Petr JIŘÍČEK, prom.mat.
Závod výpočetní techniky OKD, Ostrava

GENERÁTORY, PŘEDNOSTI A NEVÝHODY TÉTO TECHNIKY

Ú v o d

V referátu se zabývám univerzálními programy vůbec a speciálně generátory programů. Omezuji se na oblast programování vědeckotechnických úloh, ekonomických výpočtů a pod. V této oblasti totiž často sám autor programu anebo jeho blízcí spolupracovníci jsou ti, kdo pak převážně program používají. Oni tedy také rozhodují, jak má být program obecný.

Možná, že tvrzení a závody, které zde vyslovím, platí i v jiných oblastech programování.

Jistě jsme se všichni setkali s obecnými univerzálními programy. Některé z nich byly schopny splnit to, co jsme od nich očekávali, jiné ne. V první kapitole se pokouším najít příčinu selhání mnohých obecných programů v praxi. Soudím, že je to proto, že byly psány jako obecné předčasně. Je lépe začít určitý typ problému řešit pomocí jednoúčelových speciálních programů. Tím získáme zkušenost a správněji pak koncipujeme příslušný obecný program.

Obecný program tedy píšeme ve zralejším stadiu, když již umíme problém programovat. V tomto stadiu je velmi přirozený takový další postup, že naprogramujeme toto dobře zvládnuté

programování. Vytvoříme generátor.

Ve druhé kapitole srovnávám vlastnosti generátorů s vlastnostmi obecných interpretačních programů. Za přednosti generátorů pokládám větší univerzálnost, snadnější ladění, větší rychlost samotného zpracování. Nevýhodou je nutnost nové kompilace pro nové parametry.

Ve třetí kapitole se zabývám několika poznatky z oblasti programování generátorů. Čtvrté kapitola informuje o podmínkové kompilaci (preprocessor) a o jejím vztahu ke generátorům.

1. U n i v e r z á l n o s t

Někdy programátor musí sám volit míru univerzálnosti, obecnosti programu, který má napsat. Je to časté třeba při řešení výzkumných úkolů, ekonomických studií a podobně. Předpokládejme, že potřebujeme napsat program, o němž víme, že jej budeme používat jen po krátké období.

Uvedu příklady:

- Zpracování sociologického výzkumu. Chceme vytvořit a statisticky zpracovat větší množství kontingenčních tabulek. Známe dotazník a žádané tabulky (kombinace otázek).
- Máme sledovat nějakou akci pomocí síťového grafu. Graf předem známe.
- Potřebujeme vyhodnotit řadu variant důlní dopravy pro určité patro určitého dolu. Na vyhodnocovacím algoritmu zde nezáleží. Může jít o simulační výpočet, o prosté použití několika kalkulačních vzorců atd.

Předpokládejme, že jde o tvůrčí záležitost, nikoliv o jednoduché přepsání algoritmu z literatury. Dále, že jde o program, který bude používat především autor nebo jeho blízký spolupracovník.

Pak podle míry obecnosti existují dva zcela rozdílné způsoby, jak úlohu naprogramovat. Jsou natolik rozdílné, že je možné je rozlišit na první pohled, neboť zcela určují samu zá-

kladní stavbu programu.

První z nich postrádá jakoukoliv obecnost. Budeme takovému programu říkat speciální. Programu napsanému druhým způsobem říkejme naopak obecný. Nyní zcela schematicky uvedu příklady takových programů:

Specialní program pro vyhodnocení sociologického výzkumu

```
      Začátek;
      deklarace včetně inicializace;
      .
      .
B1:   READ ... čtení našeho dotazníku
                - at end  GO TO F1;
      .
      .
      přičtení jedničky na příslušné místo prvé tabulky;
      .
      .
      přičtení jedničky na příslušné místo druhé tabulky;
      .
      .
      přičtení jedničky na příslušné místo poslední tabulky;
      GO TO B1;
F1:   tisk prvé tabulky;
      .
      .
      tisk druhé tabulky;
      .
      .
      tisk poslední tabulky;
      STOP;
```

Uvedeným programem můžeme zpracovat dotazníkový průzkum za předpokladu, že byl použit náš dotazník a že jsou žádány právě naprogramované tabulky.

