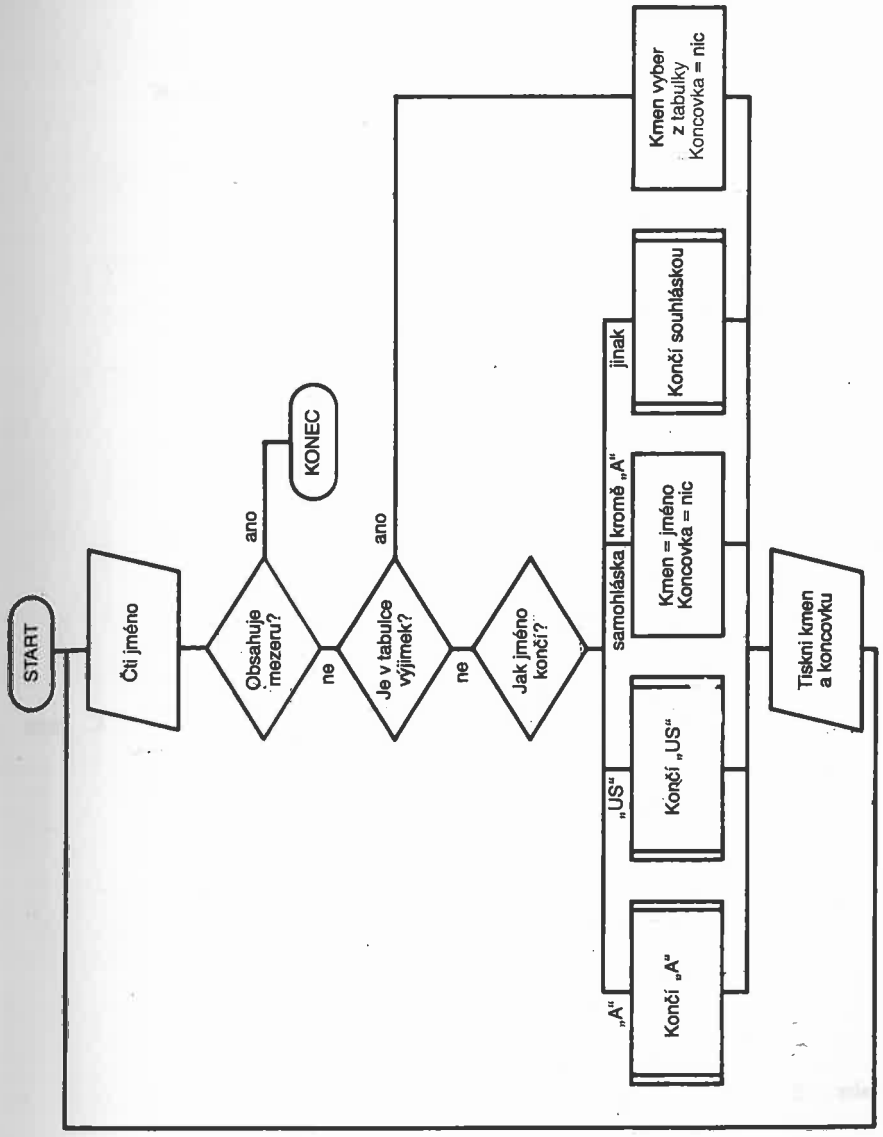
SOUBOR A NAHRAZUJE SOUBOR B

nechybí žádná čárka ani háček, a přesto není jasné, který soubor je nahrazen kterým.

