Ing. Bohumil Vondráček
Michal Kretschmer prom. mat.
MÚZO Praha
(část Základní principy, Závěr)

RNDr Pavel Drbal CSc
VÚMS Praha
(část Inverze programu)

Jiří Suchomel prom. mat.
Orgaprojekt Praha
(část Zpětné sledování)

TECHNOLOGIE STRUKTUROVANÉHO PROGRAMOVÁNÍ

1. Základní principy

Úvod

Snahou autorů tohoto příspěvku je seznámit širší programátorskou veřejnost se základními principy a technikami návrhu programů, uveřejněných v knize M. A. Jackson : Principles of Program Design, Academic Press, London, 1975.

Presentovaná technologie návrhu programu představuje vlastně první konkrétní pracovní postup (recept), jak vytvářet dobře strukturované programy podle obecných zásad strukturovaného programování. Vzniklé programy vyhovují jak z hlediska spolehlivosti, jednoduchosti, efektivnosti a srozumitelnosti, tak z hlediska snadné verifikovatelnosti a schopnosti jednoduché údržby. Tato metodika je vhodná pro řešení úloh jak z oblasti hromadného zpracování dat, tak i pro řešení problémů vědecko-technických. Lze ji rovněž aplikovat pro návrh dávkově zpracovávaných úloh i pro on-line systémy.

Standardizovaný pracovní postup s výhodou uplatní nejenom zkušení programátoři, ale zejména má velký význam pro začátečníky, kteří obvykle nejsou schopni správně aplikovat obecné principy strukturovaného programování. Není totiž jednoduché

provést správnou dekomposici řešeného problému a tak vytvořit hiearchické struktury odpovídající jednotlivým úrovním podrobnosti. Rovněž tak zásada jednoduchosti (tj. používání pouze třech základních řídících struktur - sekvence, selekce a iterace) ještě neříká nic o tom, které ze základních konstrukcí mají být použity a jak mají být spolu propojeny. Důsledné použití této technologie vede k tomu, že vlastní kódování programu nebo jeho dílčí části je odsunuto do okamžiku, kdy je naprosto jasná jeho struktura. Hlavním přínosem je tedy metoda jak provádět návrh programu, tj. správnou dekomposici a syntézu řešeného problému. Vzhledem k tomu, že vychází z obecných principů strukturovaného programování zahrnuje tato technologie i všechny jeho výhody. V důsledku omezeného rozsahu referátu nebylo možné zařadit více příkladů, což je na škodu věci.

Struktury a jejich komponenty

Člověk je schopen uvažovat jasně a přesně pouze o nepříliš složitých objektech. Proto je nucen při popisu struktury složitějšího objektu postupovat po krocích, tj. provádět postupnou dekomposici objektu. Výsledkem tohoto postupu by měla být hiearchická struktura složená z jednotlivých komponent, kterou lze znázornit stromovým grafem. Jde tedy o techniku shora-dolů. Objektem může být program, dílčí úsek programu, ale také data, např. datový soubor, tabulka ve vnitřní paměti apod.

Komponenty, ze kterých je struktura složena, jsou dvojího druhu :

- elementární (tj. takové, které již nelze nebo není účelné dále dělit a tudíž neobsahují žádné další části)
- složené (tj. takové, které jsou složeny vždy z dalších komponent).

Je známo, že pro návrh struktury stačí tyto tři složené komponenty :

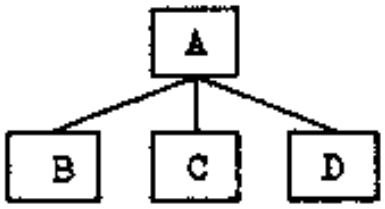
- sekvence (obsahuje dvě nebo více komponent, z nichž se každá vyskytuje jedenkrát a v uvedeném pořadí)
- iterace (obsahuje jedinou komponentu vyskytující se nulkrát nebo vícekrát)

- selekce (obsahuje dvě nebo více komponent, z nichž se vyskytuje právě jedna).

Všechny uvedené komponenty mají jeden vstup a jeden výstup. Dále uvádíme příklady jejich zápisu.

sekvence :

strukturální diagram



zápis dat v prog. jazyce

```
01  A.
    03  B   PIC  X(20).
    03  C   PIC  S9(4) COMP.
    03  D   PIC  X(7).
```

zápis programu v pseudokódu

```
A sequence
  X := X + Y;
  do C;
  do D;
A end
```

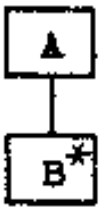
zápis programu v prog. jazyce

```
PA-SEQ.
    ADD Y TO X.
    CALL PC.
    PERFORM PD.
PA-END.  EXIT.
```

Komponenta A je sekvencí komponent B, C a D.

iterace :

strukturální diagram



until c1

zápis dat v prog. jazyce

```
05 A.
    07  B OCCURS 10 PIC X(9).
```

zápis programu v pseudokódu

```
A iteration until c1
  do B;
A end
```

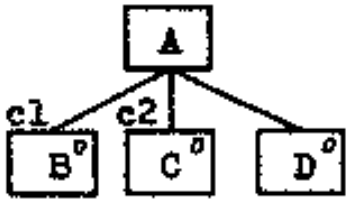
zápis programu v prog. jazyce

```
PA-ITER.
    PERFORM PB UNTIL c1.
PA-END.  EXIT.
```

Komponenta A je iterací komponenty B. Komponenta B se může vyskytnout vícekráte nebo se nemusí vyskytnout vůbec (v závislosti na podmínce c1).

selekce :

strukturální diagram



zápis dat v prog. jazyce

```
03  A.
    05  B  PIC 9(10).
    05  C  REDEFINES B PIC X(10).
    05  D  REDEFINES B PIC S9(10).
```

zápis programu v pseudokódu

```
A select c1
  do B;
A or c2
  do C;
A or
  do D;
A end
```

zápis programu v prog. jazyce

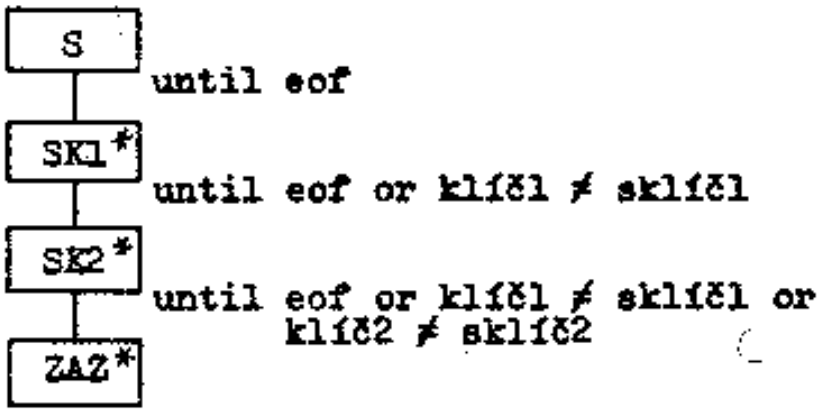
```
PA-SLCT.
    IF NOT c1 GO TO PA-OR1.
    PERFORM PB.
    GO TO PA-END.
PA-OR1.
    IF NOT c2 GO TO PA-OR2.
    PERFORM PC THRU PC-END.
    GO TO PA-END.
PA-OR2.
    PERFORM PD-SEQ THRU PD-END.
PA-END.  EXIT.
```

Komponenta A je selekcí komponent B, C a D. Komponenta B se vyskytne, je-li splněna podmínka c1, komponenta C se vyskytne, je-li splněna podmínka c2, komponenta D v ostatních případech. Některá z komponent selekce může být prázdná. Tím lze označit nepřítomnost dat v datové struktuře nebo nepřítomnost zpracování v programové struktuře.

Při popisu jednotlivých struktur dat nebo programu se užívají strukturální diagramy, pro popis programové logiky je používán pseudokód z důvodu nezávislosti na vlastním programovacím jazyce.

Strukturální diagram je velmi silný prostředek popisu. Lze pomocí něho např. provést zápis dat, jež nelze vyjádřit běžnými prostředky programovacích jazyků. Máme-li např. soubor S obsahující věty se shodným klíčem k1, v rámci nichž jsou opět skupiny vět se shodnými klíči k1 a k2, nelze tento fakt popsat

běžnými prostředky, jež známe z programovacích jazyků, kde je k dispozici pouze aparát pro popis dat ve vnitřní paměti. Strukturálním diagramem znázorníme tuto skutečnost následovně :



Zápis podmínek, jako např. podmínka ukončení iterace, není nutno uvádět při kreslení strukturálních diagramů. Ty se musí uvést při konečné fázi návrhu programu.

Při zobrazování strukturálních diagramů je třeba se v žádném případě nepřizpůsobovat možnostem počítače (rozuměj hardware a software), tj. nebrat ohled na implementaci řešení. Kdykoliv se ukáže, že možnosti počítače nevyhovují, lze navrhnout lepší počítač, který umožní problém vyřešit. Hlavní metodou návrhu je metoda shora-dolů. Přesto se však někdy jeví účelné dílčí problémy řešit metodou zdola-nahoru. V tomto případě je však třeba chápat problém následovně. Počítač, na kterém má být náš problém řešen, není dosti vhodný k jeho řešení z toho důvodu, že elementární operace jsou příliš elementární. Proto jsou elementární operace použity k vytvoření mocnějšího aparátu (počítače); tyto potom k vytvoření ještě mocnějšího atd. až máme počítač, který je pro řešení našeho problému vhodný. Lze tak vytvářet a nezávisle testovat skupiny složitějších elementárních komponent. Příkladem je obecná rutina pro vstup/výstup, komplex rutin pro operace s maticemi apod.

Postup tvorby programu

Presentovaná technologie strukturovaného programování standardizuje proces tvorby programu, jenž lze rozdělit do dvou fází :

I. Konstrukční fáze

1) Nakreslení strukturálních diagramů všech datových souborů, se kterými program pracuje. Mnohdy nejsou všechny soubory explicitně vyjádřeny (např. interní datové soubory - tabulky v operační paměti). Strukturální diagramy dat musí být odrazem řešeného problému. Proto musí vždy předcházet důkladné studium řešeného problému. Podmínky pro ukončení iterací a podmínky v selekcích se do strukturálních diagramů dat zpravidla neuvádějí, lze však např. v selekcích do příslušného bloku uvést text, ze kterého je patrno o jaká data se jedná. Ve strukturách dat by měly být obsaženy i všechny případy chybných dat, které mají být zpracovány. Jedná se o náročnou činnost, která je pro úspěšnost tohoto postupu rozhodující.
2) Odvození strukturálního diagramu programu na základě datových struktur. Pravidla pro toto odvození budou dále diskutována. Po odvození struktury programu je vhodné zkontrolovat, zda jsou v ní obsaženy všechny datové struktury.
3) Zápis seznamu elementárních operací bez ohledu na pořadí jejich provádění a jejich přiřazení ke vhodným komponentám vzniklé programové struktury. Jednotlivé elementární operace na seznamu se očíslují pořadovými čísly a do programové struktury se přiřazují pomocí této číselné odvolávky. Někdy je zapotřebí při přiřazení operace k iteraci nebo selekci zavést pomocný blok, aby nebylo narušeno pravidlo o používání základních řídících struktur. Takto vzniklé komponenty označujeme jménem nadřazené komponenty se sufixem BODY. V tomto okamžiku je vhodné uvést všechny podmínky pro ukončení iterací a podmínky v selekcích. Jestliže se zápis těchto podmínek vejde do nakresleného strukturálního diagramu programu, je možno je zde uvést přímo, v opačném případě nebo jestliže by byla narušena čitelnost a přehlednost této struktury, je vhodné napsat jejich seznam, očíslovat je (např. C1, C2,...) a v programové struktuře se pak na ně pouze odvolávat.