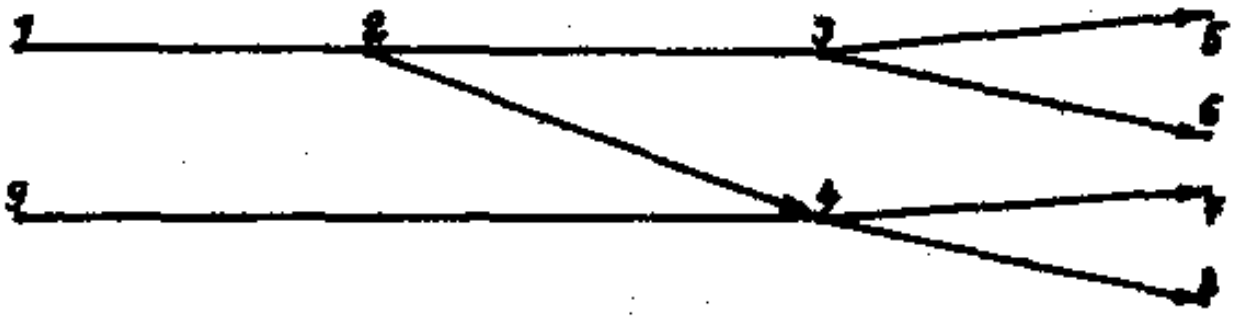
Obecný program pro vyhodnocení sociologického výzkumu

```
      Začátek;
A1:   READ ... čtení tabulky TAB1 udávající formát dotazníku;
A2:   READ ... čtení tabulky TAB2 udávající kombinace otázek,
               t.j. žádané kontingenční tabulky;
      Nulování POLE;
```

```
B1:   READ ... čtení dotazníku
                - at end GO TO F1;
      DO ... smyčka probíhá tabulku TAB1;
             jeden průchod: uložení event. konverse jedné
             položky z dotazníku;
             END;
      DO ... smyčka probíhá tabulku TAB2;
             jeden průchod: přičtení jedničky na příslušné
             místo POLE;
             END;
      GO TO B1;
F1:   DO ... smyčka probíhá tabulku TAB2;
             jeden průchod: vytištění té části POLE, která
             odpovídá příslušné kontingenční tabulce;
             END;
```

Uvedeným programem lze zpracovat průzkum s jakýmkoliv dotazníkem a s jakýmikoliv požadovanými kontingenčními tabulkami.

Příklad vyhodnocení grafu



Dejme tomu máme provést určitý výpočet pro každou spojnici dvou uzlů a pak jiný, sumarizační výpočet pro každou větev vycházející z bodu 1 nebo 9 a končící v bodě 5,6,7 nebo 8. Můžeme si představit, že provádíme kalkulaci pro nějakou dopravní síť. V závěru sumarizujeme náklady na přepravenou tunu z počátečního do koncového bodu.

Příslušný speciální program

```
Výpočet spojnice  1 - 2;
  -"-     -"-     2 - 3;
  -"-     -"-     2 - 4;
  -"-     -"-     3 - 5;
  -"-     -"-     3 - 6;
  -"-     -"-     4 - 7;
  -"-     -"-     4 - 8;
  -"-     -"-     9 - 4;
Sumarizace  1-2-3-5;
  -"-       1-2-3-6;
  -"-       1-2-4-7;
  -"-       1-2-4-8;
  -"-         9-4-7;
  -"-         9-4-8;
STOP;
```

Každá část "výpočet spojnice" obsahuje také čtení vstupních hodnot pro danou spojnici. Každá část "výpočet spojnice" nebo "sumarizace" obsahuje také příslušné tisky. Část "sumarizace 9-4-7" může vypadat třeba následovně:

```
NAKL = NAKL0904 + NAKL0407;
WRITE NAKL;
```

Všimněme si, že program je zcela přímočarý.

