OPTIMALIZACE PROGRAMU

NEJSOU TAKÉ ŽÁDNÉ ČÁRY

Jiří Pobříslo

V návodu k TI 58/59 je jedna velice moudrá věta: „Program není třeba upravovat, dává-li správné hodnoty, nejsou-li komplikace v jeho obsluze a vejde-li se do paměti. Tato věta platí – jenže i tady se vyskytne to všude přítomné ale. Někdy se stane, že potřebujeme trochu víc místa, že se nám nedostává pamětí, že nemáme programové kroky nebo že výpočet trvá dlouho. Potom je třeba program upravit, doladit, prostě dotáhnout do konce. Mnohdy nám stačí jej zminimalizovat. Začátečník s tím mívá mnoho starostí, ale ve skutečnosti to není tak složité.

Cvičný program uvedený pod názvem „Program nejsou žádné čáry" v ročence AR-81 je výborně podaný návod na „výrobu programu". Chybí mu snad jedině postupový diagram, ale přiznejme, že jednak není vždy nutný a jednak by mohl začátečníky i trochu poplést. Podíváme-li se tedy na uvedený program z hlediska té citované věty, tak určitě vyhoví. Jenže – jestliže cvičně vyrábíme nějaký program, proč si nezkusit jej cvičně minimalizovat a upravit, jakoby nás tísnil naprostý nedostatek kroků. Až se nám to stane opravdu, velice se nám hodí, když už v tom budeme umět chodit. Mohu vám zaručit, že se dostanete do situace, kdy vám bude jeden jediný ušetřený krok dobrý, neřkuli víc.

Zkusme to tedy s naším cvičným programem pro převod dekadický kód na binární a opačně. Nejdříve podrobíme program myšlenkovému rozboru. Co je na něm dobrého, je nasnadě. Je logický, matematicky správný a funguje. Více nás bude zajímat to, co se nám na něm nelíbí.

1. Zařazení vynulování všech pamětí (CMs) jako předposlední krok. Tak jak je program napsán, to je v pořádku a paměti budou pro každý výpočet znulovány. Jenomže programy, které budeme dělat, nejsou cvičné, používáme je ke skutečným výpočtům a lehko se může stát, že po proběhnutí výpočtu budeme na kalkulátoru počítat ručně a snadno bychom mohli zapomenout nějaký údaj v paměti (registru) 02. Když potom spustíme výpočet, obdržíme chybný výsledek, protože paměti se nevynulují. Ve skutečnosti nemusíme nulovat všechny paměti, stačí aby se vynulovala ta paměť, do které se přičítá výsledek. To je paměť 02. Jenže příkaz 0 STO 02 je přece jen delší a tak zůstaneme u CMs. Je tedy výhodné zařadit nulování paměti na začátek jako první krok. Tady se ovšem objeví malá potíž. Začátky jsou dva: A nebo B. Musíme tedy CMs dát do obou. To nám kroky neušetří, naopak přidá, ale je to nutné a tak se s tím musíme smířit. Později se podíváme, zda by se s tím nedalo něco dělat.
2. Horší situace je v registru t. Ten není v celém programu vůbec definován. O to snadněji může dojít k chybě. Registr t se často při ručním výpočtu používá jako rychlá operační paměť a snadno v něm může něco zůstat. Protože testujeme proti nule, je nutno ji do t zapsat, a to příkazem 0 x/t, což jsou dva kroky a je tedy lepší použít instrukci CP, kterou můžeme zařadit kamkoli do společné větve programu před první test x = t.

