Návrh algoritmů – rady a příklady

Doporučeným postupem při návrhu strukturovaných algoritmů je vycházet ze struktury dat.

Je výhodné graficky (nejlépe stromovým strukturogramem znázornit strukturu vstupních dat a strukturu požadovaného výstupu a znázornit vzájemné korespondence „co čemu na vstupu odpovídá na výstupu“).

Přitom platí tyto zásady (jde o rámcové „***rady***“, ne o přesný postup):

- Nehomogenní struktuře (záznamu) odpovídá sekvence v algoritmu zpracování (date představující různé informace, které následují po sobě, a je třeba je zpracovat všechny je třeba zpracovat postupně).
- Položkám různého typu odpovídá selekce při zpracování (může-li položka v datech mít různé významy, je třeba rozhodnout jaký ten význam je a podle toho volit různé alternativy zpracování).
- Homogenní strukturu (pole nebo proud) zpracováváme iterací (stejnou činnost opakujeme pro všechny položky téhož typu). Přitom je třeba odlišit statické a dynamické struktury:

❖ U statické struktury – **pole** známe předem jeho rozsah. Konec iterace musíme tedy vázat na počet prvků pole. Položky pole jsou přímo přístupné pomocí indexu. Tedy na počátku naplnit indexovou proměnnou počáteční hodnotou *J* (obvykle 1 nebo 0), v těle cyklu ji zvětšovat a podmínku iterace testovat $J \leq N$ nebo $J < N$, kde *N* je počet prvků v poli. Musíme dát pozor na to, zda hodnotu indexové proměnné zvětšujeme po zpracování položku nebo před zpracováním a zda test je na neostrou nebo ostrou nerovnost. Některé programovací jazyky pro takový cyklus nabízejí jednoduchou konstrukci typu `for J := 1 to N do … ,` nebo její modifikace. Ta provede „sama“ počáteční naplnění indexu i jeho zvětšování.

❖ U dynamické struktury – **sou**bor nebo **spojový sezn**am je třeba testovat koncovou značku `EOF` nebo `NIL`. K tomu, abychom mohli zpracování proudu algoritmizovat strukturovaně a vyhnout se potřebě vyskakovat z cyklu zpracování při „pozdním“ objevení koncové značky je třeba užívat tak zvanou **zásadu čtení s předstihem** [read ahead]. Ta požaduje, abychom prvé čtení položky proudu provedli na začátku zpracování, před testem zda provést krok iterace. Další čtení pak provedli ihned poté, co jsme předchozí položku zpracovali a již ji nepotřebujeme. To nám umožní mít k dispozici informaci o případné zarážce včas, dříve než budeme rozhodovat zda cyklus ukončit nebo provádět další krok.
Poznámka: Některé programovací jazyky (například TURBO PASCAL) tuto zásadu dodržují skrytě. TURBO PASCAL přečte (bez vyžádání programem) prvou položku proudu automaticky. Příkaz (procedura) **`read`** *pak pracuje tak, že tuto položku z uživateli nepřístupné skryté proměnné přenese do proměnné, která je parametrem procedury* **`read`** *a poté ihned naplní skrytou proměnnou novou položkou proudu.* Při *čtení se plní logická proměnná* **`EOF`***, která nabývá hodnoty* **`TRUE`** *právě když byla přečtena zarážka do skryté proměnné. Toto čtení je o záznam napřed oproti proměnné, kterou zpřístupňuje procedura* **`read.`** *Tuto hodnotu lze testovat pro ukončení cyklu. To je pro uživatel „pohodlnější“, ale „málo pedagogické“. V původním návrhu jazyka PASCAL profesora Wirtha, který měl na zřeteli spíše pedagogické než praktické aspekty jazyka, byla možnost užít proceduru* **`read`** *pouze alternativní. Základním prostředkem pro zpracování proudu bylo čtení přímo z proudu příkazem* **`get`**, *který TURBO PASCAL nezná.*

Tento velmi zjednodušený návod (u jen trochu složitějších úloh mohou samozřejmě nastat další problémy) budeme ilustrovat na dvou příkladech. Jednom zcela primitivním, druhém (poněkud) složitějším.

Příklad 1

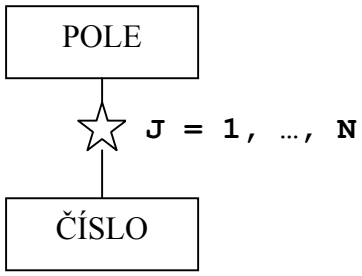
Jsou dána celá čísla. Úkolem je vypočítat součet těch čísel která jsou kladná a počet čísel která jsou záporná.

Toto zadání je pochopitelně neúplné. Není jasné jak čísla do zpracování vstupují. Zda jde o homogenní či nehomogenní strukturu. Budeme úlohu řešit ve dvou variantách.

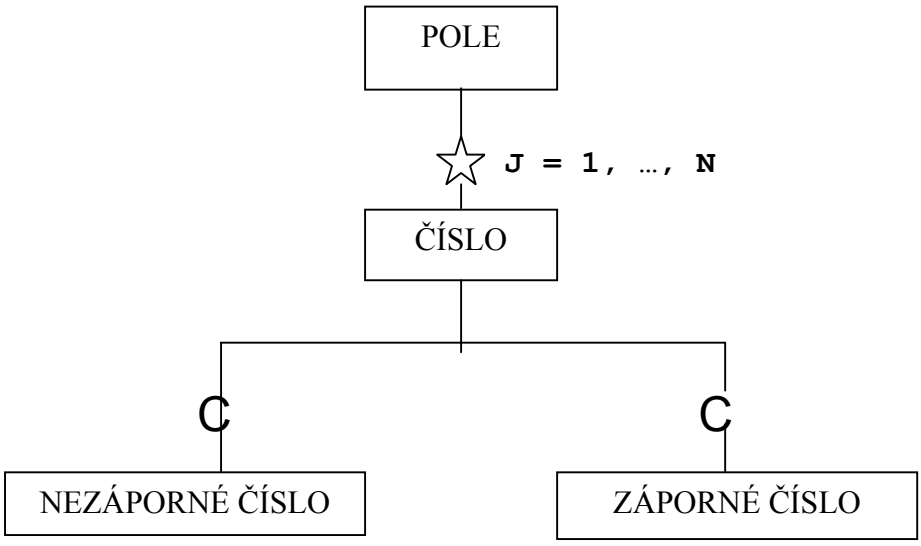
Varianta A

Čísla vstupují do zpracování jako pole známé délky *N*. Lze je indexovat A[1], A[2], …, A[N].