Příslušný obecný program

```
READ ... přečtení vstupních hodnot, především těch
         parametrů, které zadávají graf (incidenční
         tabulkou nebo seznamem);
DO   ... smyčka proběhne incidenční tabulku nebo
         seznam zadávající graf;
         jeden průchod pro každou spojnici dvou uzlů;
         provedou se příslušné výpočty;
         .
         .
         .
         END;
```

```
    DO ... smyčka generuje z incidenční tabulky nebo seznamu
           veškeré cesty z počátečního do koncového bodu;
           pro každou z nich provede příslušný sumarizační
           výpočet;
       END;
STOP;
```

Na obou příkladech vidíme, že mezi speciálním a obecným programem vede jasná hranice. Obecný program čte data zobrazující strukturu věcí, na př. popis dotazníku, incidenční tabulky, seznamy. Budeme takovým datům říkat parametry. Speciální program je nepotřebuje číst - má to v sobě zabudováno. Čte pouze konkrétní naplnění dané struktury čísly: soubor nebo podsoubor dotazníků, jednu z variant důlní dopravy pro danou síť a pod. Právě ukázané hranice mezi obecným a speciálním programem je typická - analogické příklady se dají najít v mnoha jiných aplikacích.

Speciální program je vždy jednodušší, více přímočarý. Obsahuje méně smyček. Je čitelnější jak pro programátora tak pro neprogramátora. Méně používá speciálních technik - zpracování seznamů, práce s maticemi a pod. Bývá rychlejší. Může být pro svou přímočarost náročnější na paměť, ale také nemusí.

Obecný program, pokud je dobře napsaný, má vícenásobné použití.

Při rozhodování o obecnosti programu, který má řešit nějakou úlohu, se často dělá chyba ve prospěch obecných programů. Práce na obecném programu, i když je rozsáhlejší, je často zpočátku přitažlivější pro řešitele. Jestliže se navíc předpokládá, že v budoucnu se vyskytne řada sice nikoliv stejných, avšak velmi podobných úloh, vzrůstá nebezpečí úvah asi takových: "Když už se ten program dělá, napíšeme jej obecně. V budoucnu asi obecný program s podobným zaměřením stejně budeme muset napsat. Ušetříme námahu i prostředky tím, že jej napíšeme již teď."

Není to správná úvaha. Jsem přesvědčen, že hledisko dalších budoucích použití programu má být zcela vedlejší a má být respektováno pouze tehdy, jde-li o nepatrnou práci navíc. Obecný program takového typu, jaký jsem ukázal v příkladech, je mnohem precnější. Často ani není jisté, že v budoucnu jej budeme potřebovat - na základě výsledků současné etapy se může značně změnit koncepce dalšího řešení. A především si musíme uvědomit, že realizujeme určitý nový přístup a že tedy s ním nemůžeme zatím mít dostatek zkušeností, abychom mohli vystihnout to, co by mělo být obecné.

Vraťme se k příkladu - dejme tomu k dopravní síti. Pro různé spojnice se mohou provádět výpočty podle různých vzorců, na př. v závislosti na druhu dopravy. I sumarizační výpočty se mohou pro jednotlivé cesty poněkud lišit. Způsob výpočtu v případě obecného programu musí být pro každou spojnici specifikován vstupem a někde uložen - na př. v určitých bitech slova = prvku incidenční tabulky. Jelikož píšeme obecný program, snažíme se naprogramovat všechny možnosti, které připadají v úvahu. Tak vzniká nepřehledné monstrum. Avšak obvykle velmi brzy se objeví požadavky, se kterými jsme původně nepočítali. Pak se snažíme program "napálit" tím, že používáme určitých vstupních hodnot k jiným účelům, než kterým měly původně sloužit - platí to o různých koeficientech atd. Často také provádíme do programu úpravy, což při jeho složitosti a nepřehlednosti vede k jeho znehodnocení. Postupem času se z naší bývalé chlouby stává postrach, se kterým bychom raději neměli mnoho společného.