Pozn.: Nedaří-li se nalézt jednoznačně komponentu pro přiřazení elementární operace nebo nelze-li stanovit některou z podmínek ukončení iterace nebo podmínku v selekci, je to známkou chybně navržené programové struktury. To má pak za následek, že je třeba opakovat dosavadní postup.

4) Zápis pseudokódu. Při přepisu strukturálního diagramu programu do pseudokódu jde více méně o formální záležitost. Je však nutno dbát na to, aby byl zápis v pseudokódu přehledný, tj. aby popis komponenty, který na jednom listu papíru začíná, tam také byl dokončen. Má to v podstatě za následek, že je třeba některé vnořené složené komponenty dále nerozepisovat, ale uvést je pomocí do jméno-komponenty. Tím je pouze řečeno, že uvedená komponenta se má provést na tomto místě a že její popis pseudokódem je uveden na jiném listu papíru. V žádném případě tím není naznačen způsob implementace (např. s příkazem PERFORM nebo bez něj). Elementární komponenty se zapisují jako operace vždy přímo. Složené komponenty lze též zapisovat v rozloženém tvaru, pak hovoříme o "in-line" kódování. Do jaké míry lze složené komponenty uvádět v rozloženém tvaru nebo pomocí do, závisí také na volbě stupně podrobnosti.

Pozn. 1 : Prázdným komponentám v selekcích by měl odpovídat prázdný or v pseudokódu.

Pozn. 2 : Podobný princip uvádění složených komponent by měl být aplikován i při kreslení strukturálních diagramů tak, aby byly vždy přehledné.

Pozn. 3 : Přepis do pseudokódu mohou zkušení programátoři v některých případech vynechat.

5) Optimalizace. Problémy optimalizace nejsou v rámci tohoto příspěvku diskutovány. Je však třeba si uvědomit, že ve většině případů vede optimalizace k tomu, že navržená programová struktura již neodpovídá řešenému problému.

6) Volba metody implementace navrženého řešení. Jedná se o volbu cílového programovacího jazyka, konečnou specifi-

kaci dat a pomocných proměnných (USAGE) apod.

II. Produkční fáze

7) Kódování je transformace pseudokódu do cílového programovacího jazyka. Lze zde s výhodou použít preprocesor, je-li tento automatizační prostředek k dispozici, neboť se jedná o formální záležitost. Pro jednotlivé programovací jazyky lze specifikovat transformační pravidla; v další části referátu uvádíme tato pravidla pro jazyk Cobol.

8) Dokumentace vlastně vzniká průběžně při návrhu programu. Strukturální diagramy dat a programu, seznamy operací a podmínek a jejich přiřazení k programové struktuře, právě tak jako pseudokód jsou nedílnou součástí dokumentace. Přitom je nutno dbát o to, aby popisy jednotlivých částí byly vždy přehledné, odrážely skutečně zakódovaný program a aby jim bylo vždy kompletně rozumět.

9) Testování. Při testování se s výhodou využívá stromové struktury programu. Proto stačí, aby program na základě testovacích dat prošel všemi větvemi, aby pro každou podmínku nastaly všechny alternativy, aby všechny případy vstupní datové struktury byly zahrnuty, právě tak, aby byly produkovány všechny komponenty struktury výstupních dat. Technologie strukturovaného programování vede k odstraňování chyb již ve fázi návrhu programu; při dobrém návrhu programu se při vlastním testování odstraňují převážně chyby z nepozornosti a chyby kódování.

Kódování pseudokódu v Cobolu

Aby bylo možné využívat některých dále popsaných technik (inverze programů a spětné sledování) v plném rozsahu, jsou nutná některá omezení při vlastním kódování. Z tohoto důvodu není vhodné připustit ani jednoduché vnoření (tj. použití podprogramů a složených podmíněných příkazů). Např. pro jazyk Cobol to znamená nepoužívat konstrukce PERFORM pro kódování iterace (zejména když v těle PERFORM je použita I/O operace) a nepoužívat konstrukce IF ... ELSE ... při kódování selekcí. Bližší vysvětlení přesahuje rámec tohoto referátu. Přesto však s těmito

omezeními budeme dále počítat. Znamená to tedy, že výše uvedené příkazy jazyka Cobol budou ručně kompilovány. Tento způsob kódování je označován jako bez vnoření (nest-free). Dále uvádíme pravidla pro přepis základních složených komponent do nest-free Cobolu :

```
sekvence :      jméno-SEQ.
                    .
                    .
                    .
                jméno-END.
iterace :       jméno-ITER.
                    IF podmínka GO TO jméno-END.
                    .
                    .
                    .
                    GO TO jméno-ITER.
                jméno-END.
selekce :       jméno-SLCT.
                    IF NOT podmínka-1 GO TO jméno-OR1.
                    .
                    .
                    .
                    GO TO jméno-END.
                jméno-OR1.
                    IF NOT podmínka-2 GO TO jméno-OR2.
                    .
                    .
                    .
                    GO TO jméno-END.
                .
                .
                .
                jméno-ORn.
                    .
                    .
                    .
                jméno-END.
```

Poznámka : Pochopitelně se při této ruční kompilaci uvedených konstrukcí nelze vyhnout použití příkazu GO TO. V těchto případech se však jedná o přísně řízené použití těchto příkazů.

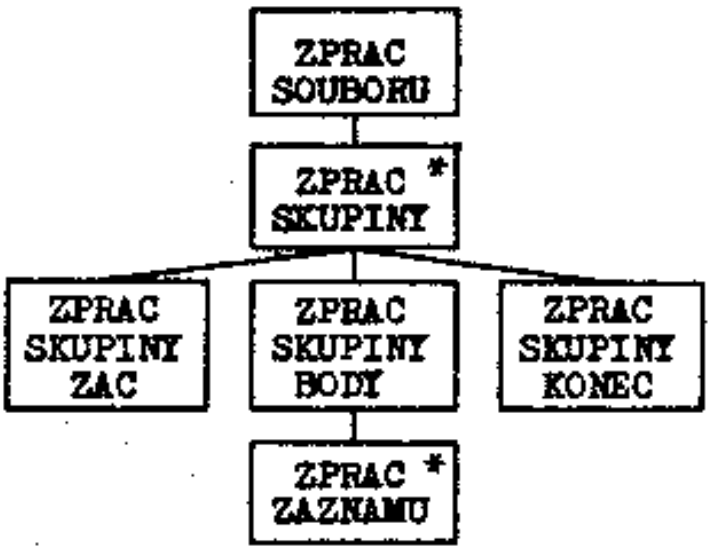
Modelové řešení problému skupinové kontroly

Tento problém je v praxi při hromadném zpracování dat dobře znám. Hlavním přínosem tzv. normovaného programování je právě řešení tohoto problému. Technologie strukturovaného programování zná však pojem změny skupiny pouze jako kriterium pro ukončení iterace. Z tohoto hlediska je model řešení známý z normovaného programování chybný, neboť způsob zpracování skupin

záznamů neodpovídá struktuře dat. Dále uvádíme datovou a programovou strukturu pro problém jedné úrovně klíčů :

datová struktura

programová struktura



Komponenta ZPRAC-SKUPINY-ZAC musí obsahovat příkaz k uložení klíče právě zpracovávané skupiny do operační paměti. Iterace ZPRAC-SOUBORU končí při dosažení konce souboru (eof). Iterace ZPRAC-SKUPINY-BODY končí při dosažení záznamu, jehož klíč je různý od klíče skupiny nebo při eof. Komponenta ZPRAC-SKUPINY-KONEC by např. mohla obsahovat tisk součtu za tuto skupinu.

Předpokladem pro zajištění správné činnosti programu je použití techniky čtení napřed (viz následující oddíl). Jestliže existuje více úrovní klíčů, tj. v rámci jedné skupiny existuje iterace dalších skupin atd., lze pro ukládání klíčů skupiny s výhodou použít podobného principu jak je znám z normovaného programování, čímž se zjednoduší zápis podmínek pro ukončení iterací. Tento postup lze však aplikovat jen v některých případech v závislosti na typech klíčů (USAGE).

Čtení sekvenčních souborů

Při přiřazování operací k programové struktuře se vyskytne v souvislosti se zpracováním sekvenčních souborů otázka "kde mají být čtena data?". Pro ilustraci uveďme tento příklad :

programová struktura

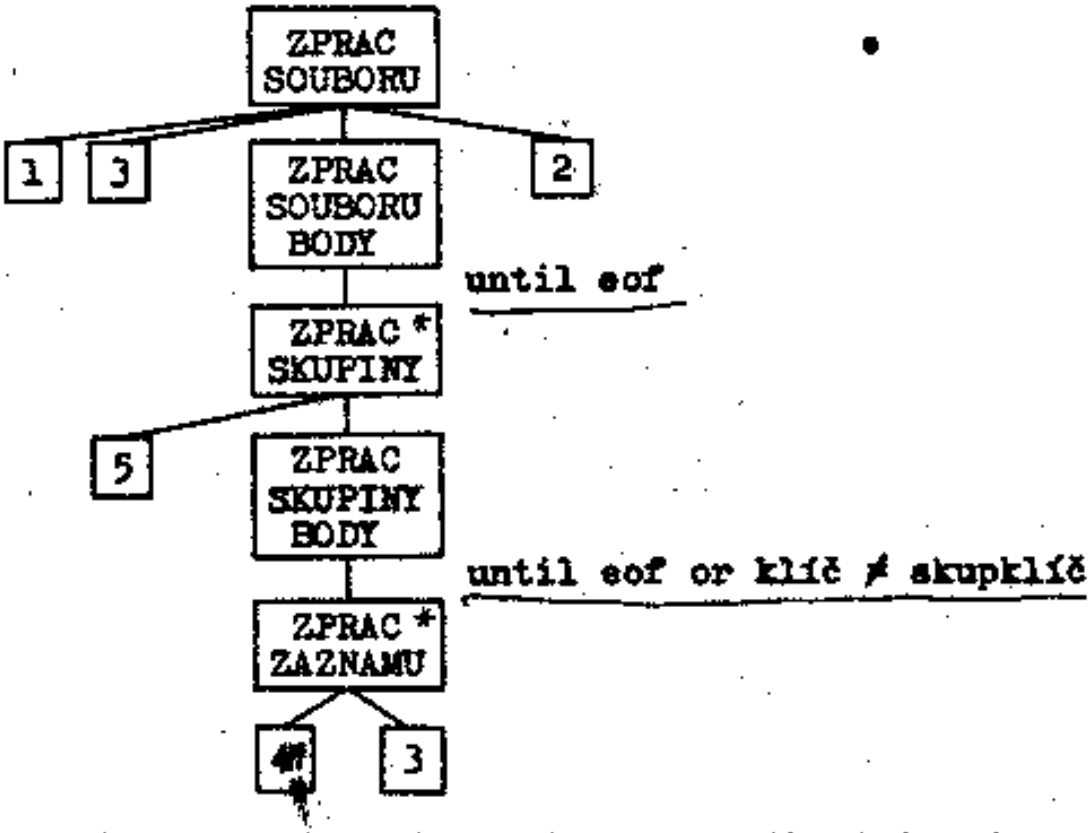
until eof

until eof or
klíč≠skupklíč

seznam operací

1 open souboru
2 close souboru
3 čti data
4 zpracování dat
5 skupklíč:=klíč

Aby bylo možné v programové struktuře provádět porovnání typu "until eof or klíč ≠ skupklíč", musí být oslovované data k dispozici. To lze docílit čtením napřed. Čtení se tedy bude provádět bezprostředně po operaci open a vždy tehdy, kdy je celý záznam zpracován. Postupujeme-li podle této filosofie, bude programová struktura s přiřazenými operacemi vypadat :



Toto pravidlo označované čtení napřed přichází v úvahu při čtení všech sekvenčních souborů a mělo by být v rámci technologie strukturovaného programování důsledně používáno. Nyní se vyskytuje ještě otázka,"jak má být čtení prováděno?". Je totiž nutno zpracovat speciálním způsobem stav konce souboru (eof). Dotaz na eof v programové struktuře předpokládá zvláštní označení,

které se objeví např. jako speciální záznam v oblasti pro čtený záznam a lze se na něj dotazovat. I/O příkazy v mnoha programovacích jazycích signalizují eof jako speciální stav. Např. příkaz pro čtení v Cobolu zná dva výstupy, jeden pro not-eof a jeden pro eof. Abychom mohli v Cobolu zpracovat eof, potřebujeme k tomu použít eof-výstup a v něm generovat eof-záznam. Jsou doporučovány dvě varianty. Existuje-li v záznamu klíč, lze toto pole využít k signalisaci např. tak, že při generaci eof-záznamu se uloží do pole klíče HIGH-VALUE. V případě, že klíč neexistuje nebo HIGH-VALUE je jednou z možných hodnot klíčů, lze pole pro záznam rozšířit o 1 byte a v něm indikovat tento stav (např. 0 pro normální záznam, 1 pro eof-záznam).