Fyzická struktura dat je jednoduchá:

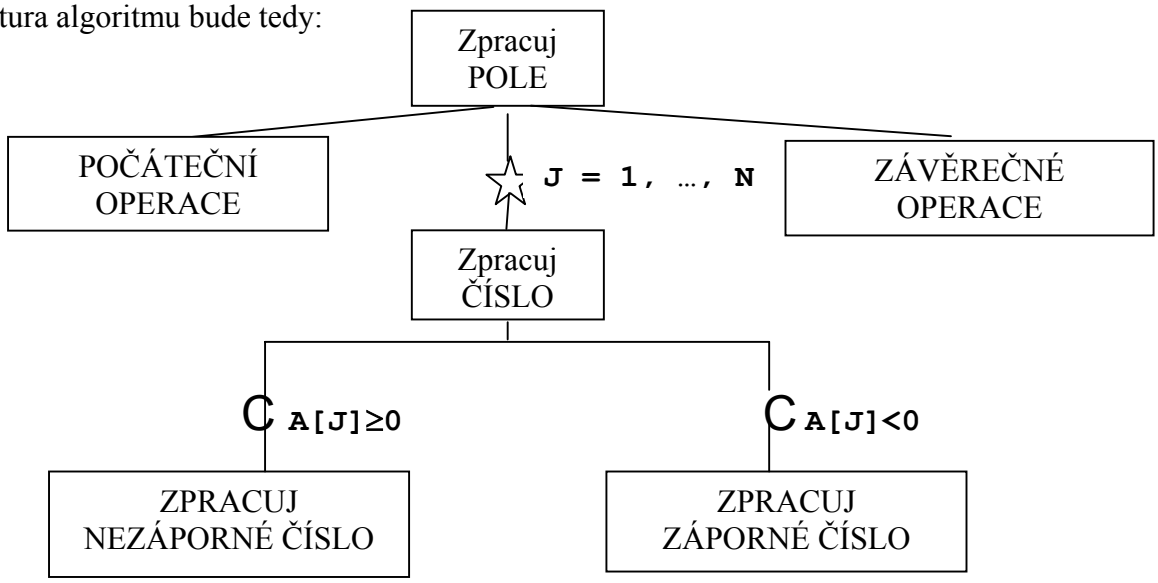


Protože kladná a záporná čísla mají být zpracovávána každé jinak, je třeba tyto fyzickou strukturu obohatit o logický rozdíl, který se má ve zpracování uplatnit. Je třeba se zamyslet nad tím, jak si poradit s hodnotou 0. Ta se nemá započítávat ani do součtu, ani do počtu. Zdá se tedy, že pro ni je třeba počítat s třetí alternativou. To však nutné není. Přičtení nuly totiž součet nezvýší. Můžeme si tedy náš úkol zjednodušit tím, že budeme počítat místo součtu všech kladných čísel součet všech nezáporných čísel. Schéma logické strukturu dat, která do algoritmu vstupují je tedy toto:



Výstupem mají být pouze dvě čísla. Není tedy nutné při přípravě struktury algoritmů brát na výstup zřetel. Stačí vyjít ze struktury dat a přidat do ní nutné operace, které bude třeba zařadit na začátek a na závěr zpracování.

Struktura algoritmu bude tedy:



Nyní již stačí zvážit co bude třeba zajistit na začátku zpracování, co v závěru a jak ošetříme iteraci.

Úkol zní určit součet a počet více čísel. Tyto hodnoty bude třeba střádat ve dvou proměnných. Pojmenujme je například:

SKL … pro součet kladných čísel,

PZAP … pro součet záporných čísel.

Oba tyto střadače bude na počátku nulovat a v každém kroku iterace je příslušně měnit. Index **J** v poli se musí na začátku položit roven 1 a po provedení každého kroku iterace zvětšovat o 1. V závěru zpracování bude třeba hodnoty **SKL** a **PZAP** zobrazit (případně s příslušným vysvětlujícím textem.

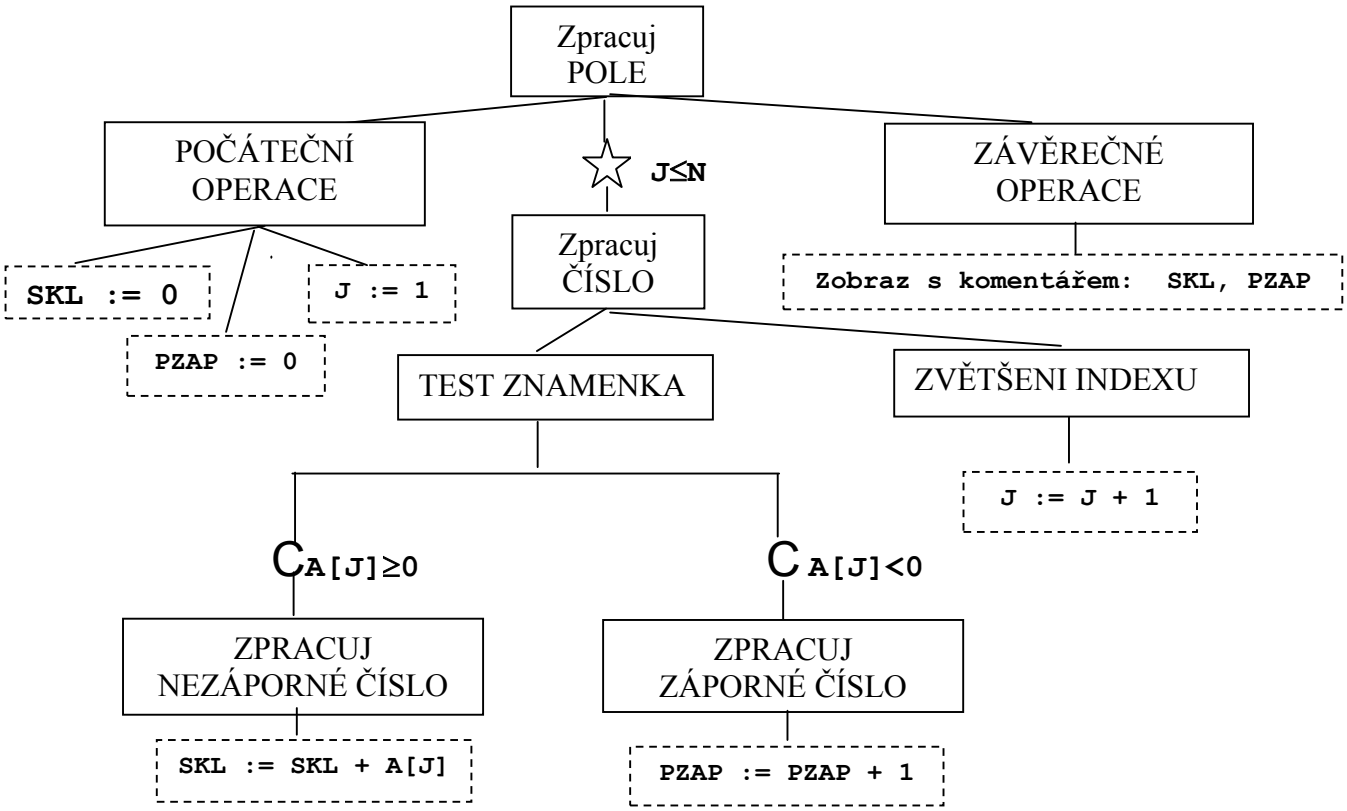
Nulování proměnných, plnění indexregistru číslem 1, přičítání čísla k proměnným, zvětšování proměnných o 1 a testy na nerovnost dvou čísel budeme považovat již za dále nedělitelné kroky algoritmu. Vyšší programovací jazyky mají pro tyto operace jednoduché příkazy. Při programování v jazyce symbolických adres strojového kódu nebude potíž tyto příkazy rozložit každý do několika málo strojových instrukcí. Poznamenejme, že při užití vyšších programovacích jazyků bychom pravděpodobně měli k dispozici prostředky jak iteraci programovat tak, že bychom nemuseli obsluhovat cyklus sami (plnit počáteční hodnotu a zvyšovat index). To by bylo možné vyžádat jediným příkazem typu

```
for J := 1 to N do
```

Nepoužili jsme jej spíše z pedagogických důvodů a pro to, abychom ukázali jak lze kroky algoritmu přepsat do strojového kódu.