Jestliže jsme zvolili cestu jednoúčelového programu, jsme v jiné situaci. Program je brzy hotov a je přehledný. Logicky je velmi jednoduchý. Nevýhodou může být, že vzhledem k jeho přímočarosti se v něm opakují skupiny stejných výroků. Z nich však můžeme vytvořit makra nebo uzavřené podprogramy. Rozsáhlé použití podprogramů není velký ústupek od koncepce jednoduchého přímočarého programu. Přitom jej pomáhá učinit přehledným. Každá specialita (na př. má-li se spojnice 9-8

počítat podle zcela speciálního vzorce) se dá jednotně naprogramovat.

Přes jeho jednoúčelové použití se snažíme napsat takový program kvalitně - to je především přehledně, jasně a srozumitelně. Tím si vytváříme univerzálnější prostředek, než by byl obecný program. Každou další a další podobnou úlohu sice programujeme znova - avšak většinu opisujeme ze starších podobných programů, ve kterých se dobře vyznáme. Přitom pečlivě promýšlíme ty části, které jsou tentokrát nové.

Jestliže se původní předpoklad ukázal být pravdivým a skutečně je velká potřeba výpočtů daného typu, teprve pak se stává z neustálého programování téže úlohy skutečná zátěž - jednotvárná mechanická práce. Teprve nyní je čas k napsání jednoho univerzálního programu. Jsem přesvědčen, že to je cesta, kterou vznikají kvalitní, skutečně univerzální programy.

Můžeme to shrnout: Dobrý univerzální program můžeme vytvořit teprve tehdy, jestliže již máme zkušenosti s několika jednoúčelovými programy, které jsme pro obdobné úlohy napsali. Z programování takových úloh se stala zatěžující mechanická rutina. Právě tu má univerzální program odstranit.

2. Interpretátory a generátory

Existují přinejmenším dva různé přístupy, jak napsat dobrý univerzální program.

První přístup představuje onen obecný program, který byl popsán v příkladech. Předpokládám však, že není napsán předčasně, nýbrž po nabytí zkušeností s daným typem úlohy. Takový program čte jako vstupní data parametry, t.j. zobrazení samotné struktury, na př. popis dotazníku nebo incidenční tabulku grafu. Mimoto čte naplnění této struktury čísly - čte dotazníky, technické parametry, ohodnocení jednotlivých spojnic grafu a pod. Program obsahuje řadu smyček, ve kterých prohlíží přečtené struktury a příslušné položky interpretuje. Jelikož vlastní zpracování probíhá neustálou

interpretací příslušných uložených parametrů, nazýváme programy tohoto typu interpretátory nebo obecné interpretační programy. Rozdíl mezi takto vzniklým interpretátorem a předčasně napsaným obecným programem, kterým jsem se zabýval v první kapitole, spočívá právě jen v rozdílu zkušeností a znalostí úlohy.

Druhý přístup, kterým můžeme docílit univerzálnosti, vyplývá bezprostředně ze závěru předchozí kapitoly: Jak bylo uvedeno, z programování dané úlohy se stala dobře zvládnutá mechanická záležitost. To je pro zralé stadium řešení úlohy typické. Postup programování daného typu úlohy dovedeme dobře popsat. Proto je zcela přirozené, že jej můžeme naprogramovat. Tak vznikne programující program - generátor. Generátor programuje to, co jsme předtím programovali sami: speciální, jednoúčelové, logicky jednoduché programy. A programuje tak, jak jsme předtím programovali sami: ve stejném jazyku, stejným postupem.

