4. MODULÁRNÍ VÝSTAVBA PROGRAMOVÝCH SYSTÉMŮ

Pojem "modulární výstavba systému" je v současné době již dostatečně široce známý z různých technických oborů a řada čtenářů má dobrou představu o modulární výstavbě různých elektronických zařízení. Smyslem modulární výstavby je rozčlenit složitý systém na soubor spolupracujících podsystémů - modulů, kde každý modul je samostatný celek plnící jistou funkci a komunikující se svým okolím (s ostatními moduly nebo s okolím systému) jen nejnutnějšími, přesně definovanými vazbami. Tyto vazby členíme ve vztahu k modulu na vstupní (dovozní, importující) a výstupní (vývozní, exportující). Stejný význam má modulární výstavbě programových systémů, ve které členění rozsáhlejšího programu na programové moduly sleduje především rozdělení velkého problému na řadu malých, vzájemně spolupracujících malých problémů. Účelem je zvýšení srozumitelnosti, spolehlivosti, udržovatelnosti, modifikovatelnosti a stavebnicovosti programového systému, za cenu nutnosti použití aparátu pro výstavbu modulů, se kterým se programátor musí seznámit a jehož použití vede k jisté (ne však významné) dodatečné spotřebě počítačového času i paměti pro modulárně uspořádaný program. Cílem této kapitoly je seznámit se s prostředky pro modulární výstavbu programu, které jsou nadstavbou jazyka Pascal EC tak, aby mohly být v další kapitole použity při implementaci abstraktních typů dat a aby je mohli studenti používat při řešení svých problémů v rámci předmětu Programovací techniky, ale i v rámci dalších programovacích a diplomních projektů. Důvody, které vedou k modulárnosti programů i způsoby, jak dekomponovat programový systém na jednotlivé moduly, budou podrobněji rozvedeny v kapitolách třetí části skript.

4.1. POJEM PROGRAMOVÝ MODUL

4.1.1. Vymezení základních pojmů

S pojmem modul se v oblasti výpočetní techniky setkáváme jak v problematice technické konstrukce počítače, tak při vytváření jeho programového vybavení. Protože má v těchto oblastech odlišný význam, bývá zvykem hovořit o počítačovém modulu a o programovém modulu. Pojem programový modul nebývá chápán jednotně a v literatuře se setkáváme s celou řadou různých definic pojmu programový modul. Společným rysem těchto definic je to, že chápou modul, jako určitou část programového systému. Tato část musí být dostatečně rozsáhlá vzhledem k tomu, aby mohla být relativně nezávislá, musí však být přiměřeně rozsáhlá, aby si zachovala přehlednost a srozumitelnost. Rozdíly nastávají při vymezení části programového systému jako programového modulu. Můžeme se setkat s požadavkem vymezujícím rozsah modulu určitým počtem (např. 100) příkazů, nebo s hlediskem provozním, vymezujícím modul jako samostatně překládaný celek apod. Vymezme proto pojem programový modul tak, jak jej budeme dále používat.

Programový modul (dále jen modul) je část programového systému, která má definované jméno a rozhraní (interface) a skládá se z množiny proměnných a množiny operací.

Každá proměnná modulu je určitého typu a vzhledem k modulu může být :

- veřejná proměnná, jejíž hodnota je přístupná nejen uvnitř operací daného

modulu, ale i uvnitř operací jiných modulů. Novou hodnotu však může získat pouze prostřednictvím operací svého modulu.

Pozn.: Tato vlastnost je významná pro modulární výstavbu a uplatní se především při ladění, kdy zodpovědnost za stav veřejné proměnné se lokalizuje na rozsah modulu.

- soukromá proměnná, jejíž hodnota je přístupná pouze uvnitř operací daného modulu, a které mohou její hodnotu měnit.

Typ proměnné modulu může být :

- konkrétní, a pak je vyjádřen přímo prostředky programovacího jazyka
- abstraktní, to znamená, že jeho struktura je "skryta" za identifikátorem.