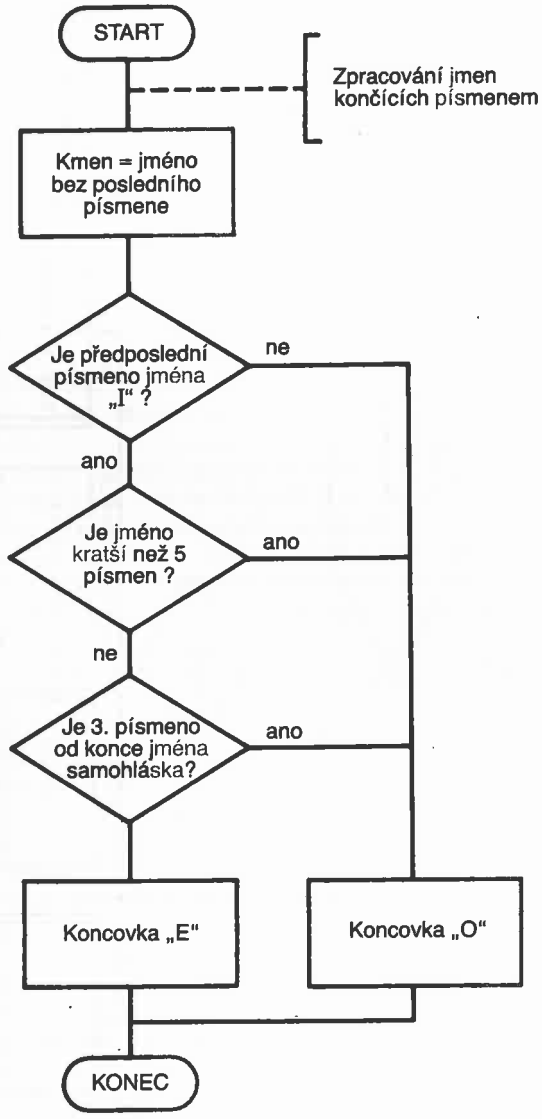
Ani slang ani potíže s českou abecedou nepatří ovšem k největším problémům (s abecedou jsme na tom pořád ještě lépe než Japonci nebo Arabové). „POCITACSTINU" lze uživateli opepřit například i používáním nedostatečně srozumitelných nebo matoucích zkratek, jako je např. PRM (parametr? program?), SOHO (má znamenat současnou hodnotu, ale spíše vyvolává asociace se známou londýnskou čtvrtí neřesti), MICKA (kdepak kočička — minimální cenová kalkulace! — s předběžnou cenovou kalkulací bychom dopadli ještě hůř), PGM NAM (program name — kdo nezná anglicky, je ztracen), MPRCK DJ (empirické údaje — zkratka je „vtipně" inspirována anglickým zvykem vynechávat přednostně samohlásky).

Avšak se skutečnými obtížemi se setkáme až při pokusech o počítačovou manipulaci s českým textem. V angličtině je to podstatně jednodušší — podstatné jméno má ve většině případů pouze dva tvary, sloveso tři nebo čtyři, zatímco v češtině se při skloňování a časování setkáme většinou alespoň s desítkou různých tvarů. Navíc v angličtině často tentýž slovní tvar může podle postavení ve větě označovat podstatné jméno, přídavné jméno nebo sloveso (např. „store" znamená „paměť", „paměťový" i „uložit do paměti"), zatímco v češtině tato slova musíme přísně rozlišovat. Modifikovat program Eliza z předchozí kapitoly tak, aby s uživatelem konverzoval v mluvnicky správné češtině, by proto vůbec nebylo jednoduché.

Uvedeme zde program pro mnohem jednodušší úlohu: přečíst křestní jméno a oslovit uživatele v 5. pádě. Postup tvorby 5. pádu je dán vývojovým diagramem na obr. 1—5 a odpovídající program vypadá takto (čtenář ho samozřejmě nemusí detailně studovat; slouží nám jednak jako ilustrace složitosti zdánlivě jednoduchého problému, jednak jako příklad použití vývojových diagramů popisovaných v příloze 2):