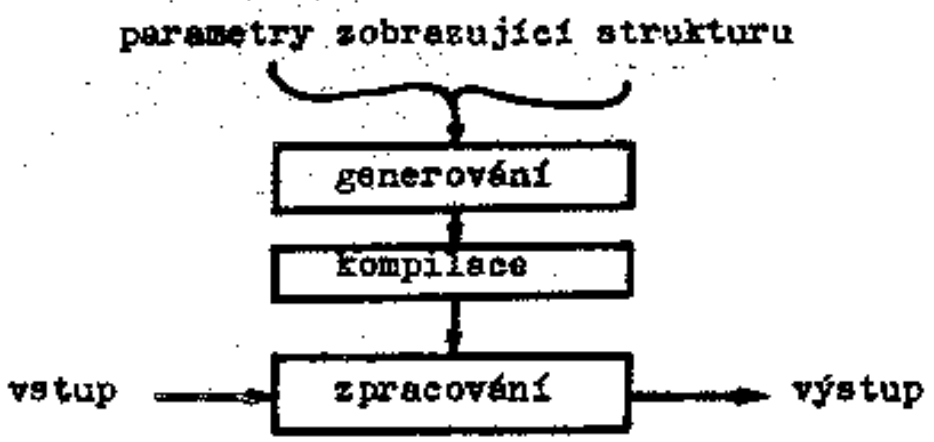
U generátoru jsou odděleny parametry - data, o nichž jsme říkali, že zobrazují strukturu, od dat, kterým jsme s větší či menší nepřesností mohli říkat číselné naplnění úlohy. První data jsou vstupem generátoru, druhá jsou vstupem generovaného programu.

Zvláštní, extrémně složitý případ je implementace programovacího jazyka. Opět jde o protiklad interpretační program - generátor. Generátorům se zde říká kompilátory a jejich vstupními daty jsou zdrojové programy. Pro nejrozšířenější čtyři jazyky se používá výhradně kompilátorů, avšak některé speciální jazyky (LISP) jsou jazyky interpretační nebo obojí. Uvedená problematika však už je zcela mimo sféru zájmu tohoto referátu. Nebudeme se zabývat ani jinými typy (obvykle firemních) generátorů, které vytvářejí přímo object - kód (sort/merge a pod.). Vytváření zdrojového kódu je nesrovnatelně přístupnější. Kromě toho kompilátor sám pro nás udělá pozoruhodný kus práce.

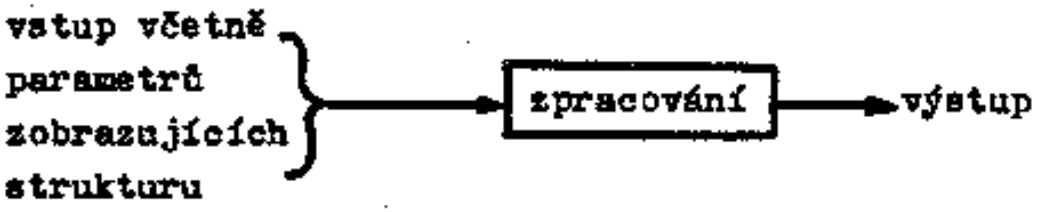
Srovnejme z několika hledisek generátory a obecné interpretační programy:

2.1 Základní schema práce

Generátor:



Interpretační program:



2.2 Strojový čas

Srovnáme-li generátor s interpretačním programem, vidíme na první pohled, že pro velmi malé výpočty je na tom interpretační program lépe. Rozsah samotného výpočtu je totiž zanedbatelný (pak je i generování zanedbatelné), takže generátor má větší spotřebu času o kompilaci.