Malý problém tvoří zařazení příkazu zvyšování indexu J. Mohli bychom jej zařadit dvakrát za zpracování nezáporného čísla i za zpracování záporného čísla. Profesionálnější je však zařadit zvyšování indexu pouze jednou. V sekvenci za selekci, která nezáporná a záporná čísla rozlišuje.

Po zařazení výkonných příkazů pro kroky algoritmu (v čárkovaných obdélnících) bude strukturogram algoritmu následující: Obdélníky v stromovém grafu označené nepřerušovanou čarou jsou bloky příkazů – programové celky. Je důležité je vhodně pojmenovat, tak aby orientace v strukturogram byla snadná. Čárkovaně orámované listy stromového grafu jsou výkonné příkazy. Tedy dále nedělené kroky algoritmu.

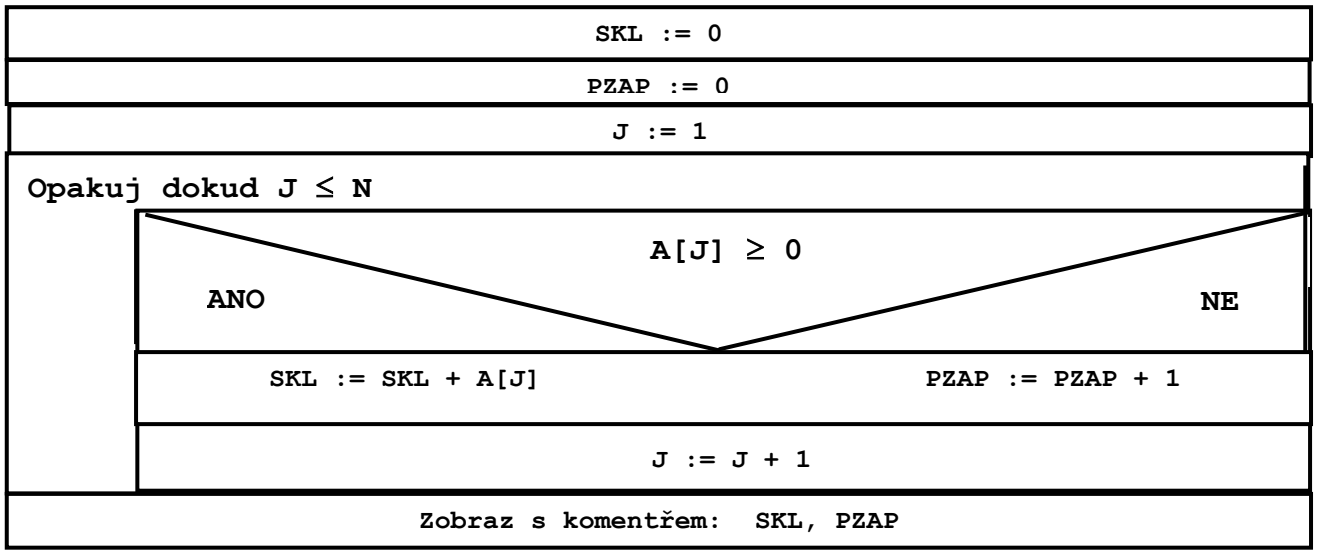


Vidíme, že i u tak jednoduché úlohy, jako je ta naše, je již obrázek dosti zaplněný.
U složitějších úloh lze doporučit nepsat do schématu texty výkonných příkazů, ale jako listy uvádět pouze čísla 1, 2, … . Právě tak podmínky u selekcí a iterací nevypisovat, nahradit je symboly C , C2, … a seznam příkazů a podmínek zapsat nezávisle, mimo obrázek.

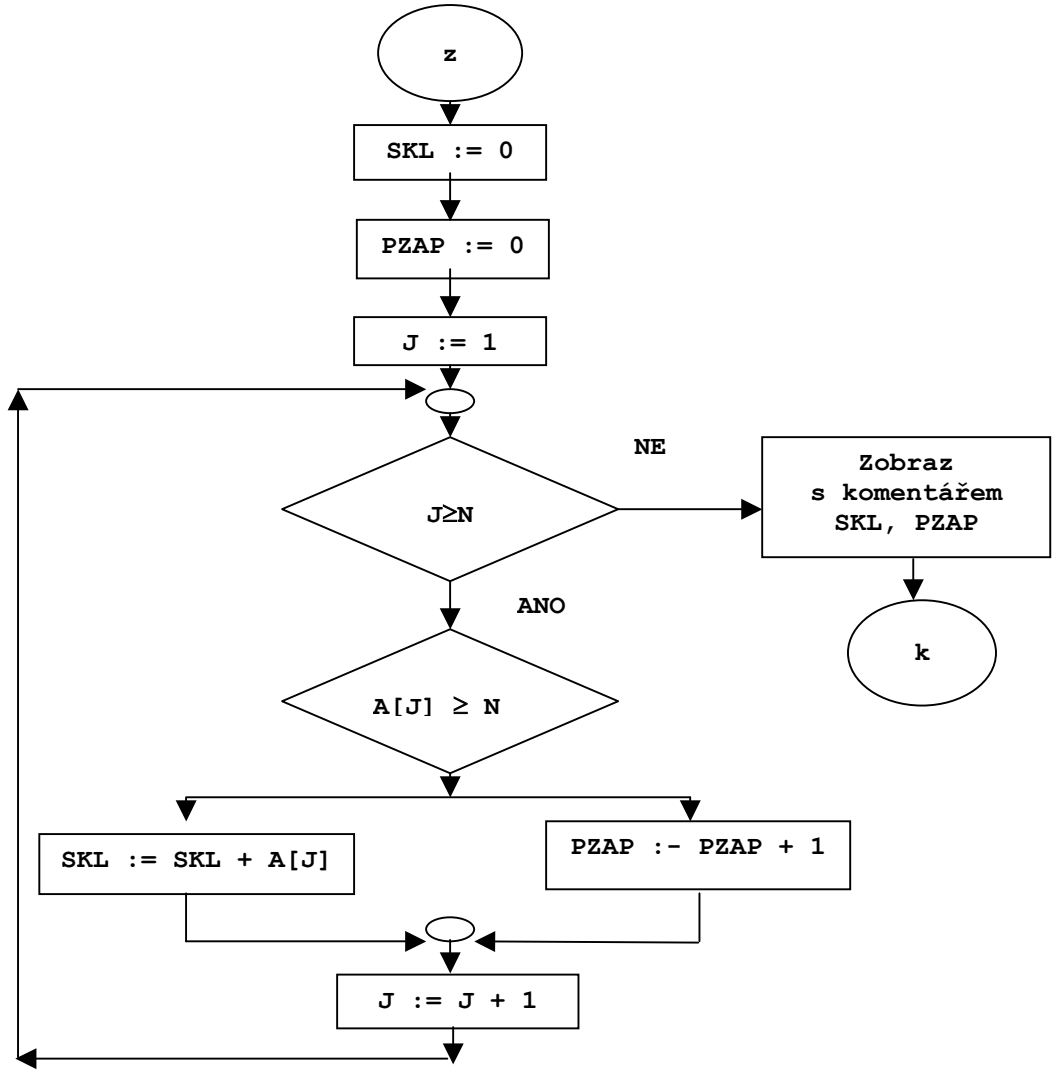
Jednoduchý příklad jsme rozepsali snad zbytečně podrobně. Bylo tomu tak proto, že podobný postup lze užít i pro případy složitější. Pouze výkonných příkazů bude třeba víc.