U některých úloh může být užitečné dotazovat se na stav většího množství záznamů směrem kupředu (kromě aktuálního záznamu, který se právě bude zpracovávat, ještě dotazy na následující záznamy). Lze pak tuto techniku rozšířit a provádět vícenásobné čtení napřed.

Pravidla pro odvození programové struktury z datových struktur

1) Části datových struktur, které se kryjí, tj. pro něž platí korespondence 1:1, lze sjednotit. Korespondence 1:1 dvou srovnávaných komponent je definována tím, že obě mají shodný počet a pořadí elementů.

datové struktury

programová struktura

2) Jestliže existuje korespondence 1:1 mezi dvěma sekvencemi, sjednotí se do výsledné struktury komponenty podřazené oběma sekvencím v pořadí daném zpracováním.

datové struktury

programová struktura

3) Jednotlivé alternativy v selekcích, které se nekryjí, se vzájemně násobí.

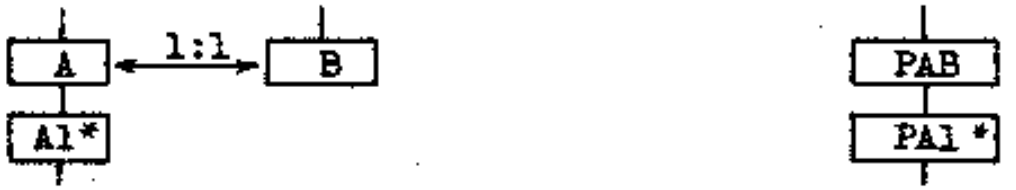
datové struktury programová struktura

4) U alternativ v selekcích, které se částečně kryjí, se sjednotí ty, pro něž platí korespondence 1:1 a zbylé se vzájemně násobí.

datové struktury programová struktura

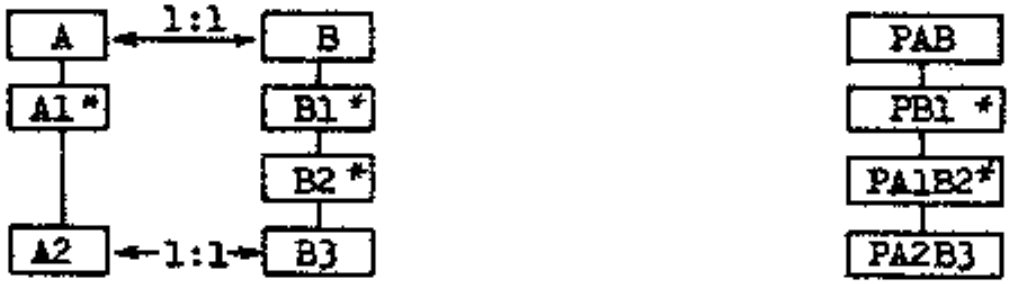
5) Komponenty na spodním konci datových struktur (nad nimi existuje vzájemná korespondence), které se ve druhé struktuře nevyskytují, lze převzít do výsledné programové struktury.

datové struktury programová struktura



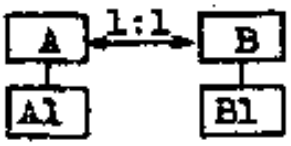
6) Jestliže chybí iterace uprostřed jedné datové struktury, lze ji převzít do výsledné programové struktury za předpokladu, že datové struktury odpovídají níže uvedenému modelu a že počet výskytů A1 v A je roven součinu počtu výskytů B2 v B1 a počtu výskytů B1 v B.

datové struktury programová struktura

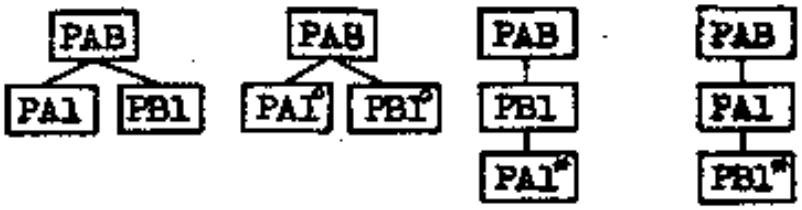


7) Jestliže nelze dvě komponenty, které se nekryjí, vyjádřit jedním z následujících vztahů, existuje mezi strukturami rozpor. Problémům konfliktů struktur je věnován v dalším textu samostatný oddíl.

datové struktury



programové struktury

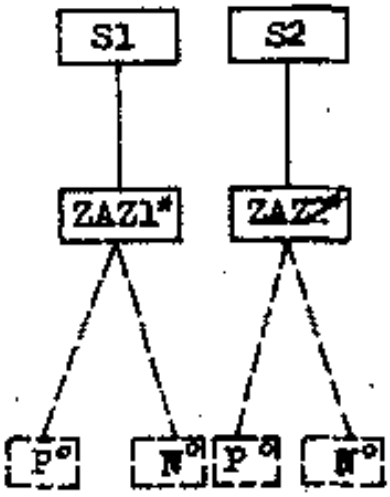


Rozpor mezi strukturami A a B existuje, nelze-li výslednou programovou strukturu vyjádřit jako sekvenci nebo selekci nebo celočíselnou iteraci.

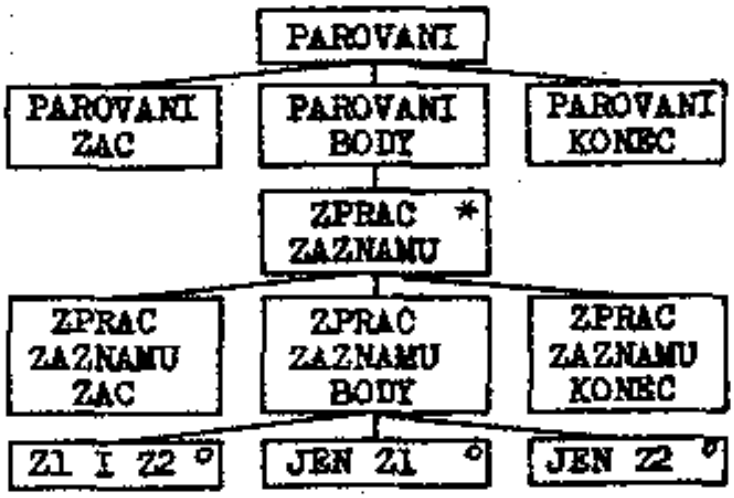
Modelové řešení problému párování sekvenčních souborů

Mějme soubory S1 a S2, které jsou vzestupně setříděny dle nějakého klíče. Pro jednoduchost předpokládejme, že se v žádném ze souborů nevyskytují záznamy o stejném klíči. Má být vytvořen výstupní soubor, jehož obsah je určen tím, zda existují záznamy o shodném klíči v S1 a S2, nebo zda záznam o určitém klíči existuje pouze v souboru S1 resp. S2. V praxi se jedná nejčastěji o slučování sekvenčních souborů do jednoho výstupního souboru nebo změnové řízení.

datové struktury



programová struktura



Výsledná programová struktura vychází z pravidla o násobení alternativ v selekcích. Na úrovni S1 a S2 existuje korespondence 1:1; na úrovni ZAZ1 a ZAZ2 tato korespondence neexistuje, neboť v kterémkoliv ze souborů může existovat záznam o klíči, jenž není přítomen v druhém souboru. Lze však oba soubory doplnit o takovéto chybějící fiktivní záznamy, takže potom existuje korespondence i na úrovni ZAZ1 a ZAZ2. Do datových struktur obou souborů se doplní selekce, kde P znamená záznam přítomen, N záz-

nam nepřítomen.

Komponenta PAROVANI-ZAC obsahuje operace otevření souborů, čtení záznamu z obou souborů (čtení napřed) a operaci MINKLIC:= min(KLIC1,KLIC2), kde KLIC1 resp. KLIC2 jsou hodnoty klíčů přečtených záznamů ze souborů S1 resp. S2. Komponenta PAROVANI-KONEC obsahuje uzavření souborů. Podmínka pro ukončení iterace PAROVANI-BODY je eof-S1 and eof-S2. Komponenta ZPRAC-ZAZNAMU-KONEC obsahuje operaci MINKLIC:=min(KLIC1,KLIC2). Podmínka pro selekci programové komponenty Z1-I-Z2 je KLIC1=MINKLIC and KLIC2=MINKLIC. V rámci této komponenty se vyskytují operace specifické pro případ, že klíče záznamů čtených z S1 a S2 jsou shodné. Na závěr této komponenty se provede čtení záznamů z obou souborů. Podmínka pro selekci programové komponenty JEN-Z1 je KLIC1=MINKLIC and KLIC2≠MINKLIC. Tato komponenta pak obsahuje operace specifické pro případ, že v souboru S2 neexistuje záznam o klíči stejném jako má záznam přečtený z S1. Na závěr komponenty se provede čtení ze souboru S1. Komponenta JEN-Z2 obsahuje operace specifické pro případ, že v souboru S1 neexistuje záznam o shodném klíči jako má záznam přečtený z S2. Na závěr této komponenty se provede čtení z S2.

Pozn. 1: Operace čtení souboru v sobě zároveň zahrnuje signalizaci eof tím, že do pole pro klíč je přenesena hodnota větší než je možný maximální klíč (např. HIGH-VALUE).

Pozn. 2: Podobným způsobem lze řešit úlohu, kdy např. soubor S2 bude soubor s přímým přístupem. V tomto případě odpadne komponenta JEN-Z2. Čtení souboru S2 bude prováděno zpravidla podle klíče, jehož hodnota je obsažena v některém poli ZAZ1. V programové struktuře by se čtení z S2 provádělo v komponentě ZPRAC-ZAZNAMU-ZAC, aby bylo možno vyhodnotit podmínky v selekci.

Pozn. 3: V některých úlohách je množina možných klíčů známá, např. je souvislou číselnou řadou, a může být tudíž požadováno zpracování i pro případ, že záznam o některém klíči z možných klíčů není přítomen v žádném ze souborů S1 a S2. V programové struktuře se to projeví tak, že v selekci přibude ještě další možnost. Komponenta PAROVANI-ZAC pak obsahuje naplnění MINKLIC minimem množiny možných klíčů, v komponentě ZPRAC-ZAZNAMU-KONEC

je prováděna operace MINKLIC:=MINKLIC+1. Podmínkou pro ukončení iterace PAROVANI-BODY bude MINKLIC maximum množiny možných klíčů.