   Zatím jsme tedy provedli spíše minimalizaci naruby, protože jsme přidali dva kroky, ale byly nutné. Nemusíme před spuštěním programu nic hlídat.
3. Na začátku programu zapisujeme jednak převáděné číslo (říkejme mu X) a několik konstant. Ve větvi A je to 0,1; 2; 10 a ve větvi B 0,5; 2; 10. Protože se tyto konstanty zapisují v každé větvi, jsou jakoby „zbytečně" zdvojené a vyplatilo by se co nejvíc zredukovat toto zdvojení. Tak především – proč jsou v programu hodnoty 0,1 a 0,5? Hodnota 0,1 vytváří číslo 10^{n-1} a 0,5 vytváří číslo 2^{n-1}. Ve skutečnosti však 0,1 a 0,5 vůbec nepotřebujeme, protože při výpočtech vytváříme řady: 1 10 100 . . . atd. a 1 2 4 . . . atd. V obou řadách začínáme shodným členem = 1. Jestliže v programu přesuneme ono vynásobení konstantou 2 (10) tedy programové kroky 033 až 036 – čili instrukce RCL 03 Prd 01 někam dál, až za místo, kde zjišťujeme dělitelnost beze zbytku, tak můžeme začít číslem 2^0 a 10^0, což je 1 a je to hodnota společná, která se dá zapsat ve společné větvi a nikoli zvlášť. Navíc je zápis 1 STO 01 kratší než 1 STO 01. I tady na tom trochu ušetříme.
4. Začátek výpočtu by se po těchto úvahách mohl realizovat a vypadal by asi takto:

```
Lbl A          Lbl B
CMs            CMs
STO 05         STO 05
10 STO 03      2 STO 03
2 STO 04       10 STO 04
GTO ---------------->
                1 STO 01
                CP
```

   Vidíme, že obě části jsou shodné – až na prohození konstant R3 a R4. Začne se nám jevit výhodné z těchto dvou větví udělat společný podprogram a odskočit si na něj pomocí příkazu SBR z hlavního programu. V části A ponechat zápis tak jak je a v části B, konstanty registrů 3 a 4 prohodit. Sdružením na podprogram něco ušetříme, prohozením, které je navíc, zase něco ztratíme. Snažme se tedy příkazy k prohození udělat co nejkratší. Jak na to? Nabízí se řešení tohoto typu:

```
RCL 03
Exc 04
STO 03
```

   To je celkem šest kroků. Je tady ovšem ještě jiné řešení, které vychází z dobré znalosti vnitřního modulu TI 58/59. Každý z programů, který je v něm a je nám kdykoli k dispozici, provádí řadu matematických úkonů a také organizačních úkonů jako je nulování, přesuny v registrech a podobně. Když si postupně přepíšeme programy těchto modulů a rozebereme-li je, zjistíme, že mnoho částí těchto programů můžeme použít, protože tím ušetříme kroky vlastního programu. V tomto okamžiku nám stačí vědět, že jestliže vydáme instrukci **Pgm 4E'**, odskočí si kalkulačka na program 4 a provede toto: vynuluje zobrazovač, číslo z R1 zapíše do R3, číslo z R2 zapíše do R4, číslo z R3 zapíše do R1 a číslo z R4 zapíše do R2. Stručně řečeno: vymění mezi sebou paměti 01 a 03 a paměti 02 a 04. My sice používáme paměti 03 a 04, ale není problém místo 03 použít paměť 02. Příkaz k prohození by potom zněl Pgm 4E; to jsou jen tři kroky a začíná to vypadat slibně. Že program prohodí také paměti 01 a 03 je nám jedno, protože v nich zatím nic nemáme.
5. Už bychom tedy mohli napsat, jak bude program vypadat, ale musíme si ještě něco říci o návěstích a příkazech ke skokům. Jako příkaz k odskoku na podprogram slouží instrukce SBR a návěstí – tedy například SBR x^2. Když si však uvědomíme, že instrukce A znamená ve skutečnosti SBR A, zjistíme, že by nám to mohlo ušetřit kroky. A i B máme obsazeny, tak použijeme tlačítko C. Obdobným způsobem můžeme instrukci třeba GTO 033 nahradit přímo instrukcí D. Ovšem nesmíme potom již nikde použít instrukci INV SBR, ale to splníme. Podprogram umístíme na začátek programu. Mohl by být v podstatě třeba na konci, ale při chodu programu pracuje kalkulátor takto: narazí-li někde v programu na instrukci (třeba) C, pochopí ji jako SBR C. Přečte si číslo následujícího kroku a uloží jej do své paměti jako adresu návratu. Potom skočí na adresu 000 a jede po programu, aniž provádí zapsané instrukce tak dlouho, dokud nenarazí na návěstí Lbl C. Od následujícího kroku začne provádět výpočet podle zapsaných instrukcí tak dlouho, dokud nenarazí na instrukci INV SBR (s indexem 92, která bývá také v programech označována jako RTN – return). Pak se vrátí na adresu návratu, kterou si zapsal, a od tohoto kroku pokračuje ve výpočtu. Z těchto důvodů umístíme podprogram na začátek, aby kalkulátor zbytečně neběhal. Šetříme tím čas výpočtu.