Pro úplnost uvedeme ještě plošný strukturogram a vývojový diagram téže úlohy.

Plošný strukturogram:



Nakreslíme pro úplnost ještě vývojový diagram:



V jazyce TURBO PASCAL by algoritmu odpovídal s využitím cyklu **`while`** tento text programu:

```
begin
   SKL := 0;
   PZAP := 0;
   J:= 1;
   while (J >= N) do
      begin
         if (A[j] >= 0) then SKL := SKL + A[J]
                        else PZAP := PZAP + 1;
         J := J + 1
      end;
   writeln('Součet kladných je: ', SKL, 'Součet záporných   je: ', SZAP)
end
```

S využitím cyklu `for`:

```
begin
   SKL := 0;
   PZAP := 0;
   J:= 1;
   for J := 1 to N do
               if (A[j] >= 0) then SKL := SKL + A[J]
                              else PZAP := PZAP + 1
      end;
   writeln('Součet kladných je: ', SKL, 'Součet záporných   je: ', SZAP)
end
```

Napíšeme ještě program jazyce symbolických adres pro náš hypotetický počítač. Nepůjde opět o úplný program, ale pouze o jeho úsek. Budeme předpokládat, že pole **A** délky **N** je již naplněno a že adresy **SKL** a **PZAP** jsou k dispozici, právě tak jako pomocná adresa **JEDNA**. Vyhneme se také problému zobrazení výsledku. Budeme předpokládat, že ten zabezpečíme voláním podprogramu, který to provede.

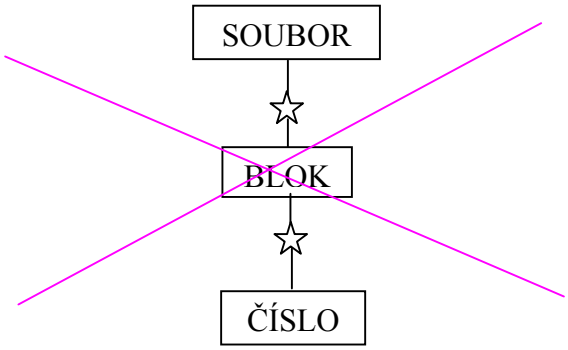
…

```
                LNA   1            % pomocna operace ulozeni cisla 1
                STA   JEDNA        % na adresu JEDNA
                LNA   0
                STA   SKL          % nulovani stradace suctu kladnych
                STA   PZAP         % nulovani stradace poctu zapornych
                LNA   1
                STI                % nastaveni indexregistru na 1
DALSI_CISLO:    SUB   N            % odecteni N od stradace
                JZ    KONEC        % pokud N > J, potom konec cyklu
                LDA   A[IND]       % v IND je vzdy aktualni hodnota J
                JZ    ZAPORNE      % pro zaporne preskoc zprac. kladnych
                LDA   SKL
                ADD   A[IND]       % pricteni kladneho cisla k SKL
                STA   SKL
                JMP   ZVEC_IND     % preskoci zprac. záporných
ZAPORNE:        LNA   1            % cislo 1 do stradace
                ADD   PZAP         % pricteni k PZAP
                STA   PZAP
ZVEC_INDEX:     LDA   0[IND]       % obsah indexregistru do stradace
                ADD   JEDNA
                STI                % novy obsah indexregistru
                JMP   DALŠÍ_CISLO % skok na test zda opakovat krok iterace
KONEC:          volání podprogramu pro zobrazení výsledků
```

…

Varianta B

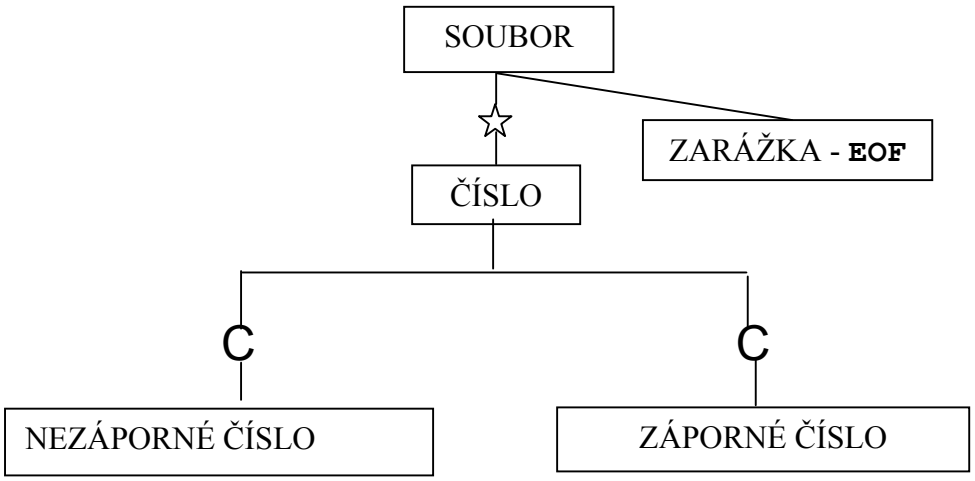
Čísla do zpracování vstupují jako **proud**. Jsou uložena jako **soubor** na vnější paměti. Konec souboru lze identifikovat koncovou značkou **EOF** za posledním záznamem (číslem). Fysická struktura vstupu na vnější paměti je:



Ta je ovšem pro nás „neviditelná“. Jsme od ní odstíněni tím, že vstup a výstup zajišťuje operační systém. Ten nám umožní užívat „logické čtení“ a nestarat se o to, zda vyvolá ve skutečnosti „fyzické čtení“ dat do vyrovnávací paměti nebo jen odebírání vstupu z vyrovnávací paměti a příslušný posun ukazatele v této paměti.

Pro návrh našeho algoritmu nemá tato struktura žádný význam.