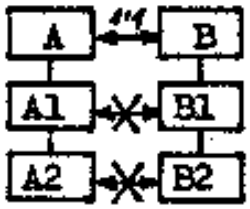
Rozpoznání a klasifikace rozporů struktur a způsoby jejich řešení

Nelze-li podle uvedených pravidel pro odvozování programové struktury z datových struktur některé komponenty datových struktur sjednotit nebo mezi nimi nalézt jeden ze vztahů sekvence, selekce nebo celočíselné iterace, existuje mezi strukturami rozpor. Jestliže se při návrhu programové struktury přehlédne rozpor struktur, lze ho obvykle odkrýt později při přiřazování operací k programové struktuře. Nedaří-li se totiž nalézt pro každou operaci její přirozené místo v programové struktuře nebo nelze-li stanovit podmínky pro iteraci nebo selekci, je to důsledkem neodhaleného rozporu struktur (nebo chybně navržené struktury).

Technologie strukturovaného programování rozeznává následující typy rozporů struktur :

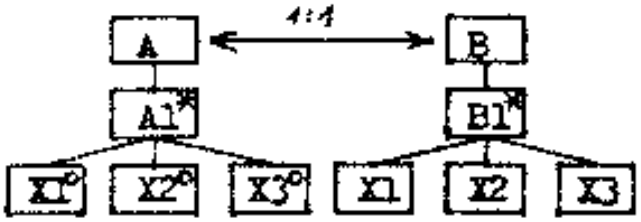
- rozpor setřídění : a) rozpor pořadí
  b) rozpor proložení
- rozpor mezí.

Rozpor pořadí je charakterisován tím, že komponenta A2 je identická s komponentou B2, ale vyskytuje se v rozdílném pořadí. Data A jsou např. tříděny podle jiného klíče než data B. Počet elementů A2 i B2 je shodný. Rozpor pořadí lze řešit dvěma způsoby :



- použitím standardního třídícího programu, který vytvoří mezisoubor, jenž je vhodný k dalšímu zpracování (např. třídění A dle klíče B).
- načtením všech dat A do paměti s přímým přístupem (např. operační paměti). Na tato data lze použít libovolnou strukturu.

Rozpor proložení je charakterisován tím, že A je částečně uspořádáno. Obsahuje komponenty B1, v našem případě trojice X1, X2, X3 v tomto pořadí, avšak jednotlivé elementy v trojicích nemusí následovat bezprostředně po sobě, nýbrž mohou být proloženy

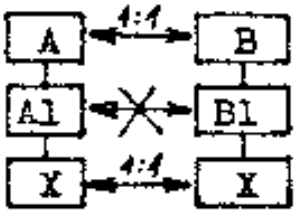


trojicemi jiného B1. Proto je v našem příkladě znázorněna struktura A1 selekcí. Problém však vyžaduje zpracování podle struktury B.

Rozpor proložení lze řešit třemi způsoby :

- tříděním struktury A
- rozdělením dat tak, že data každé komponenty B1 jsou zpracovávána samostatným programem, takže existuje n identických programů, kde n je počet komponent B1.
- multithreading (jiné uspořádání předchozí varianty). Identické programy z předchozí varianty jsou realizovány fyzicky jedním reentrant programem, který pracuje vždy s daty odpovídající komponenty B1 (blíže viz oddíl stavová proměnná a kmenový soubor).

Rozpor mezí je charakterisován tím, že komponenta A1 může začínat kdekoliv v rámci komponenty B1 a může končit kdekoliv v rámci stejné nebo jiné komponenty B1. Komponenta B1 může začínat kdekoliv v rámci komponenty A1 a může končit kdekoliv v rámci stejné nebo jiné komponenty A1.



Rozpor mezí lze řešit těmito způsoby :

- rozdělením problému do dvou programů (jeden odpovídá struktuře A, druhý struktuře B), které si mezi sebou vyměňují data prostřednictvím vloženého souboru. Vložený soubor je vlastně iterací komponenty X, proces A rozlišuje komponentu A1 a neví nic o existenci komponenty B1, naopak proces B rozlišuje komponentu B1 a neví nic o existenci komponenty A1.
- inverze programu, což je v podstatě optimalizovaná varianta předchozího případu, kdy dva programy běží jako korutiny a prostřednictvím operační paměti si vyměňují jednotlivé záznamy. Blíže viz kapitola inverze programu.

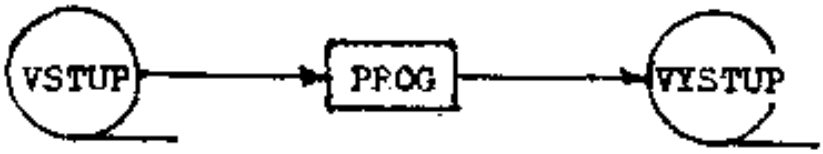
Rozpor mezí je patrně nejčastěji se vyskytující konflikt struktur. Již při vytváření poměrně jednoduchých sestav, které provádějí stránkování s tiskem hlaviček, se tento rozpor vy-

skytne. Nejobvyklejší způsob řešení, který je obvykle používán, vede k použití různých přepínačů, neřízených příkazů GO TO apod.

2. Inverze programu

Úvod

Rozpory struktur často řešíme rozdělením programu na dvě části, které si navzájem předávají data pomocí umělého vloženého souboru. Tak máme-li řešit nějaký problém pomocí programu PROG



můžeme si řešení usnadnit rozdělením programu na dvě části a zavedením souboru MEZI

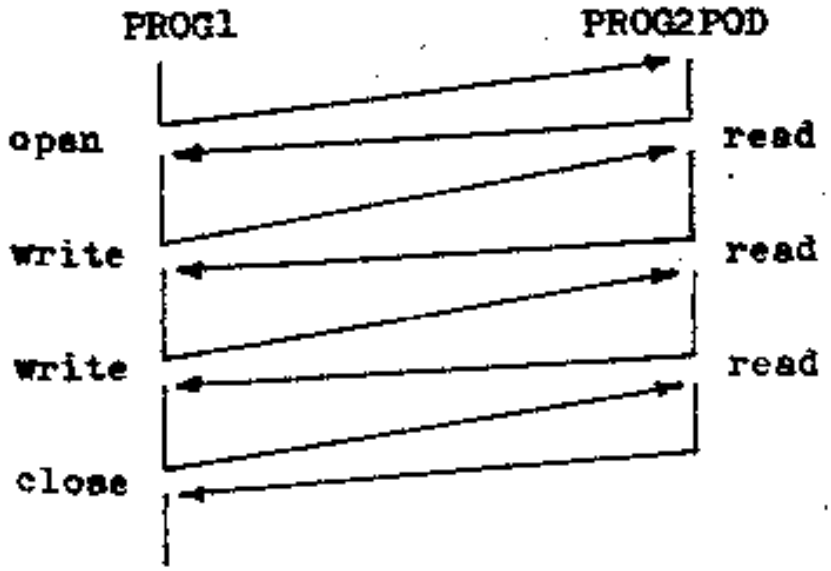
Rozpor datových struktur problému nemusí být jediný důvod rozdělení programu na části. Toto rozdělení můžeme provést i proto, aby program mohlo psát i ladit více programátorů nezávisle na sobě, nebo prostě z pohodlnosti si problém zjednodušíme.

Skutečně však realizovat vložený soubor by bylo nehospodárné. Vložený soubor jsme zavedli pouze z metodologických důvodů, pro zjednodušení problému. Je zbytečné aby PROG2 čekal se zahájením výpočtu až bude vytvořen celý soubor MEZI, když může začít ihned po vytvoření prvého záznamu. Zbytečně by se prodlužoval čas výpočtu o akce

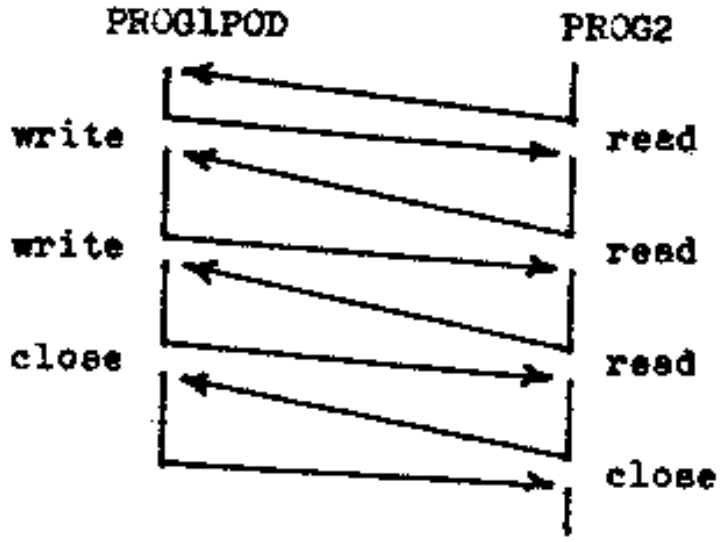
přídavného zařízení použitého pro soubor MEZI.

Soubor MEZI můžeme vyloučit tím, že jeden program zvolíme za hlavní a druhý invertujeme v podprogram. V našem případě zvolíme například PROG1 za hlavní a PROG2 invertujeme v podprogram PROG2POD. Příkazy "read" a "write" jsou vlastně vyvolání podprogramů - rutin IOC, které provádí příslušné akce se souborem. Úprava hlavního programu je jednoduchá, vyvolání rutiny IOC nahradíme vyvoláním vedlejšího programu. Úprava vedlejšího programu - inverze v podprogram - je složitější, nelze zde provést tak jednoduchou náhradu. Jestliže "write" ve PROG1 předává záznam ke zpracování, tak "read" ve PROG2POD tentýž záznam přináší a posloupnost příkazů za "read" jej zpracovává. Toto zpracování je ukončeno až dalším "read". Tedy vyvolat PROG2POD znamená provést jej od jednoho "read" k dalšímu. Pokud zná čtenář pojem korutiny /souprogramu/, pak inverze programu je totéž co kódování korutiny.

Předávání řízení lze vyjádřit tímto schematem:

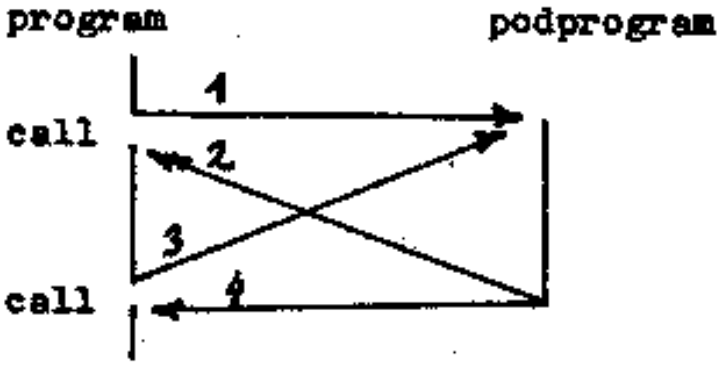


Svislé čáry zde označují průběh řízení uvnitř programu, čáry se šipkami předávání řízení mezi programy. Poslední "read" nedostává záznam, ale informaci o konci souboru. Je to ve shodě s tím, že při přečtení celého souboru je počet příkazů "read" o jeden větší než počet záznamů. Stejně dobře lze zvolit za hlavní program PROG2, pak schema vypadá takto:



Vidíme, že oba programy se v obou případech chovají skoro symetricky, jeden vyvolává druhý jako podprogram. Nesymetričnost je v tom, že hlavní program má o jedno vyvolání více. Je to známý případ, že počet příček žebříku se liší o jednu od počtu polí, které ohraničují.