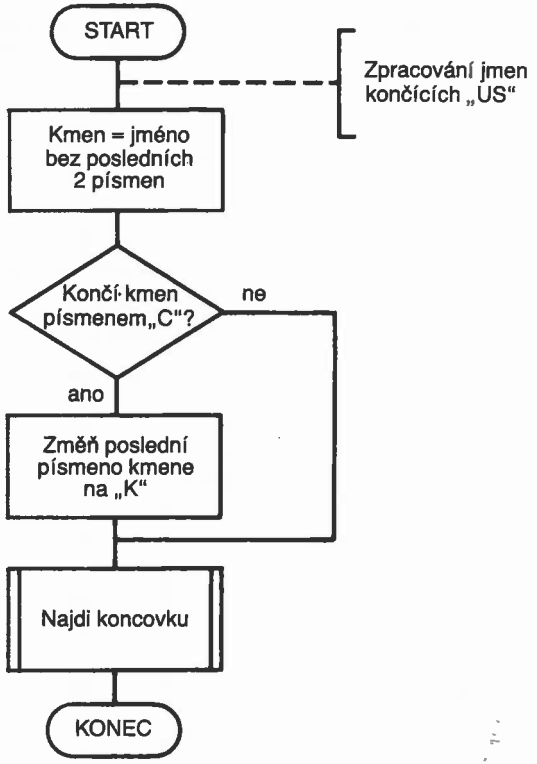


obr. 1



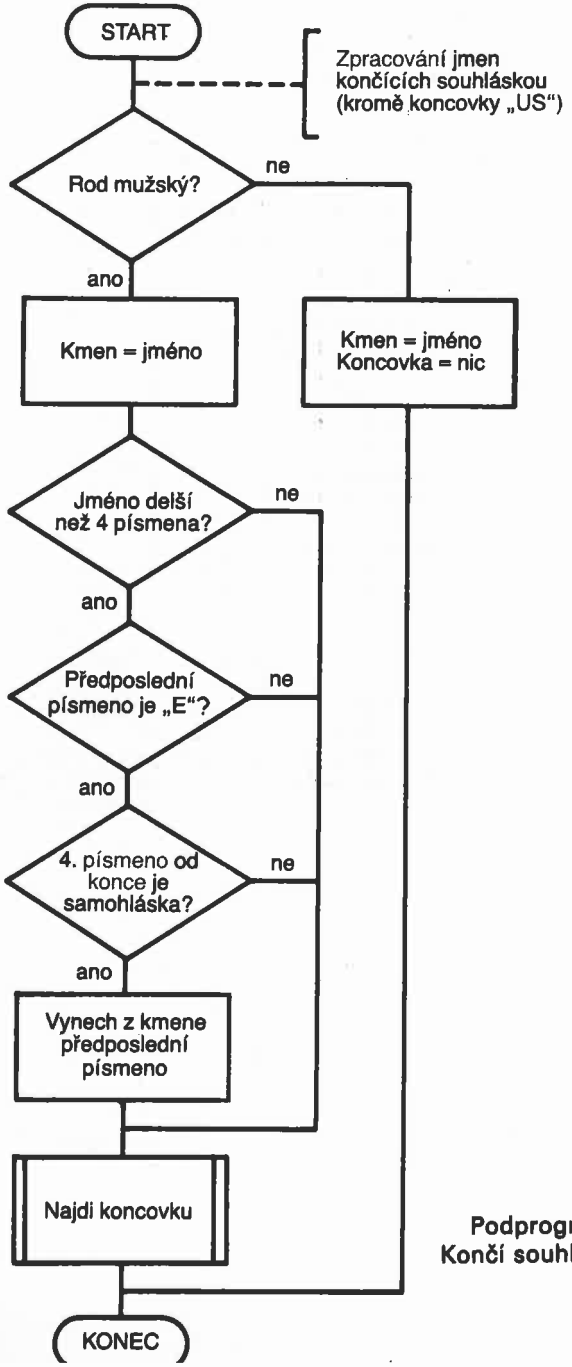
obr. 2

Podprogram Končí „A“



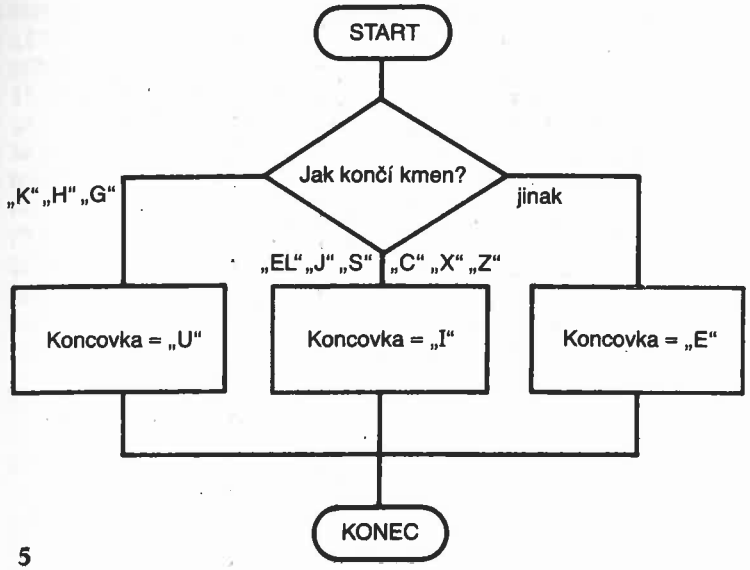
obr. 3

Podprogram Končí „US“



obr. 4

Podprogram
Končí souhláskou



obr. 5

Podprogram Najdi koncovku

```
10 REM PREVOD KRESTNIHO JMENA DO 5. PADU
20 REM TABULKA VYJIMEK
30 DIM N$(9),V$(9)
40 READ N
50 MATREAD N$,V$
60 DATA 9
70 DATA LEV,JOSEF,DAGMAR,ZEUS,JUPITER,BUH,VUL,BLBEC,KUN
80 DATA LVE,JOSEFE,DAGMARO,DIE,JOVE,BOZE,VOLE,BLBCE,KONI
90 REM CTI JMENO
100 PRINT "JAK SE JMENUJES (KRESTNI JMENO)?"
110 INPUT J$
120 LET L=LEN(J$)
130 IF POS(J$," ")=0 THEN 150
140 STOP
150 REM ZPRACUJ VYJIMKY A VTIPALKY
160 FOR I=1 TO N
170 IF J$<>N$(I) THEN 210
180    LET Z$=V$(I)
190    LET K$=""
200    GOTO 740
```

```
210 NEXT I
220 REM JMENO NENI V TABULCE VYJIMEK
230 REM URCI POSLEDNI PISMENO
240 LET P$=SUB$(J$, L, 1)
250 IF P$<>"A" THEN 380
260 REM JMENO KONCI "A"
270 LET Z$=SUB$(J$, 1, L-1)
280 IF SUB$(J$, L-1, 1)<>"I" THEN 360
290 IF L<5 THEN 360
300 REM 3. PISMENO OD KONCE JE SAMOHLASKA?
310 LET O$=SUB$(J$, L-2, 1)
320 GOSUB 970
330 IF V=1 THEN 360
340 LET K$="E"
350 GOTO 740
360 LET K$="O"
370 GOTO 740
380 IF SUB$(J$, L-1, 2)<>"US" THEN 460
390 REM JMENO KONCI "US"
400 LET Z$=SUB$(J$, 1, L-2)
410 IF SUB$(Z$, LEN(Z$), 1)<>"C" THEN 430
420 Z$=SUB$(Z$, 1, LEN(Z$)-1)&"K"
430 REM NAJDI KONCOVKU
440 GOSUB 770
450 GOTO 740
460 REM KONCI JMENO SAMOHLASKOU KROME "A"?
470 LET O$=P$