   Můžeme tedy zkusit přepsat začátek programu i když možná i do tohoto programu ještě zasáhneme.

   Začneme podprogramem. Nejdříve návěstí **Lbl C.**
   Potom vynulujeme všechny paměti **CMs.**
   Vynulujeme registr t. O příkazu CP jsme na začátku tvrdili, že jej umístíme kamkoli do společné cesty před první x = t. Umístíme jej tedy sem **CP.**
   Na zobrazovači máme převáděné číslo X. Předchozí operace je nezměnily, zapíšeme je do registru 5 **STO 05.**
   Zapíšeme konstantu 10 do registru 2 **10 STO 02.**
   Druhou konstantu zapíšeme do registru 4 **2 STO 4.**
   Všimněte si, že jsem nenapsal STO 04. To proto, že za tímto příkazem nebude následovat číslo. Kalkulátor si sám zapíše naši instrukci jako STO 04. Neušetříme sice krok, ale při programování mačkáme o tlačítko méně. Tohoto postupu se budeme držet během celého programu.
   Zvolíme registr pro vytváření čísla 10^{n-1} (2^{n-1}). Kvůli výměně registrů R1–R3 a R2–R4 použijeme raději registr R6. Pro zápis 1 se nabízí sekvence 1 STO 06, ale raději použijeme instrukci Op 26, což je o krok méně. Operace 26 přičte 1 k paměti 06 a protože víme, že v paměti R6 máme 0, můžeme ji použít. **Op 26.**
   Ukončení podprogramu (return) **INV SBR.**
   Výpočtová větev A (převod dekadický na binární) začne návěstím **Lbl A.**
   Odskočíme si na podprogram **C.**
   Po jeho proběhnutí je v této větvi vše hotovo a můžeme začít se samotným výpočtem (v původním pramenu to byl

krok 33). Protože my si označíme začátek výpočtu návěstím (Lbl D), stačí nám zde příkaz, který nahradí instrukci
GTO 033 **D.**
Toto je celá větev A.
Výpočtová větev B (převod binární na dekadický). Začátek bude stejný **Lbl BC.**
Teď máme ale konstanty obráceně a musíme je prohodit. Jak jsme již dříve řekli, provedeme to sekvencí **Pgm 4 E'.**
Tím větev B končí a můžeme začít výpočet a to návěstím **Lbl D.**
Dále se musíme stejným způsobem podívat na samotný výpočet.

6. Na začátku výpočtu je test „je již výpočet skončen"? (kroky 40 až 42), kde odpověď „ano" přesune výpočet na konec, na výpis střádací paměti (na kroky 061 atd.). Těsně před tímto (na krocích 058 až 060) je příkaz ke skoku na začátek, aby se opakoval výpočet (GTO 033). Tyto dva příkazy ke skoku můžeme sloučit a to tak, že test na skončení výpočtu dáme na konec výpočtové řady, a současně otočíme pomocí instrukce INV znění otázky, takže dostaneme test: „je ještě co převádět?". Odpověď „ano" nám vrátí výpočet na začátek a příkaz GTO může zcela odpadnout. Bude to mít sice za následek, že při prvním běhu nebude proveden test zda ještě je nějaké číslo k převodu, ale při prvním běhu je vždy, jinak bychom je nepřeváděli. Ostatní běhy toutéž smyčkou již testovány budou. Prakticky to můžeme realizovat sekvencí **RCL 05 INV x = t** a jako adresu nemusíme použít číslo kroku začátku výpočtu (033), ale protože máme na začátku výpočtu návěstí, stačí nám jako adresa instrukce **D.**