V jedné věci se korutiny podstatně liší od podprogramů. Při vyvolání začíná podprogram svou činnost vždy od jednoho místa - svého vstupního bodu, kdežto korutiny pokračují dále ve výpočtu, tj. jejich vstupní bod závisí na jejich minulosti, na minulých vyvoláních. Schematicky se vyvolání podprogramu dá znázornit takto:



čísla určují časový sled předání řízení.

Vzpomeňme si na pravidlo, že prvé "read" následuje bezprostředně za otevřením vstupního souboru /lze tedy oba příkazy spojit - jak je to např. v PL/1/, a že "close" výstupního souboru je výkonný příkaz podobně jako "write" - zapisuje příznak konce souboru. "Open" výstupního a "close"

vstupního vloženého souboru slouží pouze k provedení úvodní či závěrečné části invertovaného programu.

Poznámka: U reálných souborů je otevření hlavně kontrolní akce. Kontroluje se, zda-li je přítomno medium s požadovaným souborem. Uzavření vstupního souboru může i u reálných souborů často chybět, aniž toto opomenutí nějak ovlivní výsledek výpočtu.

Bystrý čtenář si jistě všiml, že zatím je popisován případ dočtení souboru do konce. Obecnějším případem se budeme zabývat později.

Technika inverze

Je důležité si uvědomit: Inverze je věc kódování, tj. převodu pseudokódu do programovacího jazyka, nikoliv součást návrhu struktury programu. Ukážeme si techniku inverze PROG2 v PROG2POD v různých jazycích /viz následující stránka/.

V Cobolu můžeme použít obou technik, jak rozskoku /viz Fortran/, tak záměny návratů z podprogramů /viz Assembler/. Technika rozskoku je bezpečnější, vychází z vlastností jazyka. Technika záměny návratů je nebezpečná, poněvadž existují překladače, které po opuštění podprogramu nezajistí uschování návratu. V PL/1 lze použít pouze rozskoku, kde proměnná S má atributy LABEL a STATIC /do S dosazujeme přímo návěští/.

Proměnná S je soukromá proměnná podprogramu, žádný jiný program či podprogram nemá právo ji měnit. Je to typická lokální proměnná.

Pseudokód:

```
PROG2 seq
      ...
      open mezi
      read mezi
      ...
      read mezi
      ...
PROG2 end
```

Fortran:

```
   SUBROUTINE PROG2 (VETA)
   INTEGER S
   DATA S/1m1/
   GO TO (51,52,53...)S
51 ...
   ...
   S=2
   RETURN
52 ...
   ...
   S=3
   RETURN
53 ...
   ...
   RETURN
   END
```

Assembler /předpokládáme vyvolávání posloupností

```
      L 15,=V(PROG2POD)
      BALR 14,15 /
PROG2POD EQU   *
         USING *,15
         STM   3,2,EXTRUKL
         LM    3,2,INTRUKL
         DROP  15
         USING PROG2POD,3
         BR    14
PROG2SEQ EQU   *
         ...
         BAL   14,READ
         ...
         BAL   14,READ
         ...
PROG2END LM    3,2,EXTRUKL
         BR    14
*
READ     STM   3,2,INTRUKL
         LM    3,2,EXTRUKL
         BR    14
EXTRUKL  DS    16F
INTRUKL  DC    A(PROG2POD) reg 3
         DS    10F
         DC    A(PROG2SEQ) r.14
         DS    4F
```

Cobol:

```
LINKAGE SECTION.
01 DATA.
   02 ZAZNAM.
      03 ...
         ...
   02 END-OF-FILE PIC X.
      88 EOF VALUE "E".
...
77 S PIC 999 VALUE 1.
...
PROCEDURE DIVISION.
    ENTRY "PROG2POD" USING DATA
    GO TO Q1, Q2, Q3 ... DEPENDING ON S.
Q1.   ...
      ...
    MOVE 2 TO S GOBACK.
Q2.   ...
      ...
    MOVE 3 TO S GOBACK.
Q3.   ...
      ...
    GOBACK.
```

Zdůrazňuji, posloupnost MOVE i TO S GOBACK.

Qi. ...

je kódování příkazu "read" pseudokódu; chápat strukturu jako

```
PROG2 seq
      PPP select S=1
           ...
      PPP or S=2
           ...
      PPP end
PROG2 end
```

je chybné a vede k vyjímečně nepřehledným programům. Proměnná S v našich příkladech určuje stav programu, známe-li její hodnotu, víme co program dělá. Proto se jí též říká stavová proměnná.

Poznatky o technice inverze lze shrnout následujícím způsobem:

- Invertované "read" /"write"/ lze realizovat makrem.
- Invertované "read" /"write"/ je "read" /"write"/, pseudokód se nemění.
- Invertovaný kód a neinvertovaný jsou rovnocenné.
- Vyhýbáme se použití invertovaného "read" /"write"/ v podprogramu invertovaného programu. U invertovaných programů raději podprogramy nepoužíváme /perform-free programming/.

Pro ilustraci uvedu, že INPUT PROCEDURE a OUTPUT PROCEDURE slovese SORT v Cobolu jsou invertované programy.

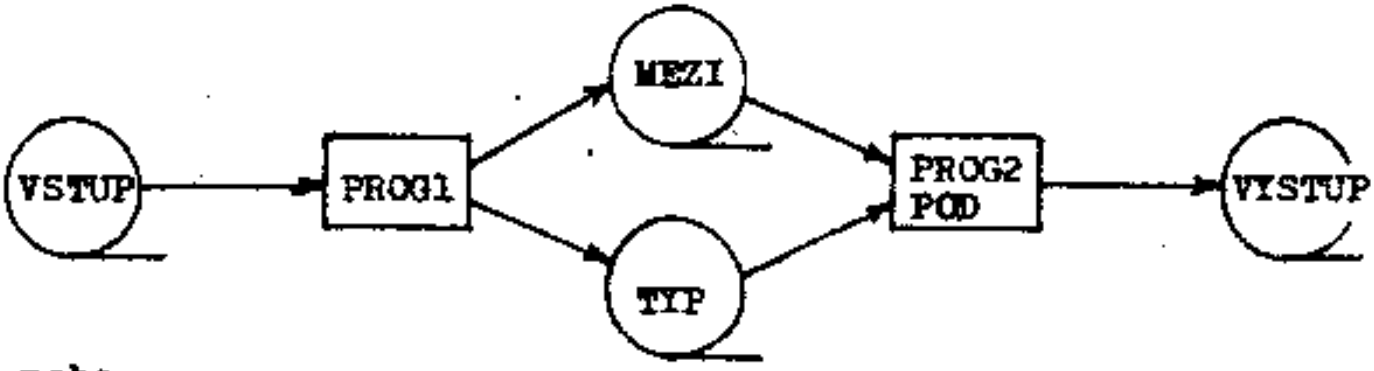
Komplexní inverze

V předchozích odstavcích jsme předpokládali, že počty příkazů "read" a "write" si odpovídají. Nyní budeme uvažovat o problému realizovat pomocí inverze všechny akce obvyklé se soubory. Vyloučíme z úvah soubory s přímým přístupem - omezíme se na sekvenční přístup. Jedná se tedy o operace "open", "close", "read", "write", "rewrite", "note", "point".

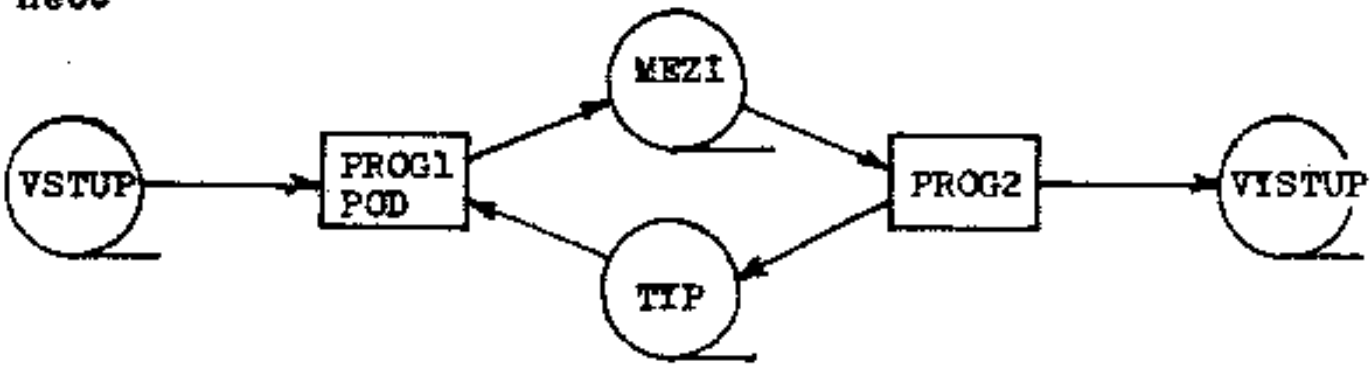
- "open" Při otevření souboru se kontroluje správnost medií a soubor se nastavuje na počátek. První akci - kontrolu - neznáme důvod realizovat - nebudeme přece kontrolovat, zda-li PROG2POD je skutečně PROG2POD. "Open" je vlastně nejčastější nesekvenční příkaz při sekvenčním přístupu. V našem případě nastavení invertovaného programu na počátek představuje naplnění stavové proměnné S počáteční hodnotou. To oceníme tehdy, jestliže invertovaný program je použit několikrát bez násobného nahrávání do paměti.
- "close" Uzavření výstupního souboru představuje zapsání záznamu s informací o konci souboru /EOF/. U vstupního souboru umožňuje ukončení zpracování aniž byl celý soubor dočten.
- "read", "write" Jejich použití je ukázáno v předchozích kapitolách.

- "rewrite" Změnit vyvolávaný program tak, aby při dalších běžích programu dával jiná data, není zrovna čistý způsob programování, pokud je vůbec uskutečnitelný.
- "note" Zaznamenání stavu sekvenčního souboru tak, aby se bylo možno na toto místo vrátit, je vlastně příkaz pseudokódu posit /pro zpětné sledování - backtracking/.
- "point" Návrat na zaznamenaný stav souboru, nesekvenční příkaz. Lze použít při realizaci příkazu pseudokódu quit /zpětné sledování/. Příkazy "note" a "point" se realizují pomocí vícenásobného čtení napřed, viz zpětné sledování.

Při komplexní inverzi hlavní program kromě toho, že vyvolává vedlejší, ještě mu oznamuje, o jaký typ vyvolání jde. Předání typu lze chápat podobně jako předání záznamů:



nebo



Údaj typu můžeme předávat podobně jako v IOC rutinách - vyvoláním podprogramu na různých vstupních bodech /možno v PL/1/, nebo předáním údaje o typu jako proměnné /vstupní parametr/, a podprogram se sám rozhodne co bude činit. Ukážeme si to na příkladě PROG1POD, kde úvodní akcí je nastavení stavové proměnné S a otevření vstupního souboru; závěrečnou akcí je uzavření vstupního souboru /vytočení magnetické pásky/ bez ohledu na to, byl-li soubor dočten.

```
    ENTRY "PROG1POD" USING ZAZNAM, TYP
    IF TYP = "OP" MOVE 1 TO S OPEN INPUT VSTUP GOBACK.
    IF TYP = "CL" CLOSE VSTUP   LOCK GOBACK.
    GO TO Q1, Q2, ... DEPENDING ON S.
Q1.   ...
```

Přidáním dat pro určení typu si nezkomplikujeme situaci při návrhu, protože tato data mají strukturu adekvátní vloženému souboru.

Stavová proměnná a kmenový soubor /multi-threading/

Tento vztah vysvětlím pomocí příkladu. Předpokládejme, že máme podnik s elektronickými píchačkami, které místo záznamu na kartu pošlou údaje /z magnetické průkazky/ do real-timového počítače. Takový údaj se bude skládat z osobního čísla, času a druhu /příchod, odchod/. Pokud program v počítači sleduje pouze jednoho zaměstnance, je situace jednoduchá. Program čeká na údaj, když jej píchačky pošlou, tak čas příchodu uloží, při zaslání odchodu vytvoří rozdíl a tento přičte k odpracované době.