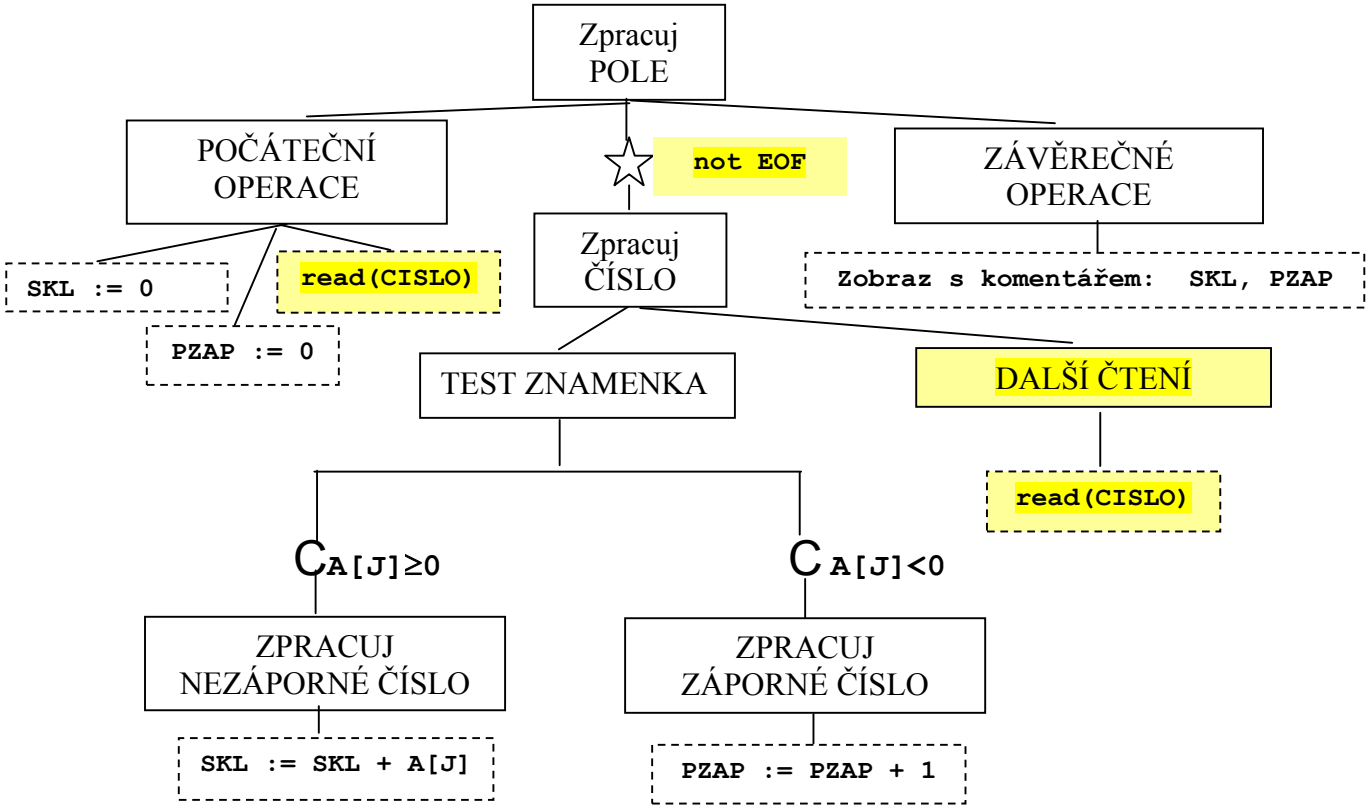
Pro nás je důležitá **logická struktura** vstupních dat. Ta je:



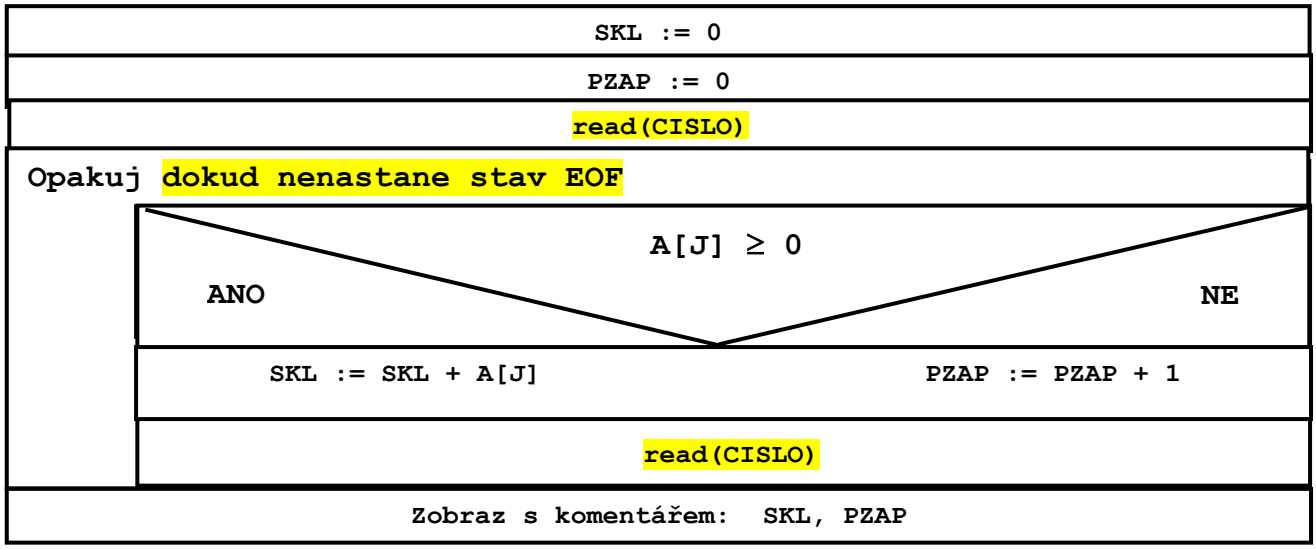
Při návrhu algoritmu postupujeme stejně jako u prvé varianty. Musíme ale zvážit tyto rozdílu.

- Přístup k prvkům pole nebude prostřednictvím indexu. Přístup bude postupný. Vždy bude k dispozici pouze jedno číslo, získané příkazem **`read`**. Ten (logicky) přečte a dá k dispozici do proměnné **`CISLO`** jedno číslo ze vstupu.
- Konec cyklu nelze řídit tím, že index je v rozsahu pole (**`J ≤ N`**), ale zarážkou. Při jejím přečtení se podmínka **`EOF`** nastaví na hodnotu **`TRUE`**. Pokud **`EOF`** nenastane, bude mít hodnotu **`FALSE`**.

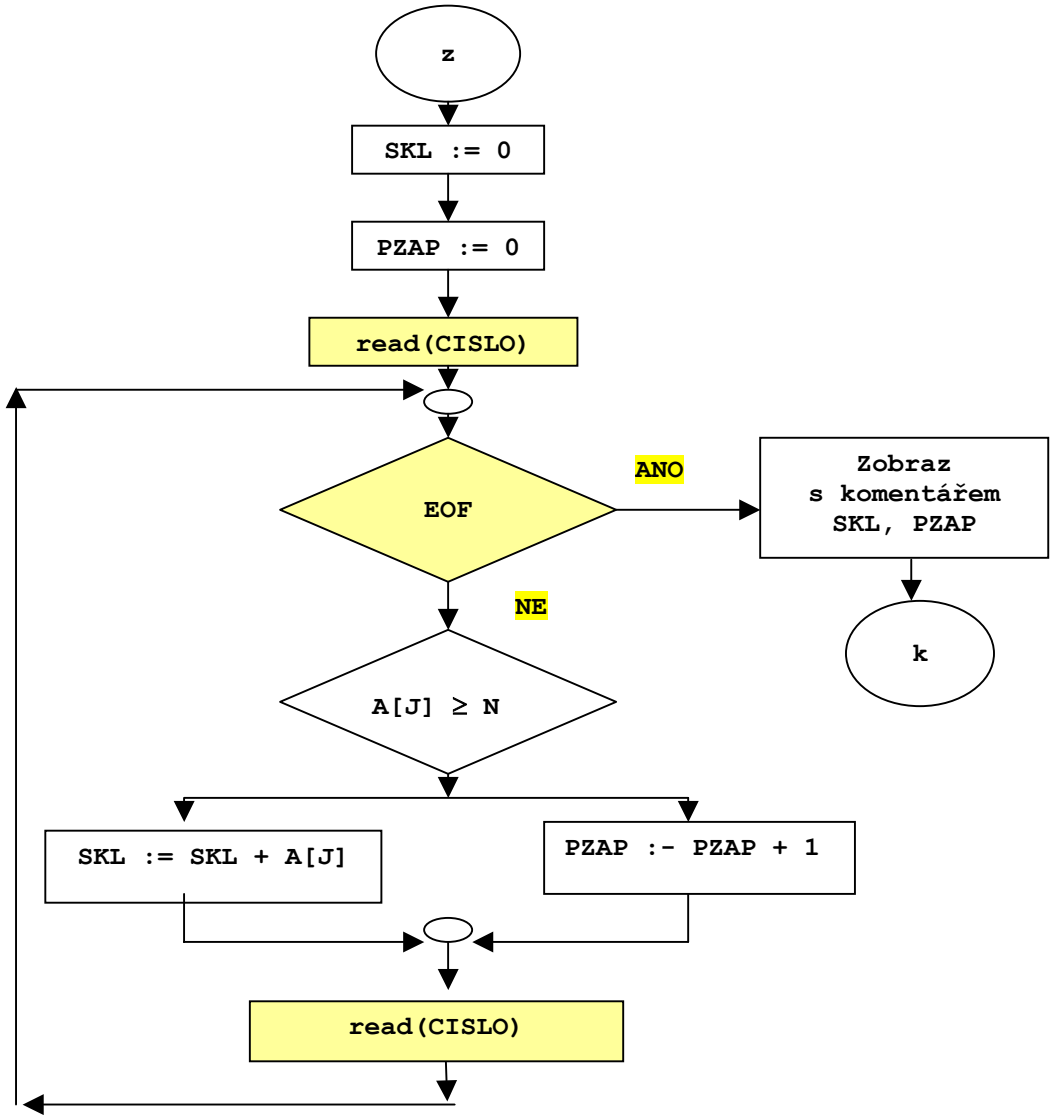
Pro umístění příkazů **`read(CISLO)`** využijeme čtení s předstihem. Poprvé na začátku, znova ihned jakmile předchozí číslo zpracujeme a již nebudeme potřebovat. Stromový strukturogram algoritmu bude (bloky a podmínky v kterých se algoritmus liší od varianty A jsou zvýrazněny žlutě):