480 GOSUB 1010
490 IF V=0 THEN 540
500 REM JMENO KONCI JINOU SAMOHLASKOU NEZ "A"
510 LET Z$=J$
520 LET K$=""
530 GOTO 740
540 REM JMENO KONCI SOUHLASKOU
550 PRINT "JSI MUZ?"
560 INPUT O$
570 IF SUB$(O$, 1, 1)<>"N" THEN 620
580 REM ZENSKE JMENO KONCICI SOUHLASKOU
590 LET Z$=J$
600 LET K$=""
610 GOTO 740
620 REM MUZSKE JMENO KONCICI SOUHLASKOU
630 LET Z$=J$
640 LET L=LEN(Z$)
650 IF L<5 THEN 720
```

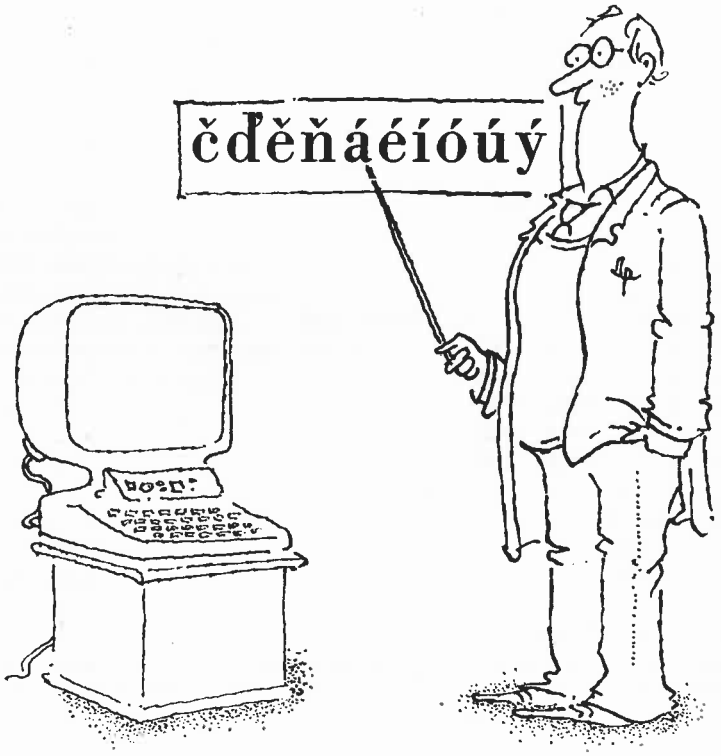


```
660 IF SUB$(Z$, L-1, 1)<>"E" THEN 720
670 REM JE 4. PISMENO OD KONCE SAMOHLASKA?
680 LET O$=SUB$(Z$, L-3, 1)
690 GOSUB 970
700 IF V=0 THEN 720
710 LET Z$=SUB$(J$, 1, L-2)&P$
720 REM NAJDI KONCOVKU
730 GOSUB 770
740 REM TISKNI OSLOVENI
750 PRINT "AHOJ ";Z$;K$;"!"
760 GOTO 90
770 REM PODPROGRAM "NAJDI KONCOVKU"
780 LET L=LEN(Z$)
790 LET O$=SUB$(Z$, L, 1)
800 IF O$="K" THEN 840
810 IF O$="H" THEN 840
820 IF O$="G" THEN 840
830 GOTO 860
840 LET K$="U"
850 GOTO 960
860 IF SUB$(Z$, L-2, 2)="EL" THEN 930
870 IF O$="J" THEN 930
880 IF O$="S" THEN 930
890 IF O$="C" THEN 930
900 IF O$="X" THEN 930
910 IF O$="Z" THEN 930
920 GOTO 950
930 LET K$="I"
940 GOTO 960
950 LET K$="E"
960 RETURN
970 REM PODPROGRAM "SAMOHLASKA"
980 REM JE-LI O$ SAMOHLASKA, POLOZI V=1
990 REM PRI VCHODU RADKEM 1010 SE NETESTUJE "A"
1000 IF O$="A" THEN 1080
1010 IF O$="E" THEN 1080
1020 IF O$="I" THEN 1080
1030 IF O$="O" THEN 1080
1040 IF O$="U" THEN 1080
1050 IF O$="Y" THEN 1080
1060 LET V=0
1070 RETURN
1080 LET V=1
1090 RETURN
1100 END
```