Při více zaměstnancích můžeme takový program napsat pro každého z nich. V počítači je distribuční program, který podle osobního čísla distribuuje údaje programům pro jednotlivé zaměstnance. Tyto programy musíme ovšem kódovat v invertovaném tvaru. Každý z nich má ve svých lokálních proměnných stavovou proměnnou, dosud odpracovaný čas, druh minulého údaje /pro kontrolu/, čas příchodu. Všechny tyto invertované programy mají stejný text, liší se pouze obsahem svých lokálních proměnných, které určují jejich stav.

Můžeme tedy udělat další myšlenkový krok. V počítači máme distribuční program a program pro výpočet odpracované doby /invertovaný/. Mimo to máme soubor stavových vektorů, každý vektor představuje stav invertovaného programu pro určitého zaměstnance. Distribuční program předává invertovanému programu časový údaj z píchaček, stavový vektor /podle

osobního čísla/ a typ vyvolání /open, write, close/. Invertovaný program si stavovým vektorem naplní své lokální proměnné, provede příslušné zpracování a navrátí nový stavový vektor distribučnímu programu, který jej uloží zpátky do souboru.

Jestliže podobným způsobem jako příchod a odchod zpracovávéme informace o narození dítěte, ženitbě a rozvodu, přestěhování ap., tak nám soubor stavových vektorů vytvoří kmenový soubor mzdové účtárny. V případě, že údaje nezpracovávéme ihned, ale pouze zapisujeme na magnetickou pásku, kterou /setříděnou dle osobních čísel/ zpracovávéme jednou měsíčně, dostaneme normální mzdovou agendu.

Tento příklad nám ujasnil podstatu kmenového souboru a ozřejmil, že návrh programu pro zpracování v reálném čase se nemusí lišit od agendového, že struktura programu na typu zpracování nezávisí.

Shrnutí

Tato kapitola ukazuje techniku řešení rozporu ve strukturách dat při návrhu programu. Postup při inverzi lze shrnout takto:

- zjištění rozporu mezi datovými strukturami
- návrh struktury vloženého souboru
- návrh struktur obou programů, na které se řešení programu rozpadá
- normální postup návrhu obou programů až do úrovně pseudokódu
- rozhodnutí, který program je hlavní a který vedlejší
- úprava hlavního programu, tj. nahražení příkazu se souborem vyvoláním vedlejšího programu jako podprogram
- inverze vedlejšího programu v podprogram

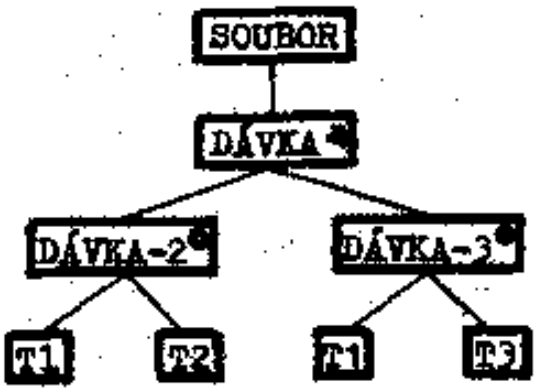
V článku se jedná vždy o jeden vložený soubor a dva programy, čtenář však si jistě již všiml, že stejně lze řešit i rozpory, které si vynutí vložení více než jednoho souboru.

3. Zpětné sledování

Násobné čtení vpřed

Při vytváření struktury porgamu používáme složku selekce a složku iterace ve tvaru, kde test podmínky předchází provedení části. Používáme také metodu čtení vpřed, abychom mohli zpracovat sekvenční vstupní soubory. Výsledkem jsou jednoduché a snadno srozumitelné programy. Jednoduchost programu spočívá převážně ve skutečnosti, že není možné provést žádnou složku programu, dokud neexistují příslušná data. Metoda čtení vpřed zajišťuje dostupnost nezbytných dat v hlavní paměti pro provedení testů podmínky.

Někdy nepostačí číst vpřed pouze jeden záznam. Mějme soubor se strukturou:



Budeme-li předběžně číst pouze jeden záznam z tohoto souboru, pak vždy před začátkem zpracování každé dávky jsme schopni pouze zjistit, že další záznam je T1, a nebude existovat způsob jak určit, zda tato dávka je dávkou typu 2 nebo dávkou typu 3 (kontext zpracování T1 v DÁVKA-2 je jiný, než kontext zpracování T1 v DÁVKA-3). Je zřejmé, že tento problém můžeme překonat předběžným čtením dvou záznamů. Dospějeme k programu (v závorkách je specifikovaný zpracovávaný záznam):

```
PROGR   seq
            open; čti další; čti další+1;
PBODY       itr until eof
PDAVKA          sel další je T1 a další+1 je T2
                    zpracuj T1 ad 2 (další);
                    zpracuj T2 (další+1);
                    čti další; čti další+1;
PDAVKA          or  další je T1 a další+1 je T3
```

```
                zpracuj T1 ad 3 (další);
                zpracuj T3 (další+1);
                čti další; čti další+1;
PDAVKA      end
PBODY     end
          close; stop;
PROGR end
```

Uvedení příkazů čtení je poněkud nešikovné. Je dobré vždy po zpracování každé vstupní věty přečíst větu další. Není na závadu mít dva různé příkazy čtení? Neměli jsme příkazy čtení umístit za konec složky DAVKA? Jak bychom pak řešili případ, kdy DÁVKA-3 by byla sekvencí vět T1, T3, T1? Po DÁVKA-2 bychom pak museli provést dvojí čtení, ale po DÁVKA-3 dokonce trojí. Jak zajistit bezchybné zpracování prázdného souboru?

Abychom mohli odpovědět na tyto otázky, pokusme se uspořádat metodu násobného čtení vpřed. Uvidíme, že odpovědi na naše otázky budou překvapivě snadné.

Předpokládejme, že chceme přečíst metodou násobného čtení vpřed právě n záznamů. To znamená, že v kterémkoliv bodě programu chceme mít možnost prověřit záznamy další, další+1 až další+n-1. Deklarujme n oblastí záznamu. Požadujeme, aby v kterémkoliv bodě programu každá z těchto oblastí obsahovala vždy stejnou větu z n dalších, nejlépe: 1. oblast bude obsahovat záznam další, 2. oblast další+1, atd. až n-tá oblast bude obsahovat záznam další+n-1.

Poněvadž označení konce souboru považujeme za prodloužení záznamu, za jeho nedílnou logickou součást, budeme pro každou oblast záznamu potřebovat označení konce souboru. Například v Cobolu lze psát:

```
01 BUFFER.
  02 BUFFER1.
    03 FBUFØ PIC X(délka-záznamu).
    03 FBUF1 PIC X(délka-záznamu) OCCURS (n-1).
```

```
    03 FEOFØ PIC X.
    03 FEOF1 PIC X OCCURS (n-1).
 02 BUFFER2 REDEFINES BUFFER1.
    03 FBUF2 PIC X(délka-záznamu) OCCURS (n-1).
    03 FBUFN PIC X(délka-záznamu).
    03 FEOF2 PIC X OCCURS (n-1).
    03 FEOFN PIC X.
```

Potom operace čtení bude provedení odstavce:

```
FREAD.  MOVE FBUF1 TO FBUF2.
        MOVE FEOF1 TO FEOF2.
        IF FEOFN=SPACE
           READ F INTO FBUFN AT END MOVE "E" TO FEOFN.
```

a operace otevření vstupního souboru bude provedení odstavce:

```
FOPEN.  OPEN INPUT F.
        MOVE SPACE TO FEOFN.
        PERFORM FREAD  n  TIMES.
```

Inicializovali jsme FEOFN, abychom si byli jisti, že podmínka "konec souboru" je rozhodnutelná před pokusem o první příkaz READ a provedli jsme n jednoduchých operací čtení.

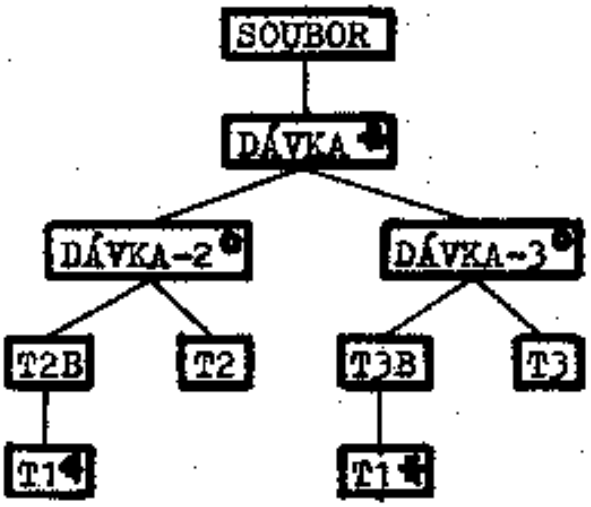
Při řešení našeho problému bude n=2 a lze navrh-nout program:

```
01 BUFFER.
...
PROCEDURE DIVISION.
PROGR.  PERFORM FOPEN.
        PERFORM PBODY UNTIL FEOFØ="E".
        CLOSE F.
        STOP RUN.
PBODY.  IF FBUF1-POZICE1=2
           CALL "PT12" USING FBUFØ.
           PERFORM FREAD.
           CALL "PT2" USING FBUFØ.
           PERFORM FREAD.
```

```
            ELSE
                CALL "PT13" USING FBUFØ.
                PERFORM FREAD.
                CALL "PT3" USING FBUFØ.
                PERFORM FREAD.
        FREAD.  ...
        FOPEN.  ...
```

Všimněme si závažné skutečnosti. Zásada umisťování operací čtení se ve srovnání s metodou jednoduchého čtení vpřed vůbec nezměnila. (Samozřejmě je možné navrhnout výhodnější aparát násobného čtení vpřed, a to i v Cobolu, kdy není nutné v hlavní paměti přemisťovat celé oblasti záznamů.)

Násobné čtení vpřed vyřeší některé úkoly, nikoli však všechny. Mějme například soubor:



Takový soubor nedokážeme zpracovat násobným čtením vpřed nějakého pevně stanoveného počtu záznamů. V systémech zpracování v dávkách je to problém velice běžný. Ovšemže není nemožné vyřešit takové úlohy nalezením individuálního řešení ad hoc pro každou dávku. My však budeme hledat řešení obecnější.

Problém zpětného sledování

Máme za úkol zapsat programovou složku pro zpracování jediného štítku. Tento štítek obsahuje tři pole - F1, F2 a F3. F1 by mělo být numerické v rozsahu od 1 do 99 a v kladném případě ho může být použito k vyhledání informace z tabulky. Informace obsahuje dvě mezní hodnoty a násobitele. Je-li F2 v daném rozsahu, může být násobeno násobitelem a součin je adre-

sou na disku. Záznam v adrese na disku obsahuje řetězec znaků. F3 by mělo být subřetězcem tohoto řetězce znaků. V kladném případě má být vytištěna jeho předpona.

Vznikne tudíž následující situace. Všechna pole - F1, F2 a F3 - mohou obsahovat bezchybná nebo chybná data. Existují zde tři procesy P1, P2 a P3, které jsou prováděny při F1, F2 a F3 v daném pořadí:

P1 vyhledává informaci v tabulce,
P2 zpřístupňuje záznam na disku,
P3 tiskne předponu F3 v řetězci znaků.

Programová složka by měla provádět následující operaci: Jsou-li všechna pole bezchybná, vytiskni předponu F3, jinak proveď chybovou rutinu FX.