Plošný strukturogram:



Vývojový diagram:



Napíšeme ještě příslušný úsek programu v jazyce TURBO PASCAL:

```
begin
   SKL := 0;
   PZAP := 0;
   read(CISLO);
   while (not EOF) do
      begin
         if (A[j] >= 0) then SKL := SKL + A[J]
                        else PZAP := PZAP + 1;
         read(CISLO)
      end;
   writeln('Součet kladných je: ', SKL, 'Součet záporných   je: ', SZAP)
end
```

Cyklus **`for`** zde pochopitelně užít nelze.

Poznamenejme, že pokud by vstup tvořil proud dat v hlavní paměti realizovaný dynamickou strukturou se sekvenčním přístupem spojový seznam, tvořený položkami tvaru

byl by algoritmus stejný. Pouze bychom testovali zda nemá tento odkaz hodnotu **`NIL.`**

Poznámky:

Velice podobně bychom postupovali při návrhu algoritmu, který by měl zjistit jiné charakteristiky vstupního pole čísel. Například, kdybychom hledali **extrémy** (minimum a/nebo maximum), užili bychom tutéž strukturu dat i algoritmu. Rozdíl by spočíval pouze v tom, že místo počátečního nulování střadačů bychom do proměnných uložili prvý prvek pole nebo proudu. V cyklu bychom s tímto „kandidátem na vítězství“ porovnávali postupně další prvky pole. Pokud by byli „lepší“ (menší v případě minima,. Větší v případě maxima), provedli bychom výměnu kandidáta na vítězství. Po ukončení cyklu by zde byl extrém.

Formálně jde někdy tento postup zjednodušit tak, že cyklus porovnávání provedeme přes všechny prvky pole (včetně prvého prvku). V tom případě je třeba na počátku do proměnné pro hledaný extrém uložit hodnotu, která „určitě nevyhraje“. Hledáme-li maximum z kladných čísel tak třeba nulu.

Takový algoritmus nalezne ten extrém, který je v poli nebo proudu prvý. Pokud by byla úloha složitější a požadovala nalézt třeba všechny výskyty extrému, je-li jich více, je potřeba postupovat takto: Zřídíme pomocnou proměnnou **`P_extremu`** a pomocné pole **`vitezove`** délky rovné maximálnímu počet stejných prvků na vstupu (nebo zásobník). Při zkoumání prvků pole v cyklu rozlišíme tři případy (naznačíme to pro hledání maxima):

1. **`A[J] > MAX`** – pak : **`begin P_extremu := 1; MAX := A[J]; vitezove[1] := J end`**
2. **`A[J] = MAX`** – **`begin P_extremu := P_extremu + 1; vitezove[P_extremu] := J end`**
3. **`A[J] < MAX`** – *neprováděj žádnou činnost.*

Při jiné modifikaci úlohy, kdybychom chtěli například znát tři největší prvky, byl by algoritmus podobný. Prováděli bychom v každém kroku cyklu postupně tři testy **`A[J] > PRVY,`** pokud ne, tak test **`A[J] > DRUHY,`** pokud ne, test **`A[J] > TRETI.`** Při kladném výsledku testu potřebné přesuny na „stupních vítězů“ (pozor na pořadí těchto přesunů!). Proveďte to!

Příklad 2

Uvedeme ještě případ, kdy sestavení vhodné logické struktury dat tak, aby vyhovovala požadavkům úlohy, vyžaduje trochu hlubší zamyšlení. Taková úvaha nad daty, dříve než se pustíme do návrhu algoritmu se však vždy vyplatí.

Vstupem naší úlohu bude text (například nějaké knížky, skript nebo článku). Text se skládá ze znaků, které mají nějakou grafickou representaci – tzv. „grafémů“ (písmenka, číslice, diakritická znaménka …) a oddělovačů (mezera, tvrdé mezery, tabelátory, přechody na nový řádek, novou stránku a podobně). Vstup tvoří proud a příkazem **`read`** máme možnost získat k dispozici vždy jeden znak – grafém nebo oddělovač. Konec „díla“ je signalizován speciálním znakem **`EOF`** (zarážkou) na vstupu. Před prvým zobrazitelným znakem mohou ale nemusí být různé oddělovače, mezi posledním zobrazitelným znakem a zarážkou mohou (ale nemusí) být také oddělovače.

Pro výpočet autorského honoráře je běžně základem počet znaků díla **`PZ`** definovaný tak, že znakem je každý zobrazitelný znak (včetně interpunkce) a každá skupina navzájem bezprostředně následujících oddělovačů. Tedy jedna mezera mezi slovy se počítá jako jeden znak, padesát mezer a přechod na novou stránku také jako jeden znak přispívající do **`PZ`**. Tato definice brání možné snaze autorů zvyšovat svůj honorář nadbytečným plýtváním místem, psaním „do veršů“ apod. Takové mravy jsou sice možné, autorskou odměnu však nezvýší. Jednotka pro výpočet honoráře – autorský arch – **`AA`** je definována jako **`AA = PZ / 36000`** (20 stran po 30 plně obsazených řádcích, každý délky 60 znaků).