Ačkoli program má více než 100 řádků, neřeší úlohu zcela uspokojivě (například pro jména „Řehoř", „Bohuš" nebo cizí jména „Jacques" a „Krésus").

Oč jednodušší práci má anglický, ruský, německý či francouzský kolega, kterému stačí napsat něco jako:

```
10 PRINT "WHAT'S YOUR FIRST NAME?"
20 INPUT J$
30 PRINT "HI ";J$;"."
40 END
```



5. PROGRAM, NEBO DŽUNGLE?

Jsou dvě cesty, jak přistupovat
k psaní počítačových programů:
první — vypracovat je tak
jednoduché, že v nich zřejmě nejsou
žádné nedostatky;
druhá — vypracovat je tak složité, že
v nich nejsou žádné nedostatky
zřejmé.

T. Hoare

Cesta do hlubin počítačovy duše vede přes zdrojový text programu. Obvykle není snadné se v něm vyznat — existují však i programy zapsané tak, že pokus zorientovat se v nich je hazardováním s duševním zdravím. Naproti tomu kolují zprávy o programech, v nichž jsou i rozsáhlé a komplikované algoritmy zapsány srozumitelně a přehledně. Šťastlivců, kteří takové programy spatřili, není mnoho; zato je mnoho knih a článků o tom, jak je vytvářet. Podíváme se teď na některé typické prohřešky proti srozumitelnosti.

Totéž se dá vyjádřit různě

Zde je např. úryvek z básníka Greye, dosud
známého badatelům ve školních škamnách:
„Což v urně nebo bustě na hrobkách
se v těla schránu vrátí prchlý duch?
Což můž' břesk pocty vzkřísit zmlklý prach
či lichocení vniknout v smrti sluch?"
V podstatě stejnou myšlenku, avšak v jiné
formě, čteme v současném úryvku
z Huxleyho Fyziologie:
„Na otázku, zda je možno po okamžiku, kdy
nastala smrt, uvést srdeční komory znovu do
pohybu umělým zaváděním kyslíku, musíme
odpovědět záporně."
Oč prostší, a přece daleko výraznější než
Greyovy umně složité obraty!

S. Leacock: Literární poklesky

Podobně jako přirozené jazyky dávají i programovací jazyky dosti velkou volnost v tom, jak vyjádřit určitou myšlenku, či jak zapsat určitou část algoritmu. Například program pro vytištění prostředního z čísel A, B, C můžeme v dialektu Basiku, použitelném např. na počítačích ZX Spectrum nebo Commodore 64, napsat takto:

```
100 IF A>B AND B>C THEN PRINT B: GOTO 160
110 IF B>C AND C>A THEN PRINT C: GOTO 160
120 IF C>A AND A>B THEN PRINT A: GOTO 160
130 IF A<=B AND B<=C THEN PRINT B: GOTO 160
140 IF B<=C AND C<=A THEN PRINT C: GOTO 160
150 IF C<=A AND A<=B THEN PRINT A
160 REM PROSTREDNI CISLO BYLO VYTISTENO
```

Zná-li náš Basic funkci MIN a MAX, můžeme napsat stručně:

```
100 PRINT A+B+C-MAX(A, B, C)-MIN(A, B, C)
```

nebo:

```
100 PRINT MAX(MIN(A, B), MIN(B, C), MIN(A, C))
```

nebo také:

```
100 PRINT MIN(MAX(A, B), MAX(B, C), MAX(A, C))
```

V „čistém" Basiku se nám program tak přehledně napsat nepodaří — dopadne to např. takto:

```
100 IF A>B THEN 170
110 IF B>C THEN 140
120 PRINT B
130 GOTO 200
140 IF A>C THEN 190
150 PRINT C
160 GOTO 200
170 IF B>C THEN 120
180 IF A>C THEN 150
190 PRINT A
200 REM KONEC ALGORITMU
```