Vlastní zpracování je pro generátor rychlejší. Nemusí při něm totiž neustále interpretovat tabulky, seznamy a pod. Generátor se vyplatí všude tam, kde má vygenerovaný kód vícenásobné použití, tedy v těchto případech:

- jde o pracný výpočet, o bohaté opakování základní smyčky: buď se zpracovává velký soubor (na př. daňových tazníků) nebo se smyčka hodně opakuje z jiných důvodů (na př. u dopravní sí-

tě používáme stochastickou simulaci většího časového úseku).

- jde o více výpočtů na stejné struktuře. Z každé sady parametrů vygenerujeme program a zkompilovaný jej uložíme do knihovny. (Na př. vygenerujeme program pro síťový graf jedné akce.) Programem můžeme zpracovat řadu vstupů (na př. stav akce každého dne, sledujeme-li její průběh).

2.3 Nároky na paměť

Zásadně není ani jeden z obou přístupů podstatně lepší. Různé aplikace přinášejí různé výsledky. Působí zde dva protichůdné faktory:

- plus pro generátory: zpracování je rozloženo do tří kroků - generování, kompilace, vlastní zpracování. Proto nároky na paměť každého ze tří programů mohou být nižší než u interpretátoru, který vše provádí v jednom kroku. Nároky kompilátoru nemůže uživatel ovlivnit. Jako špičkový firemní program se však kompilátor musí "chovat slušně".)
- minus pro generátory: Generátor zobrazí strukturu danou parametry v redundantní formě kódu, instrukcí generovaného programu. Generovaný přímočarý program může mít vysoké nároky na paměť.

2.4 Programování

Zde je srovnání obzvlášť zajímavé. Zkušenost učí, že programování obou přístupů je překvapivě podobné. Programování generátoru a interpretátoru přináší pro stejný typ úlohy stejné problémy.

Jak bylo řečeno, interpretační program čte parametry a ukládá je do tabulek, seznamů a podobně, aby je pak při výpočtu interpretoval. Totéž musí provádět generátor. Generátor tyto parametry rovněž interpretuje, interpretuje je při generování vlastního zpracujícího programu. Z toho plynou stejné problémy ukládání parametrů do struktur v paměti, tytéž problémy prá-

ce s nimi. Na př. ukládáme-li je do struktur vyžadujících rekursivní zpracování, v obou případech použijeme rekurse. Použije-li ji generátor, má to tu výhodu, že ji nemusí použít vlastní zpracující program. Ten by ji musel použít vícekrát. Rekurse je pro stroj náročná věc. Vyžaduje dynamické přidělování a uvolňování paměti, proto má značnou spotřebu času centrální jednotky. Proto úlohy rekursivní povahy raději řešíme pomocí generátoru. (Hledání půlením, některé úlohy pracující s grafy a pod.). Z hlediska pracnosti a stylu programování není však zde významnějšího rozdílu.

Použití generátoru může tedy sloužit někdy jako prostředek k využití elegantních programovacích metod, aniž by přitom došlo k většímu negativnímu dopadu na výkonnost programu.

2.5 Ladění

Ladění generátoru je snadnější než ladění interpretačního programu. Rovněž ladění parametrů pro generátor je snadnější než ladění parametrů pro interpretační program. Důvodem k tomu je v obou případech existence výpisu generovaného programu. Obdobný mezičlánek u interpretačního programu chybí.

2.6 Další poskytnuté možnosti - pružnost, univerzálnost

V tomto bodě nemůže žádný interpretační program konkurovat obdobnému generátoru. Důvodem pro to je skutečnost, že cyklus generování - zpracování - v sobě zahrnuje použití kompilátoru - nositele nezměrného bohatství určitého programovacího jazyka.