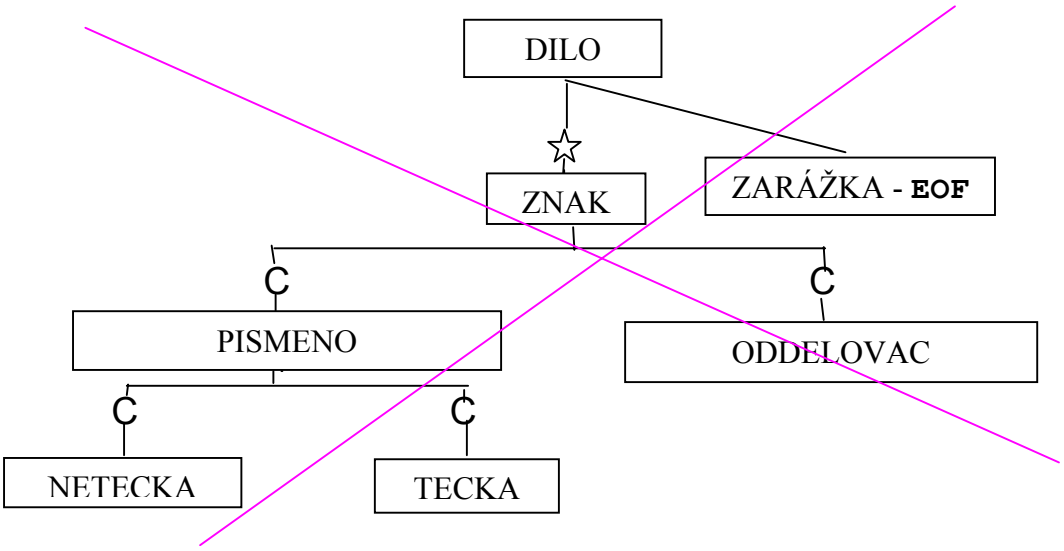
Máme navrhnout algoritmus (a posléze sestavit program), který přečte celé dílo a zjistí:

1. Počet autorských archů díla.
2. Počet slov v díle. Za slovo budeme považovat skupinu bezprostředně následujících zobrazitelných znaků, oddělenou od další takové skupiny jedním oddělovačem nebo skupinou oddělovačů).
3. Průměrný počet slov v jedné větě. Za větu považujeme pro jednoduchost (tuto definici lze důvodně kritizovat) část textu ukončenou tečkou (znak „**.**“). Poznamenejme, že tento údaj je dobrým predikátorem čitelnosti textu. Čím delší věty, tím nižžší srozumitelnost.

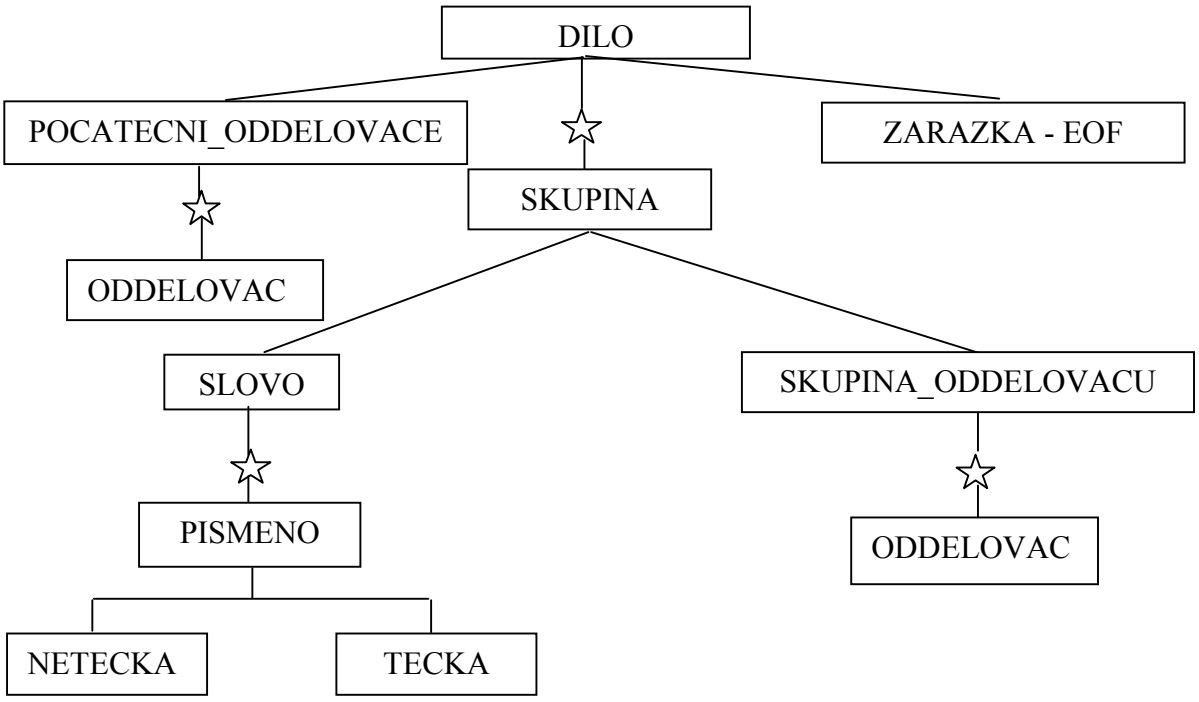
Je jasné, že budeme potřebovat znát

- Počet zobrazitelných znaků (říkejme jim pro jednoduchost písmena). **- `PP`**
- Počet skupin oddělovačů za slovy. **- `PSODD`**
- Počet slov. **- `PS`**
- Počet teček. **- `PT`**

Prvá struktura dat, která se nabízí je obdobná jako u příkladu 1, varianta B:



Tato struktura by nám však pro náš úkol nebyla příliš platná. Náš úkol je rozlišovat skupiny písmen, která za sebou bezprostředně následují a (slova) a skupiny oddělovačů mezi slovy.Tyto dva typy skupin znaků se vzájemně pravidelně střídají. Sdružíme tedy za sebou vždy slovo a následující oddělovače. Těmto dvěma skupinám budeme říkat **`dvojice`**. Dílo pak bude iterací dvojic, následovanou zarážkou **`EOF`**. Výjimku tvoří pouze možné oddělovače na začátku textu. Před prvým písmenem. Ty se nikam nezapočítávají a musíme je při zpracování „přeskočit“. Budeme tedy zpracovávat následující strukturu „**logickou z hlediska požadovaného zpracování**“.