Časté skoky strukturu programu zaplétají — v mnoha programech však vznikají přirozenou cestou ještě mnohem větší zapletence. Celkem snadno zjistíme, že spolehlivě program zašmodrcháme, budeme-li často používat příkaz skoku — GOTO. Koneckonců, můžeme tak vytvořit celý úsek programu, který nebude dělat vůbec nic (tj. nic rozumného — po skončení výpočtu bude všechno jako na začátku), ale zato bude díky své nesrozumitelnosti budit obdiv „myšlenkovou hloubkou":

```
100 LET I=1
110 IF I>1 THEN 130
120 GOTO 220
130 LET I=I+1
140 GOTO 210
150 IF I>5 THEN 190
160 IF I>2 THEN 180
```



```
170 GOTO 130
180 GOTO 110
190 LET I=2
200 GOTO 240
210 LET I=I+1
220 IF I>5 THEN 150
230 GOTO 190
240 LET I=I-1
```

Toto malé bludiště jsme sice zkonstruovali uměle, ale uvažte, že je to vlastně jen pár řádků a že se zde vyskytuje jediná proměnná. To dává do jisté míry tušit, v jak nepřehlednou džungli se může změnit podobný program o stovkách nebo tisících příkazů a proměnných. (Mezi programátorskou obcí se takovým programům říká někdy špagety—programy, neboť jejich propletená struktura připomíná italské národní jídlo.)

Viděli jsme, že některé programy jsou čitelné a srozumitelné — je do nich snadno vidět, jsou „otevřené", zatímco jiné jsou nečitelné a nesrozumitelné, je do nich vidět jen s obtížemi nebo vůbec ne, podobně jako do lidí s uzavřenou povahou. Vypůjčíme si opět terminologii od psychologů a programy patřící do první skupiny budeme označovat za *extrovertní* a programy patřící do druhé skupiny za *introvertní*.

Introverti

Jejich typická vlastnost, totiž že do nich není vidět, je v počítačovém světě jednoznačně na obtíž. I když může jít o programy z funkčního hlediska dokonalé, máme řadu důvodů, proč se nám nečitelnost introvertů nelíbí:

— změny, opravy a doplňky takových programů jsou obtížné;
— máme menší důvěru ve správnost programu, kterému rozumíme špatně anebo vůbec ne — i přesto, že může být zcela v pořádku a dávat dobré výsledky;
— použitelnost takových programů stojí a padá s dostupností dokonalého popisu zadávání vstupních dat a funkce programu, nebo s dostupností jejich autora.

Z větší části jsou introverti důsledkem živelného a nepromyšleného postupu při jejich tvorbě. Jejich autoři mohou být sami velmi schopní programátoři, kteří se díky své vysoké inteligenci nějakou dobu ve stvořené džungli vyznají — avšak kromě nich už nikdo. Tento typ introvertů tedy vděčí za svůj vznik nevhodnému naprogramování; mohly být stejně dobře napsány přehledně a čitelně.

Malá srozumitelnost druhého typu introvertů je přímým důsledkem algoritmu vybraného pro řešení problému; programátor zvolí nevhodný, zbytečně těžkopádný algoritmus. Zde není možno situaci napravit vhodným naprogramováním, zato nevhodným přivedeme srozumitelnost programu na absolutní nulu.

Situace, kdy introvertnost programu vyplývá již z povahy zvoleného algo-
tmu, však může nastat paradoxně také z právě opačného důvodu — totiž
ýběrem mimořádně efektivního algoritmu (např. velmi rychlého při řešení
lohy, kde je právě rychlost rozhodující, nebo zaujímajícího minimum paměo-
ového prostoru v situaci, kdy jsou rozhodující paměťové nároky). Podobně
ako geniální lidé jsou i geniální algoritmy ve většině případů introvertní.
takovém případě musíme ovšem menší srozumitelnost algoritmu a progra-
u chápat jako daň za vysokou efektivitu. (Několik jednodušších příkladů na
ektivní a zároveň introvertní programy najde čtenář v následující kapitole
- viz odstavce *Bílý trpaslík* a *Modrý přízrak*.)

xtroverti

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

```
ØØ REM PROGRAM NANIC
Ø PRINT "1+1="; 2
ØØ END
```

řestože se nepochybně jedná o velmi přehledný a snadno srozumitelný pro-
am).