Proto se vždy snažíme napsat generátor tak, aby uživateli umožnil vkládat úseky kódu v tom jazyku, do kterého se generuje /4/. Na př. předepíšeme vstupní data pro generátor následovně: nějakým jednoduchým způsobem rozlišíme příkazy pro generátor od příkazů v cílovém programovacím jazyku. Dejme tomu předepíšeme, že příkazy pro generátor mají mít v první

pozici tečku. Ostatní řádky generátor pokládá za příkazy v cílovém jazyku (do kterého se generuje) a předává je beze změny na výstup pro kompilaci. Generátor necháme číst parametry pomocí vhodné procedury:

```
ČTENÍ :  READ štítek;
         má v první pozici tečku ?
         jestliže ano, RETURN;
         zapiš štítek na výstup;
         GO TO ČTENÍ;
```

V tomto případě je vlastně použití generátoru rozšíření programovacího jazyka vhodné pro určitý typ úlohy. O tom však uživatel-neprogramátor nemusí ani vědět - píše pouhé vstupní parametry, tak jak je má předepsány.

Shrnutí druhé kapitoly: Existují přinejmenším dva různé přístupy, jak napsat univerzální program. První z nich jsou interpretační obecné programy, druhým jsou generátory.

Srovnáme-li oba přístupy, mají generátory následující výhody: vlastní zpracující program vytvořený generátorem je obvykle rychlejší, generátory jsou nesrovnatelně univerzálnější (poskytují rozšíření programovacího jazyka), někdy umožňují použití elegantních programovacích metod bez vážnějšího dopadu na spotřebu strojového času, lépe se ladí a umožňují pohodlnější ladění parametrů.

Programovací práce jsou pro oba přístupy zhruba stejné.

Nevýhodou generátorů je nutnost kompilace pro každé generování.

3. Programování generátorů

3.1 Otázka jazyka

Na jazyku, ve kterém je psán samotný generátor, v podstatě tolik nezáleží. Snad může být někdy vhodné použít tentýž jazyk pro generátor i pro generovaný program. To proto,

že při čtení generátoru myslíme současně na oba programy (generující i generovaný).

Volba jazyka generovaného programu je naopak velmi důležitá. Uplatňují se při ní dvě protichůdná hlediska:

- z hlediska úspory strojového času je nejvhodnějším jazykem assembler. Platí to jednak o čase kompilace, obvykle větším než čas generování, jednak o čase samotného zpracování;
- z hlediska možností a jednoduchosti generátoru je pro generovaný program naopak vhodný problémově orientovaný jazyk. Ukáži to na příkladu:
  Jestliže generujeme výrok

  IF X < Y THEN GO TO LAB1;

  můžeme předpokládat, že kód obsahující deklaraci proměnných X, Y dosadil přímo uživatel a nemusíme se ani ptát na jejich atributy. Kdybychom chtěli obdobný výrok generovat v assembleru, museli bychom zjistit, jak jsou X, Y deklarovány, neboť instrukce srovnání dvou položek je jiná pro binární čísla, jiná pro desítkové čísla, jiná pro znakové řetězce.

3.2 Některé další poznámky k programování generátoru

"Generátor obsahuje zárodečné segmenty kódu, t.zv. skeletory, které obsahují modely programové logiky v nespecializované formě. Generátor vybírá správné segmenty, upravuje je, skládá dohromady a takto produkuje vytvářený program." /1/

Důležité je, že uvedená funkce generátoru musí být z něho patrná. Můžeme necnat vyčnívat poněkud vpravo textové konstanty obsahující kód. To usnadní paralelní sledování logiky obou programů (generujícího a generovaného). Můžeme ty části skeletonů, textových konstant, které mají být přepsány, obsadit určitými znaky. Na př. tam, kde má být číslo uzlu, píšeme vždy '&&'. Ulehčí to další programování ('&&' nahražuje všude generátor aktuálním číslem uzlu) a ulehčí to později sledování logiky.

4. Podmínková kompilace

Generátor a interpretační obecný program nejsou jediné dvě možnosti univerzálního řešení programu. Existuje další, třetí možnost. Můžeme si ji představit někde uprostřed mezi prvními dvěma. Umožňují ji některé jazyky.