7. Během výpočtu často manipulujeme s pamětí 05. Několikrát píšeme STO 05 a RCL 05 apod. Zápis a výpis této paměti vždy zaujme dva kroky. Zkusíme místo paměti 05 použít registr t. Výpis i zápis se zjednoduší, stačí na to instrukce x≥t. Musíme si jenom uhlídat, kde co zrovna máme. Ale to zvládneme. Přímý důsledek bude třeba to, že v úvodním podprogramu (za Lbl C) nahradíme STO 05 instrukcí x≥t. Tím ušetříme krok. Dále odpadne instrukce CP, protože to celé musíme udělat trochu jinak. Přesněji vzato musíme předělat test z předchozího bodu. To půjde jednoduše, protože testujeme x = t? Je tedy jedno, zda je 0 v t a X na zobrazovači nebo naopak. Výsledek testu je stejný. V registru t máme zapsané číslo X, které chceme testovat. Náš test z předchozího bodu bude vypadat nyní takto:

```
0
INV x = t
D
```

8. Výměnou R5 s t se také změní zkouška na dělitelnost, která je v původním programu na krocích 037 až 052. Nejdříve se podívejme na způsob jakým v původním programu vypisujeme a dělíme: číslo zapsané v R5 zbavíme desetinné části, dělíme obsahem R4. Výsledek přepíšeme do R5 a dále zjišťujeme, zda za desetinnou čárkou bylo „něco" (zbytek) či ne. V původní verzi vypadá program takto:

```
RCL 05
Int
: RCL 04 =
STO 05
INV Int
```

Protože my máme základní číslo v t, bude naše sekvence vypadat takto:

```
x≥t
: RCL 04
+ Int
x/t
= INV Int
```

Vidíme, že i zde jsme ušetřili krok. Tato sekvence je na první pohled poněkud složitá, ale nesmíme zapomenout, že při instrukci STO 05 zůstane původní číslo na zobrazovači a zapíše se současně i do R4, zatímco instrukce x≥t toto číslo vymění s číslem zapsaným v t. Protože však X na zobrazovači potřebujeme i dál, využijeme paměti vnitřní hierarchie kalkulátoru (HIR) a dostaneme se do ní tak, že „předstíráme" matematický úkon nižšího řádu. V našem případě je to ono +, a to „předstírání" realizujeme tak, že k danému X přičteme 0, která se na zobrazovači objeví při druhé instrukci x≥t. Přesněji vzato na zobrazovači se objeví při prvním běhu smyčkou (půjde-li program částí A) číslo 2. Při ostatních bězích a dále v celé části B tam bude 0. Ovšem přičtení 2 v tomto jediném případě nijak nevadí, protože vzápětí odtrhneme celky a případný špatný údaj nás vlastně nezajímá. A kde se tam ty 2 vezmou?
Inu sami je tam zapíšeme instrukcí x≤t v kroku 026 našeho programu a zapsali jsme je, protože zůstaly na zobrazovači od kroku 008. Běží-li však program větví B, bude na zobrazovači 0, protože sekvencí Pgm 4 E' bude zobrazovač znulován.
Po proběhnutí nahoře uvedené části programu bude v t číslo vydělené obsahem registru 4 – tedy jeho celky, zatímco desetinný zbytek bude na zobrazovači. V původním programu testujeme, zda se tento zbytek rovná 0 pomocí registru t a instrukce x = t. My máme tento registr obsazen, dále musíme tento první běh provést a teprve po něm budeme násobit registr 6 (nezapomeňte: toto byla podmínka, která nám na začátku dovolila zrušit dvě rozdílné konstanty a sice 0,1 a 0,5 a nahradit ji společnou 1).
Z této poněkud bezvýchodné situace si velice elegantně pomůžeme pomocí instrukce **Signum.** Tato instrukce je na kalkulátoru realizována pomocí operace 10 (**Op 10**). Instrukce Signum pracuje následovně: je-li v okamžiku vydání Op 10 na zobrazovači jakékoli kladné číslo, toto číslo zmizí a na zobrazovači bude 1, je-li tam záporné číslo, po Op 10 bude –1 a je-li tam 0, bude po Op 10 na zobrazovači také 0. Když to tedy shrneme, tak s použitím Op 10 bude na zobrazovači 0 je-li zkoumané číslo dělitelné beze zbytku, a 1 bude-li při dělení zbytek.
Tím také odpadne příkaz k návratu, vyjde-li zbytek 0. Stačí nám totiž, abychom výsledným číslem vynásobili paměť, kde se nám ukládá 2^{n-1} (10^{n-1}) a nemusíme program vracet. To ponecháme již výše zmíněnému testu „je ještě co převádět?"