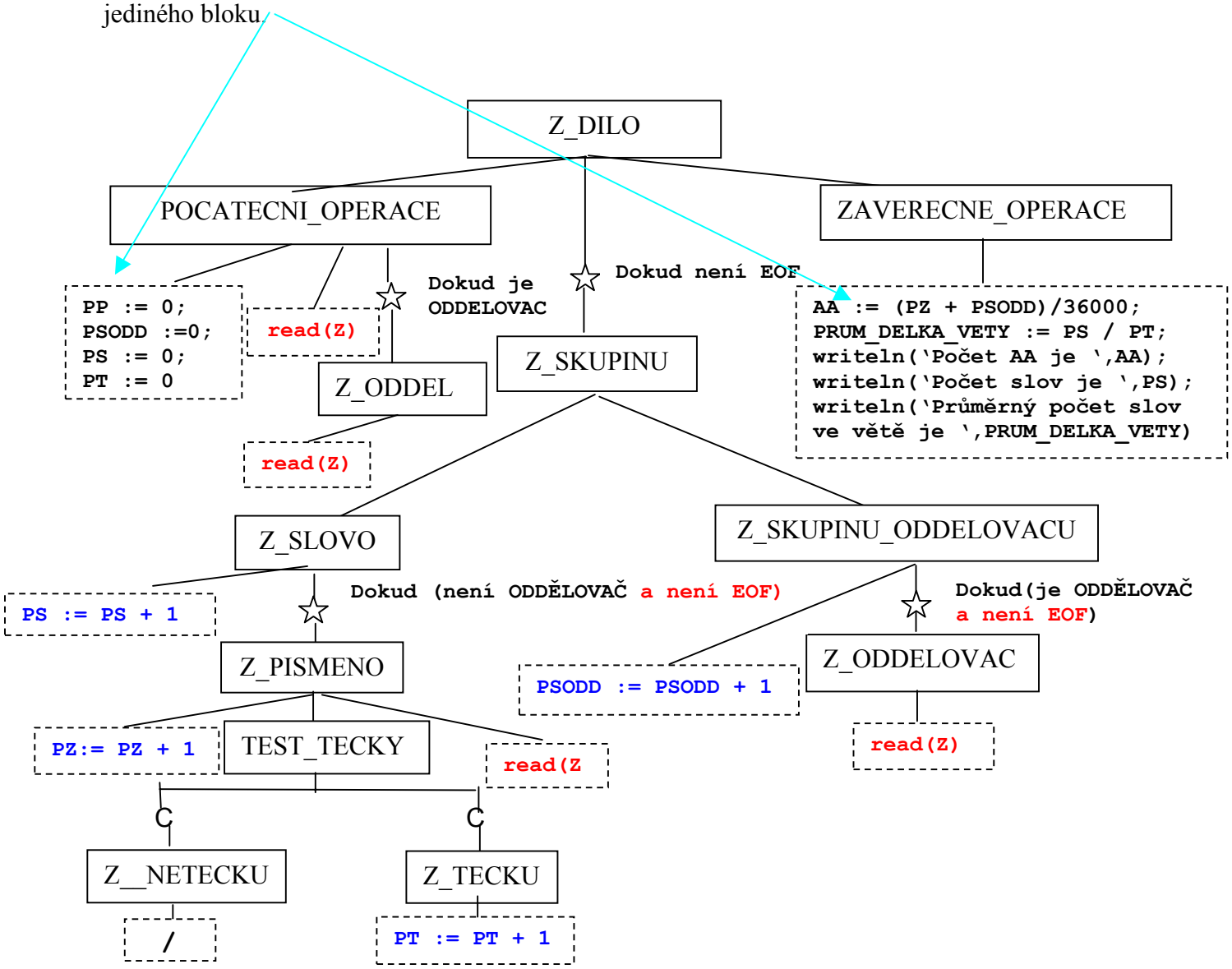
Nyní již je vše velmi snadné. V podstatě půjde již jen o mechanickou práci.

Strukturogram dat, pokud jde o hlavní bloky, lze v podstatě zkopírovat na strukturogram algoritmu. Pouze z důvodu srozumitelnosti lze doporučit přejmenovat bloky tak aby napovídali, že jde ne o data, ale o akce s daty. Uděláme to předponou Z (zpracuj!) a pro úsporu místa zkrátíme některé příliš dlouhé názvy.

Potřebné střadače počtu entit které nás zajímají: **PP, PSOD, PS** a **PT** budeme pochopitelně nulovat na začátku zpracování (spolu s přeskočením počátečních oddělovačů – uvědomme si, že iterace typu **while** může být prázdná, tedy algoritmus bude v pořádku i když na začátku oddělovače nebudou). V závěru provedeme potřebné výpočty.

Příkazy čtení umístíme pochopitelně v souladu se zásadou čtení s předstihem. Povšimněte si pečlivě jejich umístění!

Využijeme i konvence, podle které lze zapisovat sekvenci jednoduchých příkazů téhož druhu (například přiřazovacích příkazů nebo aritmetických příkazů) pro úsporu místa pod sebe do jediného bloku.



Povšimněte si podmínek u iteracích na vnořené úrovni. Zde je nutné testovat nejen konec slova nebo konec skupiny mezer, ale i zarážku **EOF** (zvýrazněno červeně). Obecně platí, že u podřízených iterací (vnořených cyklů), je vždy nutné podmínky pro opakování vnějšího cyklu přenést formou konjunkce i do podmínek pro vnořený cyklus. Opomenutí tohoto přenosu bývá častou příčinou chyb.

Povšimněte si též umístění příkazů pro zvyšování počítadel znaků, teček, slov a vět. Vyznačeno modře. Jsou vždy na příslušné úrovni ve struktuře dat.

Případný přepis do plošného strukturogram a nakreslení vývojového diagramu je již jen mechanickou práci. Přesto lze vřele doporučit.

Čtenáři přenecháme i přepis do programovacích jazyků. K tomu jen několik poznámek: Testy Booleovských hodnot **JE_ODDELOVAC** a **Není_ODDELOVAC** v jazyce TURBO PASCAL bude patrně nejvýhodnější provést deklarací konstanty **ODDELOVACE** typu **set of char**, do které uložíme všechny řídicí znaky, které se mohou v textu vyskytnout a které považujeme za oddělovače (mezery, nové řádky, …). Pro test pak užijeme logickou hodnotu získanou operací **Z in ODDELOVACE**. Pokud jazyk obdobnou možnost nemá, je nutné pro vyhodnocení této podmínky užít řetězec testů, při kterých se zkoumá, zda kódová reprezentace čteného znaku odpovídá kódu některého znaku, který považujeme za oddělovač. To by bylo patrně výhodné připravit a volat jako podprogram.

Danou úlohu by jistě bylo možné řešit i jinak, bez přímé vazby na podrobné rozkreslení logické struktury vstupních dat. Naznačený postup ale představuje návod, při kterém je možné úspěšný návrh garantovat, aniž by autor musel spoléhat na to, že dostane ten „správný nápad“, jak na to. Je to „sázka na jistotu“. Proto ji lze doporučit.

Má i výhody, že při dobré dokumentaci je postup snadno srozumitelný pro kohokoli. Případné úpravy a doplňky v něm lze snadno realizovat. Ukážeme si to na našem řešení.

Představme si, že bychom chtěli náš algoritmus změnit tak, aby kromě funkcí, které již má zjišťoval ještě frekvence užití slov dané délky. Tedy kolik slov v textu je jednopísmenných, kolik dvoupísmenných, kolik jich má tři písmena, … .

Stačí si zřídit pole **FREKVENCE** takové délky, která odpovídá nejdelšímu možnému očekávanému slovu. Jistě bude stačit délka 30. Toto pole budeme nulovat (cyklem o 30 krocích) v bloku POCATECNI_OPERACE. Jeho závěrečný stav vypíšeme v bloku ZAVERECNE_OPERACE.

Zřídíme dále proměnnou **PPSL** pro počet písmen ve slově. Tuto proměnnou budeme nulovat vždy na začátku slova. Příkaz **PPSL := 0** tedy umístíme pod blok Z_SLOVO, co nejvíce vlevo.

Zvyšování **PPSL** o jedničku provedeme u každého písmena, které není tečkou (kdybychom chtěli být přesní, vyloučíme ještě čárky, středníky dvojtečky, … umístěné ihned za posledním písmenem slova). Tedy **PPSL := PPSL + 1** zařadíme pod blok Z_NETECKU. Po ukončení zpracování slova Zvětšíme příslušnou položku pole **FREKVENCE** (s indexem podle délky ukončeného slova tedy podle získané hodnoty **PPSL**) o jedničku. Příkaz **FREKVENCE[PPSL] := FREKVENCE[PPSL] + 1** tedy zařadíme pod blok Z_SLOVO, jako poslední příkaz vpravo.

Tím je rozšíření funkce algoritmu provedeno, bez jakýchkoliv potíží.

Kdo pečlivě celý postup sledoval, jistě ocení výhody stromových strukturogramů oproti jiným grafickým prostředkům znázornění algoritmů, pokud jde o možnost doplňování akcí a pojmenovávání jednotlivých úseků algoritmu.