Ale v naší úloze platí omezení: Je-li Fn chybné, nelze provést Fn (n=1,2,3) a nebylo-li provedeno Fn-1, není možné vyhodnotit bezchybnost Fn (n=2,3).

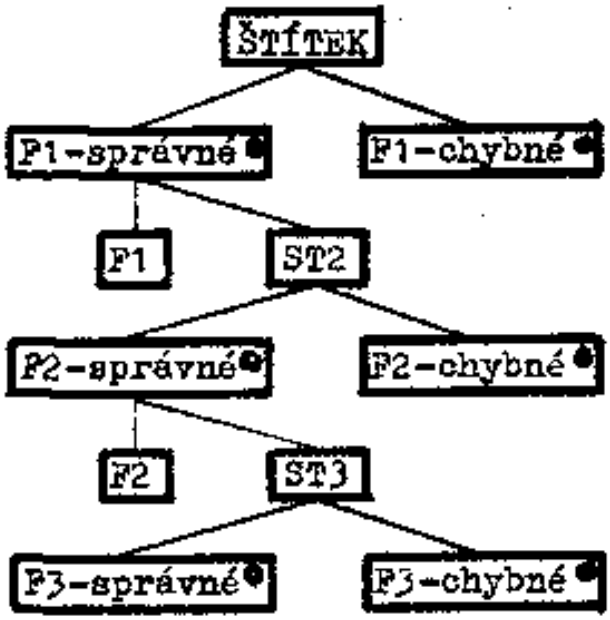
Úlohu lze vyřešit následujícím programem:

```
STITEK  sel F1 bezchybné
            proveď P1;
ST2         sel F2 bezchybné
               proveď P2;
ST3            sel F3 bezchybné
                  proveď P3;
ST3            or
                  proveď FX;
ST3            end
ST2         or
               proveď FX;
ST2         end
STITEK  or
            proveď FX;
STITEK  end
```

Omezení úlohy nám nedovolí správné elegantnější řešení. Násobné čtení vpřed nám zcela jistě nepomůže. Ale stávající řešení

není uspokojivé, poněvadž pole F jsou na štítku tři pouze náhodou. Představme si na chvíli program pro 30 polí F. Například složka PX by byla provedena z 30 různých míst programu. To nelze tolerovat, poněvadž specifikace problému není tak podrobně členěna a proto ani náš program by neměl být takto podrobně členěn.

Při návrhu programu jsme vycházeli z datové struktury:

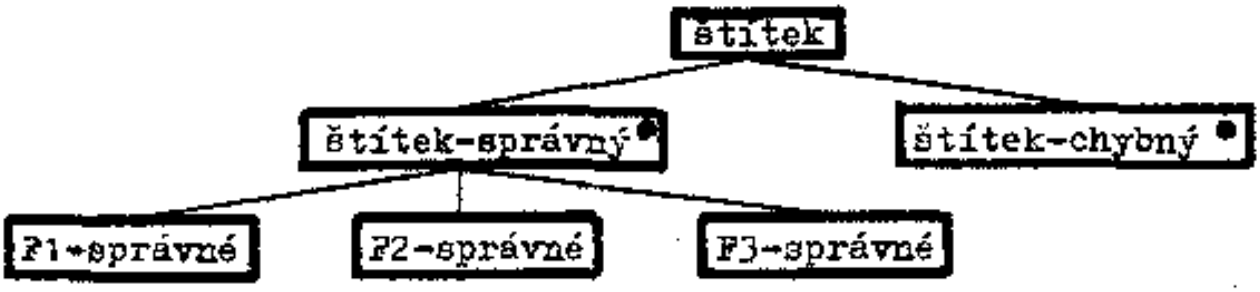


Poněvadž zjistit, zda je P2 bezchybné, má význam pouze v kontextu správného P1 a podobně to platí pro P3 a P2, rozdělujeme možné chyby do tří skupin:

- chyba v P1,
- správné P1, ale chyba v P2,
- správné P1 i P2, ale chyba v P3.

Rádi bychom takové rozdělení neprováděli, spokojili se s jedinou operací "proveď PX;". Takové rozdělení chyb konec konců ve specifikaci úlohy nebylo, je zapříčiněno pouze metodou indikace chyb.

Nejpřirozenější datová struktura pro tuto úlohu zní:



Štítek je buď správný nebo chybný. Je-li chybný, nemáme o jeho obsah zájem, provedeme pouze PX. Je-li štítek správný, je na něm bezchybné P1 (provedeme P1), bezchybné P2 (provedeme P2) a bezchybné P3 (provedeme P3). Struktura dat je perfektní, operace jsou přiděleny bezchybně, program je nádherně jednoduchý

a samozřejmě správný. Ale nechodí. Neexistuje způsob, jakým bychom při zápisu selekce "štítek" určili, zda jde o bezchybný nebo chybný štítek. Odpověď na tuto otázku může dát pouze zpracování, ale my žádáme odpověď před tímto zpracováním.

Znamená to snad, že náš program nelze sestavit ze složek definovaných výše, že jsme nuceni zavést novou programovou a datovou složku? Ne, je potřebné zavést pouze nový formát selekce, v němž test podmínky není nutně v záhlaví selekce, tj. v místě vstupu.

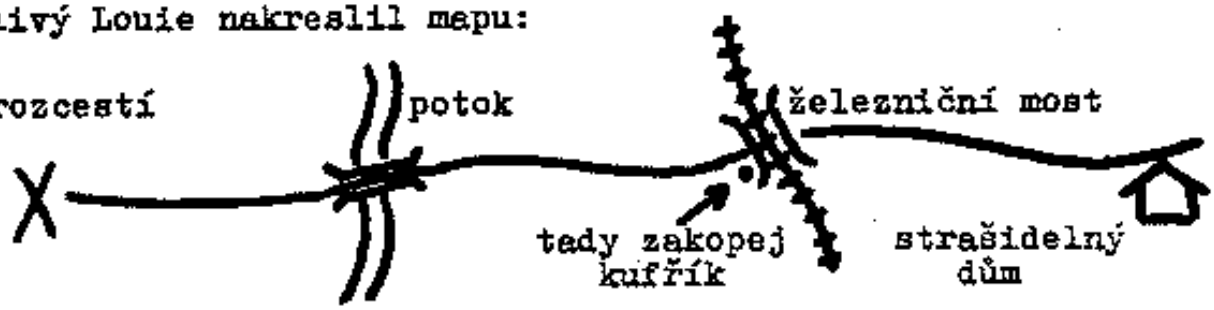
Mravní ponaučení

Ukvapeně, v naději na finanční zisk, který sahá daleko za hranice snů programátora, jste se přidali k tlupě Velkého Louieho. A tak jednoho horkého letního odpoledne jdete po venkovské cestě na setkání s Velkým Louiem. Nesete malý, ale těžký kufřík s výnosem poslední akce tlupy. Zastavíte se na rozcestí a prověříte instrukce: Louie si velice potrpí na dokumentaci.

"100 metrů za rozcestím cesta přetíná potok a po dalších 150 metrech vede pod železniční tratí. Zakopej kufřík těsně u mostu a křídou na něm nakresli kříž, aby hoši věděli, že se ti to povedlo. Po 200 metrech je strašidelný dům. Tam se spolu sejdeme. Pro štěstí si s sebou přines čtyřlístek, je to velice důležité.

Louie."

Pečlivý Louie nakreslil mapu:



Pokyny jsou výborné - až na to, že na rozcestí neviíte, zda jít doleva nebo doprava. Louie vám to zapomněl říci. Je velice horký den a tak se na chvíli posadíte, abyste to "vymysleli". Po chvilce si uvědomíte, že před vámi stojí malý mužíček s bílou bradkou. "Jsem dobrý skřítek", představuje se. "Pravidla znáš: Můžeš mi položit jednu otázku a já ti mohu odpovědět slovem "ano" nebo "ne". Jdete přímo k věci. "Mám jít vlevo?", zeptáte se. "Ano", říká on a okamžitě mizí.

Takže se vydáte vlevo. Jistě, že po 100 metrech přejdete potok a po dalších 150 metrech dospějete k železničnímu mostu. Zakopete kufřík s klenoty, uděláte křídou kříž na mostě a pokračujete dál. Později uvidíte u cesty čtyřlístek, který si zvednete, ale není ani stopy po strašidelném domě přesto, že jste

od mostu ušli už celých 200 metrů.Co teď? Posadíte se a opět "vymýšlíte". Uvažujete asi takto :"Jsem rád, že jsem skřítka potkal. Mohl jsem na rozcestí stát hezky dlouho a přemýšlet, co dál. Je jisté, že díky němu jsem aspoň někam šel, a to je určitě lepší než nic.Ale nejsem si jist, zda na něj je spolehnutí. Předpokládal jsem, že levá cesta je správná, protože to říkal on. AleLouieho mapa ukazuje, že v tomto místě po správné cestě je strašidelný dům. Ergo tot není správná cesta. Teď, když o tom tak přemýšlím, kdybych neviděl potok a železnici, býval bych také poznal, že nejsem na správné cestě. Louieho mapy nikdy nelžou. Měl bych se teď raději vrátit na rozcestí a dát se po správné cestě. Napadá mně, že bych to mohl vzít zkratkou podél železniční tratě. Ale co když existují dvě různé železniční tratě ? Proto se určitě vrátím zpět na rozcestí. A ještě jedna věc. Nesmím zapomenout vykopat kufřík s klenoty. To je opravdu důležité. Nevím, co si mám myslet o tom, kříži namalovaném křídou, ale "chleba to nejí" a já mám ještě křídy dost. Čtyřlístek si určitě nechám. Byl bych blázen,kdybych ho zahazoval. Ještě bych musel hledat nový a takové spousty jich tu zase nejsou." Takže se vracíte a na zpáteční cestě vykopete kufřík s klenoty. Ukazuje se, že pravá cesta je správná. Musela být, protože existují pouze dvě cesty a levá správná nebyla. A šťastně se ve strašidelném domě setkáte s Velkým Louiem. Když mu povíte o svém putování, utrousí cosi o tom, že programátoři jsou nejlepší lupiči a jako premii vám dá jeden či dva rubíny navíc.

Metoda zpětného sledování

Vraťme se nyní poučení k naší úloze zpracování štítku. Úlohu budeme řešit ve třech stadiích.

V prvním stadiu spoléháme na dobrého skřítka, který nám řekne, kterým směrem pokračovat (jsou dvě možnosti volby podle odpovědi ano/ne). Úloha je vlastně selekce a program zní :

```
STITEK  sel bezchybný štítek
            do P1;
            do P2;
            do P3;
STITEK  or  chybný štítek
            do PX;
STITEK  end
```

Toto řešení je dokonalé, poněvadž dobrý skřítek vždy odpoví na otázku "Je to bezchybný štítek ?" ano či ne. (Pro ty, kteří si myslí, že v programování není místo pro dobré skřítky, nechť laskavě předpokládají, že předcházející program milostivě vyděroval znak X do sloupce 80 právě v každém chybném štítku. Návrh tohoto programu nebudeme zkoumat.)