Teď tedy můžeme udělat celý program. Začátek jsme si již probrali, nezapomeňme však, že dodatečně vypadlo CP a instrukci STO 05 jsme nahradili instrukcí x≥t. Skončili jsme návěstím Lbl D. Tím tedy začneme:
Začátek výpočetní části programu – návěstí **Lbl D.**
Vypíšeme číslo X **x≥t.**
X vydělíme konstantou (2; 10), která je vložena v paměti R4 **: RCL 4.**
Naznačení „předstíraného" součtu, čili vlastně přepis do paměti HIR **+.**
Oddělení části za desetinnou tečkou **Int.**
Vložením tohoto čísla do t **x≥t.**
Touto instrukcí se současně na zobrazovači objeví 0 (2). Dále dokončíme „předstíraný" součet čili výpis z paměti HIR **=.**
Tím se na zobrazovači objeví původní vydělené číslo (+2), ze kterého odtrhneme celky a ponecháme desetinný zlomek **INV Int.**
Nyní provedeme operaci Signum **Op 10.**
Výsledkem vynásobíme paměť s číslem $2^{n-1}(10^{n-1})$ **x RCL 6 =**
a výsledek přičteme do střádací paměti, kterou si zvolíme **SUM 3;**
dále zvětšujeme obsah paměti pro 2^{n-1} (10^{n-1}) vynásobením konstantou z R2 **RCL 2 Prd 06**
a provedeme test „je ještě co převádět?" **0**
INV x = t
Je-li odpověď „ano", pak směřujeme skok na začátek výpočtové části, kde máme návěstí. Adresa bude **D.**
Je-li odpověď „ne", vypíšeme střádací paměť **RCL 3**
a zastavíme program **R/S.**
Všimněte si, že jsme několikrát použili příkazu D, tedy vlastně SBR D, jenže nikde po použití D jsme nedali instrukci INV SBR, takže jsme se nedopustili chyby.

Když spočítáme kroky, vidíme, že jsme ušetřili 10 kroků (čili asi 15 %) a to je dost. Navíc se nemusíme při spouštění programu starat o to, co momentálně je v pamětech nebo v registru t.

Převod dekadických čísel na binární a naopak	*TI-58*
000	*Lbl C CMs x≤t 1 0 STO 02 2 STO 4 Op*
012	*26 INV SBR Lbl A C D Lbl B C Pgm 4 E'*
024	*Lbl D x≤t ÷ RCL 4 + Int x≤t = INV Int Op*
037	*10 x RCL 6 = SUM 3 RCL 2 Prd 06 0*
049	*INV x=t D RCL 3 R/S*

Definitivní tvar programu

Domnívám se, že i na tomto zkráceném programu by se ještě něco dalo zkrátit, ale to už by nebylo asi nic pro začátečníky. Většinou se nejvíc ušetří tím, že použijeme jiný postup, ale šlo mi o to zkrátit program při zachování stejného matematického postupu. Aby byl program úplný, je nutno ještě udělat obslužnou tabulku. Nevěřili byste, za jak krátkou dobu zapomenete obsluhu programu, který jste chvíli nepoužívali. Tabulka může vypadat různě, ale doporučuji přidržet se způsobu, který je používán v návodu k TI 58/59.

Poř. číslo	Zadat	Stisknout	Zobrazovač
1	Číslo v dekadické formě N_D	A	N_B
2	Číslo v binární formě N_B	B	N_D

Činnosti 1 a 2 možno vyvolat nezávisle na sobě.

U složitějších případů se vyplatí uvést příklad, aby byla kontrola, zda byl program správně zapsán, zde je to zbytečné, kontrola je snadná přímo z hlavy.

Závěrem vám chci popřát mnoho zdaru a hlavně – nebojte se experimentovat, ten moudrý skřítek nemůže bohužel z kalkulačky vylézt, aby nám naplácal přes prsty, když něco spleteme a tak se vyplatí vzít rozum do hrsti.

Literatura

[1] *Sedláček, J.:* Využití podprogramů standardního modulu u TI 58/59. Sdělovací technika 11/1981.
[2] *Bíňovec, J.: Mrázek, J.:* Standardní softwarový modul pro TI 58/59. Návod k používání. Praha 1978.