Pokud se týká velkých a složitých programů, musíme po pravdě přiznat, že
ne zatím neměli to štěstí setkat se s takovým, jehož pochopení by bylo
ačkou a nevyžadovalo jistou duševní námahu. Samozřejmě, že prostudová-
rozsáhlého a komplikovaného programu je náročné. Jde však o to, aby
ostoucí velikostí programu neklesala jeho srozumitelnost neúměrně rychle.
odným naprogramováním lze mnohé zachránit a ztráty na srozumitelnosti
přehlednosti udržet i u rozsáhlých programů ve snesitelných mezích. Tím,
nám v tom může pomoci dobrá metodika programování, se budeme zabý-
t 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 pohlcuje 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.

na

ıemá nic společného s astronomií, ale přesto je v jistém smyslu opakem
ě díry — vydává příliš mnoho informací, více, než je žádoucí. Stejně tak
.ovají i programy patřící do této kategorie. Zahrnují nás opisy vstupních
:ombinovanými s hlášeními o provedených kontrolách, detailními výpisy
výsledků, zprávami o průběhu jednotlivých fází programu, pomocnými
.icími výpisy apod. Samozřejmě, že tyto informace mohou být v určitých
cích užitečné, ba dokonce i nutné. Pokud jsou však automaticky vypiso-
všechny, a navíc navzájem promíchané, mohou se změnit v informační
:s, ve kterém jen s obtížemi nalezneme skutečně potřebné informace.
:istují i patologické případy, kdy všechny výstupní údaje tvoří balast,
mž se žádné užitečné informace neskrývají. Na jednom semináři o pro-
ıování bylo referováno o výpočetním středisku, kde se každý měsíc zpra-
:vá agenda, jež vychrlí menší valník výstupních sestav. Ty se odvážejí do
ıiště postaveného zvlášť pro tento účel. Jiný smysl agenda nemá.

ý obr

ento termín jsme si opět vypůjčili z astronomie. Označují se tak relativně
dné hvězdy s nepatrnou hustotou a obrovskými rozměry. Program „rudý
se vyznačuje tím, že zabírá v paměti mnohem více místa, než kolik je
ečně zapotřebí. Příčin vzniku rudých obrů je mnoho. Jednou z nich může
neefektivní zobrazení dat v počítači. Ujasněme si to na příkladu.
áš mikropočítač nám může pomoci šetřit pohonné hmoty tím, že vybere
:ratší cestu ze zadaného města A do zadaného města B. Odhlédneme teď
aktu, že nejkratší cesta nemusí být vzhledem ke stavu komunikací tou nej-
ɔdnější, a pro jednoduchost budeme předpokládat, že dvě spolu sousedící
ta jsou spojena jen jednou silnicí. Náš program tedy bude potřebovat
ıpní data obsahující informace o tom, jaká je vzdálenost mezi jednotlivý-
ıousedícími městy. Data můžeme elegantním způsobem uložit v tzv. mati-
é formě; v programu deklarujeme dvourozměrné pole, řekněme pole P,
dnotlivá města očíslujeme postupně od jedné. Do P(I, J) uložíme vzdále-
t mezi městy I a J, pokud I sousedí s J, nebo nulu, pokud I a J nesousedí.
dpokládáme-li, že počet měst nepřekročí tisíc, musíme pole P deklarovat
:azem

```
IM P(1000, 1000)
```

ž vznikl krásný rudý obr — toto jediné pole obsahuje milión čísel (a na
ɔha počítačích se proto nedá použít, nebo alespoň nikoli v Basiku). Drtivá
šina položek v poli P bude samozřejmě obsahovat pouze nuly, takže velký
stor, který pole v paměti obsazuje, využíváme velmi neekonomicky. Po-
l si tuto skutečnost uvědomíme, můžeme tytéž informace uložit zhuštěně:
ıadneme-li, že v průměru má jedno město nanejvýš 5 bezprostředních sou-
ů, vejdou 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 šedesátým 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 tři 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 ma-ých úpravách použít i pro rafinovanější šifru, kdy se písmena nahrazují méně spořádaně, např. A se mění na F, B na T, C na A atd. I tuto šifru však lze pracovat, aniž by se z programu stal rudý obr — stačí na začátku programu efinovat šifrovací tabulku dlouhou 26 znaků a *n*-té písmeno abecedy nahra-it *n*-tým písmenem z této tabulky:

```
Ø REM SIFROVACI TABULKA
Ø LET T$="FTA..."

ØØ LET N=CHR(Z$)-64
1Ø REM N JE PORADOVE CISLO PISMENA V ABECEDE
2Ø PRINT SUB$(T$,N,1);
```

Funkce **SUB$** je opět popsána v příloze 1.)

ílý trpaslík

je v jistém smyslu opakem rudého obra. V astronomii tak nazýváme hor-ou, velmi malou, ale velmi hutnou hvězdu. Zatímco rudý obr má průměrnou ıstotu menší než vzduch a Slunce je v průměru o něco hustší než voda, má r bílého trpaslíka hmotnost miliónů tun.