Ve druhém stadiu zjistíme, že na skřítka není spolehnutí - skřítek totiž odpoví vždy "ano", musíme proto ke zpracování bezchybného štítku zaujmout skeptičtější postoj a při každé příležitosti si klást otázku, jestli se přece jenom nejedná o chybný štítek. Schema struktury dat bezchybného štítku nám poskytuje potřebnou mapu. Každý význačný orientační bod při jeho míjení odškrtneme. Takže nyní píšeme :

```
STITEK  posit bezchybný štítek.
              quit STITEK if F1 chybné ;
              do F1;
              quit STITEK if F2 chybné ;
              do F2;
              quit STITEK if F3 chybné ;
              do F3;
STITEK  admit chybný štítek
              do FX;
STITEK  end
```

Provedli jsme dvě změny. Užili jsme místo známých slov select a or slov posit (vytvoření hypotézy) a admit (připuštění jiné možnosti). Dále jsme zavedli příkazy "quit STITEK if ... " . To jsou proveditelné příkazy, které jsou umístěny v každém bodě struktury dat bezchybného štítku, kde hypotéza o bezchybnosti štítku může být prověřena skutečností. Takto "quit STITEK if F1 chybné ; " znamená : Není-li toto správné F1, pak podle mé struktury dat nemůže ani tento štítek být bezchybný a musím připustit nesprávnost hypotézy. Je nutné si uvědomit, že splnění podmínky operace quit znamená návrat na "rozcestí" ovšem s bezpečnou znalostí správné odpovědi na původní otázku.

Třetí stadium je konečným stadiem, kdy zvažujeme vedlejší účinky. Vedlejší účinky zpětného sledování rozdělujeme na negativní, neutrální a positivní.

- negativní účinky musí být odstraněny. Jestliže jsme zakopali kufřík se šperky, musíme jej znovu vykopat. Jestliže jsme na řádkové tiskárně vypsali řádku, musíme se vrátit zpět a vymazat ji. Jestliže jsme přečetli štítek, musíme se vrátit zpět ve snímači děrných štítků. Jestliže jsme inkrementovali proměnnou, musíme

ji nastavit na původní hodnotu.

- neutrální účinky mohou ale nemusejí být odstraněny. Jestliže jsme na mostě udělali křídou kříž, můžeme jej vymazat nebo jej tam můžeme ponechat. Jestliže jsme přečetli záznam na disku do hlavní paměti, můžeme jej tam ponechat nebo přepsat obsah vyrovnávací paměti - podle toho, co je výhodnější. Jestliže jsme stanovili pro proměnnou nějakou hodnotu, můžeme ji uchovat nebo ji přepsat.
- prospěšný vedlejší účinek je ten, který je pro nás výhodný. Byli bychom bláhoví, kdybychom čtyřlístek zahodili, a tak si ho ponecháme.

V naší úloze můžeme všechny vedlejší účinky považovat za neutrální. Je jedno pro část admit, zda jsme v tabulce vyhledali informaci či nikoliv nebo zda jsme záznam na disku zpřístupnili či nikoliv. Třetí stadium je snadné.

Negativní důsledky zpětného sledování

Častěji však vedlejší účinky působí vážné problémy. Je poučné prověřit, jak vznikají tyto vedlejší účinky v úlohách zpětného sledování, a obecně zvážit, jakých metod použít, abychom se s nimi vypořádali. Vězme na příklad program vytvořený ve 2.stadiu zpětného sledování :

```
A       pos (NOT(C1 OR C2))
           do X;
           quit A if C1;
           do Y;
           quit A if C2;
           do Z;
A       adm (C1 OR C2)
           do W;
A       end
```

Problém zvládnutí negativních důsledků zpětného sledování je problémem obnovení stavu výpočtu S na počátku složky A pro část admit. Stav výpočtu S byl určitým způsobem porušen při provádění části posit, mohlo být provedeno X nebo X i Y.

Metoda DO/UNDO zvládá vedlejší účinky samostatně pro každý příkaz quit. Po určení ,že má být provedeno quit, ale dříve než se tak opravdu stane, odstraníme danou skupinu momentálně se vyskytujících vedlejších účinků. Metoda STORE/RESTORE nám umožňuje vyhnout se podrobnému přezkoumání kontextu jednotlivých příkazů quit. Zaznamenáme stav výpočtu S na počátku části posit a tento stav bezpodmínečně obnovíme na počátku části admit. Tyto metody jsou ovšem vhodné pouze pro úzkou třídu možných negativních důsledků.

Jinou metodou je úplně se vyhnout vytváření vedlejších negativních důsledků v části posit. Místo operací s globálními proměnnými se můžeme omezit na nové proměnné, které jsou lokální v části posit. Příslušné lokální proměnné musíme inicializovat na vstupu do posit a na samém konci (vždy za posledním quit) přiřadíme globálním proměnným hodnoty patřičných lokálních proměnných.

Sekvenční vstupní a výstupní zařízení mohou být manipulována obdobným způsobem. Omezíme operace v části před alespoň jedním příkazem quit složky posit tak, aby samotný stav zařízení zůstal beze změny. Pro výstupní zařízení můžeme v hlavní paměti vytvořit frontu záznamů pro výstup. Pro vstupní zařízení můžeme použít metody násobného čtení vpřed, přičemž nebudeme provádět žádné operace čtení, ale podle potřeby budeme pracovat se záznamem další+1, další+2 atd.

Tuto metodu nazvěme metodou PRETENDING (předstírání) a 3. stadium řešení úlohy bude :

```
A       pos (NOT(C1 OR C2))
           pretend X;
           quit A if C1;
           pretend Y;
           quit A if C2;
           pretend Z;
           really do X;
           really do Y;
           really do Z;
A       adm
           do W;
A       end
```

4. Závěr

Z uvedeného výkladu základních principů technologie strukturovaného programování vyplývá, že ji lze aplikovat na řešení rozsáhlé třídy úloh. Jsou to všechny úlohy, které lze znázornit hierarchickou strukturou složenou ze sekvencí, selekcí a iterací. Technologie vznikla na základě požadavků praxe při hromadném zpracování dat. Rozsah její použitelnosti je však daleko větší. Metoda byla úspěšně prověřena, avšak existují určité kategorie úloh, pro které nasazení této technologie je problematické nebo alespoň nebylo v našich podmínkách vyzkoušeno. Tam náleží problémy vedoucí na rekursivní struktury, časové synchronisace více paralelních procesů, tvorba některých universálních programů, kde struktura zpracovávaných dat je známa teprve po přečtení parametrů. Technologii lze však využít pro programování interaktivních procesů. Je ji však nutné modifikovat v tom smyslu, že nelze číst napřed sekvenční soubory poselství přicházejících od terminálů. Technologii strukturovaného programování nelze chápat jako uzavřený soubor pevně daných neměnných pravidel, nýbrž je otevřená pro zapracování dalších standardů, které bude vyžadovat praxe.

Tato metodika vychází z předpokladu, že existují sekvenční data, která program zpracovává. Pokud se v programu pracuje též s daty s přímým přístupem, není struktura těchto dat převzata do struktury programu. U většiny řešených problémů z praxe lze tato sekvenční data nalézt a strukturalisovat. Výsledkem strukturalisace sekvenčních dat jsou datové struktury, z nichž je odvozována struktura programu. Návrh datových struktur není sice čistě deduktivním postupem, vyžaduje často jistou intuici a zkušenost, avšak dobře vytvořené struktury odráží objektivní realitu problému. Praxe ukazuje, že řešení různých programátorů jsou podobná, často dokonce shodná. Odvození programové struktury z datových struktur, je záležitostí aplikace uvedených pravidel a technik.

Jedna z obecných zásad strukturovaného programování je vyhýbání se příkazu GO TO. V popsané technologii je však příkaz GO TO používán při kódování v souvislosti s ruční kompilací iterací a selekcí, na začátku invertované korutiny a místo příkazu

quit v pseudokódu. Ve všech těchto případech se však jedná o řízené použití tohoto příkazu. Místo předání řízení je vždy jednoznačné a nezpůsobuje nepřehlednost kódu. Jednou ze zásad vedoucích k přehlednému a verifikovatelnému programu je to, aby program neměl vnitřní stavy, tj. nepoužíval přepínačů. Eof-indikátor takovýmto přepínačem není; jedná se pouze o jeden ze záznamů vyžadující speciální ošetření. Stavová proměnná používaná při inverzi programu je přepínačem. Přestože tato proměnná může nabývat více hodnot, nejedná se o mnoho přepínačů, které uvažovány dohromady udávají stav programu, nýbrž jedná se o samotný stav korutiny.

Použití technologie strukturovaného programování přináší řadu výhod, z nichž některé byly již výše diskutovány. Předně již analytik, který vypracovává zadání programu, může použít strukturálních diagramů k popisu dat. Může sice ještě vzniknout potřeba, aby programátor tuto strukturu zpřesnil či vypustil pro problém irelevantní komponenty. V každém případě je však položen jasný základ k formulaci problému, který je rozhodně přesnější než běžné slovní popisy. Ty velmi často nedefinují vztahy mezi jednotlivými druhy vět v souborech a přípustná chybová data.

Tato technologie představuje skutečnou metodu s jasným pracovním postupem pro vypracování strukturálního diagramu programu, která vede ke zvýšení efektivnosti již při návrhu programu. Přiřazování operací ze seznamu k jednotlivým komponentám zajišťuje, že každá operace bude zařazena na odpovídající hierarchickou úroveň ve struktuře, což vede k přehlednosti kódu. Vyskytnou-li se obtíže při přiřazování operací, plyne odtud, že navržená struktura je chybná, takže technologie zahrnuje i jistou sebekontrolu. Jen velmi málo je nekontrolovaně ponecháno intuici programátora. V tom spočívá rozdíl oproti modulárnímu programování nebo metodám strukturovaného programování, které vycházejí z funkčního přístupu, kde navrhovatel řešení provádí dekomposici problému podle své subjektivní úvahy.

Verifikovatelnost je zajišťována metodou přístupu k návrhu. Struktura programu je stromový graf. Dynamický proces běhu programu odpovídá statické struktuře programu (invertovaný program

je pouze formální transformací programu s řádnou strukturou). Stromová struktura a neužívání přepínačů vede k zjednodušení testování. Dosahuje se minimální chybovosti, usnadňuje se lokalisace chyb během testů, snižuje se potřeba lidského i strojového času pro testování. Údržba strukturovaných programů je méně náročná oproti programům chaotickým; to platí obzvláště tehdy, když tuto údržbu provádí někdo jiný než autor programu.

Podstatným přínosem je vznik části dokumentace v procesu návrhu programu. Dokumentace má jednotnou formu a kromě dokumentace pro provoz je hotová v okamžiku ukončení testování.

Tuto technologii nelze chápat odděleně od jiných technik zvyšujících efektivnost programování. Např. lze současně zavést metodu týmu vedoucího programátora. Tvorbu strukturálního diagramu a vlastní kódování s testováním lze svěřit rozdílným pracovníkům. U rozsáhlého programu lze zadat i vypracování podrobné struktury některé komponenty dalším pracovníkům, právě tak mohou některé elementární operace být odděleně realizovány. Domníváme se, že zavedení této technologie povede k specialisaci v rámci profese programátor na navrhovatele struktur a na kódovače těchto struktur. Přitom kódovače nelze považovat za méněcenné členy týmu, naopak tito pracovníci jsou za předpokladu pracovitosti a pečlivosti velkým přínosem pro zdárné ukončení projektu. Metoda potřebuje stálé procvičování; skupinové diskuse k návrhu řešení problému jsou velmi užitečné.

Zavádění technologie strukturovaného programování přináší však i některé problémy. Zkušenost nám ukázala, že i tam, kde jsou již zavedeny některé standardy, je vhodné je u nově řešených úloh opustit. Pro nasazení technologie je nutné školení spojené s praktikem, v němž by se pod dohledem instruktorů řešily složitější problémy. Zavedení technologie jako standardu v podniku vyžaduje sice podporu jeho vedení, nedá se však direktivně nařídit a očekávat od pouhého školení a příkazu zvýšení produktivity programátorských prací je ilusí. Přes řadu pravidel se jedná o tvůrčí práci a zkušení programátoři budou ochotně používat tuto technologii jen tehdy, když se sami přesvědčí o jejích výhodách.