Mluvím o tak zvaných předkompilátorech (podmínková kompilace, preprocessor).

Přímo ve zdrojovém programu mohou být umístěny speciální, od ostatního kódu rozlišitelné výroky. Tyto výroky řídí práci předkompilační fáze kompilátoru. Pomocí nich se může zdrojový program modifikovat v závislosti na několika zpočátku zadaných údajích. Výsledek předkompilační fáze teprve vstupuje do kompilace. Může přitom být pokaždé zcela jiný.

Části zdrojového programu mohou být změněny na př. následujícími způsoby /2/:

- může být změněn kterýkoliv identifikátor;
- programátor může určit, které úseky jeho programu mají být kompilovány;
- do zdrojového programu mohou být začleněny části zdrojového kódu uložené v uživatelově knihovně;

Příklady předkompilátorů:

Preprocessor PL/I a Conditional Assembler u IBM 370. /2/,/5/

Existují předkompilátory použitelné pro různé jazyky /6/.

Na podmínkový kompilátor (předkompilátor) se můžeme dívat jako na zcela obecný, firmou dodaný generátor.

S generátory má společnou tu vlastnost, že pro různé varianty úlohy je třeba znovu program kompilovat.

S interpretačními obecnými programy má použití podmínkové kompilace společné to, že opět jde o jeden program přímo řešící určitou úlohu (ovšem až po kompilaci). Odpadá bývalá dvojakost: generující program/generovaný program.

Parametry zadávající strukturu úlohy tvoří vlastně někde na začátku kus programu. Proto podléhají obecně platné syntaxi. A to může být značně omezující činitel.

Řešením může být kombinace generátoru a podmínkové kompilace. Generátor je zde triviální program přepisující parametry z přijatelné formy do žádané syntaxe.

Příklad: Obecný sociologický program napsaný na VÚEPE v Ostravě se skládá z modulů A2H1, A2H2, A2H3. Uživatel je skompletuje spolu se svými parametry:

```
%INCLUDE  A2H1;   /* ZAČÁTEK */
 parametry popisující dotazník;
%INCLUDE  A2H2;
 parametry popisující žádané tabulky;
%INCLUDE  A2H3;   /* KONEC */
```

Parametry popisující dotazník jsou vlastně rozpisem struktury v PL1, což je v pořádku.

```
DECLARE  1  V  BASED(P),
              2  X1  CHAR(3),
              2  X2  CHAR(1),
              :
              :
```

Parametry popisující žádané tabulky (každá tabulka odpovídá dvěma položkám dotazníku) se píší velmi těžkopádně:

```
%VA1 = 'V.X1'; %VB1 = 'V.X2'; %TX1 = '(''V.X1*V.X2'')';
%VA2 = 'V.X2'; %VB2 = 'V.X5'; %TX2 = '(''V.X2*V.X5'')';
 :
```

Jednodušší způsob zápisu již preprocessor nedovolil. Situaci vyřešil triviální generátor, umožňující psát

```
.  V.X1          V.X2
.  V.X2          V.X5
   :
```

Z těchto řádků generuje původní těžkopádné řádky.

Závěr referátu

Generátor je stejně přirozené a přinejmenším rovnocenné řešení jako obecný interpretační program.

Literatura

/1/ Jordain, Philip B. - Breslau, Michael :
Condensed Computer Encyclopedia

/2/ PL/I(F) Language Reference Manual
IBM System/360 Operating System, GC28-8201-3

/4/ Jiříček, P. : Vytváření programovacích jazyků uživatelem / maa '72, str. 82, číslo 3

/5/ OS/VS and DOS/VS Assembler Language
IBM 370 VS1 Release 2, GC33-4010-1.

/6/ Lacko, Branislav : Použití makroinstrukcí při racionalizaci programování / maa '73, str. 445