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

Bílí trpaslíci jsou většinou dílem profesionálů, avšak překvapivě často se skytují také jako výsledek vtipných idejí programátorů-amatérů. Za záro-k bílého trpaslíka by snad bylo možno považovat poslední šifrovací pro-am z odstavce o rudých obrech.

Klasickým bílým trpaslíkem je algoritmus pro výpočet data, na něž v da-m roce připadnou velikonoce. Podle dávné tradice je velikonoční nedělí ne-le následující po prvním jarním úplňku. Její určení tedy zdánlivě vyžaduje zsáhlé astronomické výpočty — stanovení okamžiku, kdy začíná astrono-cké jaro (to je 20. nebo 21. března v různých hodinách) a okamžiku, kdy je ěsíc přesně v úplňku. Přesto geniální německý matematik Karl Friedrich auss objevil pro tento výpočet nepříliš složitý algoritmus, který se v Basiku zapsat takto:

```
INPUT R
LET D=MOD(19*MOD(R,19)+24,3Ø)
LET D=22+D+MOD(5+2*MOD(R,4)+4*MOD(R,7)+6*D,7)
IF D<57 THEN 6Ø
LET D=D-7
IF D>31 THEN 9Ø
PRINT D;". BREZNA"
STOP
9Ø PRINT D-31;". DUBNA"
1ØØ END
```

Můžete si ověřit porovnáním s kalendáři, že algoritmus funguje pro **R** od 1901 do 2000, tj. pro kterýkoli rok ve 20. století (pro jiná století je nutno změnit konstantu **24** v řádku **2Ø** a konstantu **5** v řádku **3Ø**). Nepokoušejte se však pochopit, jak funguje — ztratili byste možná tolik času, že na dočtení této knížky by vám už žádný nezbyl. (Gauss bývá považován za člověka s nejvyšším IQ v dějinách lidstva a sledovat myšlenky géniů zpravidla není snadné.)

Často se s bílými 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**. Tutéž 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 $0{,}956.10^4$, 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. $0{,}956 . 10^4 + 0{,}75 . 10^3$) změníme zápis druhého čísla na $0{,}075 . 10^4$.
2. Takto upravená čísla nyní snadno sečteme. V našem případě bude $0{,}956 . 10^4 + 0{,}075 . 10^4 = 1{,}031 . 10^4$.
3. V případě potřeby výsledek upravíme tak, aby mantisa zůstala v rozmezí od 0,1 do 0,99999... . V našem případě $1{,}031 . 10^4$ změníme na $0{,}1031 . 10^5$. (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 $0{,}0 . 10^5$. Nezapomeňme na to, že šesté a další místo mantisy se ztrácí. A zkusme to rovnou na několika číslech:

12,345: $0{,}12345 . 10^2 + 0{,}00000 . 10^5 =$
$= 0{,}00012 . 10^5 + 0{,}00000 . 10^5 =$
$= 0{,}00012 . 10^5 =$
$= 0{,}12000 . 10^2$, tj. 12

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

0,625: $0{,}62500 \cdot 10^0 + 0{,}00000 \cdot 10^5 =$
$= 0{,}00000 \cdot 10^5 + 0{,}00000 \cdot 10^5 =$
$= 0{,}00000 \cdot 10^5 =$
$= 0{,}00000 \cdot 10^0$, tj. 0

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 počítačích založeno na dvojkové nebo šestnáctkové soustavě a nikoli na soustavě desítkové. Na principu to však nic nemění a náš 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 počítačích bychom avšak neuspěli z jiného důvodu: řada počítačů má logiku operace sčítání natolik chytrou, 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 počítač dosud nenarazili.) Další úskalí je svázáno s použitým programovacím jazykem: v řadě jazyků nelze nulu s nestandardním exponentem vůbec zakódovat (překladač vyhodnotí i zápis `ØE6` 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 trní. Ačkoli jsou bílí trpaslíci programy vzorné a záslužné, jednu nepříjemnou vlastnost bohužel mívají — často jsou totiž typickým příkladem introverta z kapitoly *Program nebo džungle*. Zavrhovat 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 později — 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 počítače sice reagují se stoickým klidem, ale stejně si myslí o tvůrci 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*-tém místě v tabulce je to písmeno, kterým se má nahradit *n*-té písmeno z abecedy" můžeme použít princip „na *n*-tém 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 tolikáté 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:

```
1ØØ DIM A(5Ø)
11Ø LET S=Ø
12Ø FOR I=1 TO 5Ø
13Ø   LET S=S+A(I)*LOG(X)
14Ø NEXT I
```

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