Operace modulu (operací se rozumí souhrnné označení pro proceduru nebo funkci) realizuje určitý algoritmus, který na základě hodnot proměnných modulu a hodnot vstupních parametrů mění hodnoty proměnných modulu a ustavuje hodnoty výstupních parametrů. Operace modulu můžeme rozdělit na :

- veřejné operace, které lze použít jak v operacích daného modulu, tak v operacích jiných modulů
- soukromé operace, které lze použít pouze uvnitř operací daného modulu

Definice operace modulu se skládá ze dvou částí :

- specifikace operace, která určuje syntaktické a sémantické vlastnosti operace (vstupní a výstupní argumenty operace a jejich funkci). Sémantika operací může být vyjádřena neformálně (přirozeným jazykem) nebo neproceduralně vhodným specifikačním jazykem
- implementace operace, která představuje vyjádření algoritmu operace prostředky konkrétního programovacího jazyka ve shodě se specifikací operace

K jedné specifikaci operace lze vytvořit řadu verzí implementace, kde jednotlivé verze se mohou lišit např. v závislosti na operačním systému, ve kterém mohou být použity, nebo tím že preferují rychlost provedení před spotřebou paměti a naopak. Pro další úvahy budeme předpokládat, že pracujeme s jednou jednoznačně přiřazenou implementací operace.

Teprve tím, že modul rozdělíme na část veřejnou a soukromou, získáme nejdůležitější přínos modularity, a tou je oddělení toho co modul dělá od toho, jak to dělá. K separaci a definici těchto funkcí slouží pojem rozhraní.

Rozhraní modulu (také spojení, angl. interface) definuje dvě funkce modulu:

- vývoz modulu (export, to "co modul dělá") zahrnuje množinu všech veřejných proměnných modulu a jejich typů spolu s množinou specifikací všech veřejných operací modulu
- dovoz modulu (import, to "s čím to modul dělá") obsahuje množinu specifikací operací a proměnných spolu se jmény modulů, v jejichž rozhraních jsou definovány [*]. Tyto operace a proměnné lze pak využít při implementaci vlastních operací.

U každého modulu budeme předpokládat existenci dvou standardních soukromých operací :

---

*) Obecnější přístup umožňuje, že se nemusí uvádět jména modulů, z nichž se operace a proměnné dovážejí.

- otevření modulu (initial) je operace, která se provede před prvním použitím ostatních operací modulu, nebo před prvním využitím hodnoty veřejné proměnné modulu. Tato operace nemá parametry a jejím výsledkem je počáteční hodnota všech proměnných modulu, jež by byly jinak nedefinované
- uzavření modulu (final) je operace, která se provede po posledním použití operace nebo proměnné modulu. Tato operace není parametrizována.

Pod pojem proměnné modulu lze zařadit také konstanty modulu. Jde o speciální případ proměnné, která nabude svou první hodnotu na začátku výpočtu (při tzv. "otevření" modulu) a tuto hodnotu si podrží beze změny až do ukončení výpočtu.

Příklad 4.1.

Vytvořme modul LISTING, který bude řídit stránkování tisku výstupních sestav. Pro popis charakteristických rysů modulu použijeme hypotetické formy využívající některých rysů Pascalu.

```
(1)  module LISTING;
(2)  interface  {rozhraní}
(3)      export   {vývoz modulu}
(4)          var PAGENUM: integer;  {pořadové číslo stránky}
(5)          procedure LINE;     {provádí tisk jednoho řádku a případnou
                                  úpravou čísla stránky PAGENUM}
(6)      import  {dovoz modulu}
(7)          from module TITLE;
(8)               procedure HEADER;  {provádí posun na novou stránku a
                         tisk hlavičky (hlavička=angl.header) stránky}
     end; {rozhraní}

(9)  var LINENUM: integer;   {pořadové číslo řádku}
(10) procedure LINE;
         begin LINENUM:=LINENUM+1;
               if LINENUM=55
                    then begin
                              LINENUM:=0;
                              PAGENUM:=PAGENUM+1;
                              HEADER;
                         end;
         end;
(11) initial
         begin LINENUM:=0;
               PAGENUM:=1;
               HEADER;
         end;
(12) final
         writeln ('END OF REPORT');
end. {Modulu LISTING}
```

Modul má definováno jméno LISTING (1), rozhraní modulu (2) rozdělení na část vývozu (3) a dovozu (6). Skládá se z proměnných PAGENUM, LINENUM a z operace LINE. Proměnná PAGENUM je veřejná a je uvedena v části vývozu (4). Její typ je konkrétní (integer). Proměnná LINENUM je soukromá. Operace LINE je veřejná a je uvedena v části vývozu. Má uvedenu samostatně specifikaci (5) a implementaci (10). Pro implementaci modulu LISTING je zapotřebí operace HEADER, kterou vyváží modul TITLE. Tato skutečnost je definována v části dovozu (7), (8). Modul obsahuje dále dvě standardní soukromé operace otevření modulu (11) a uzavření modulu (12). Definice rozhraní je ukončena omezovačem end. Syntax standardních soukromých operací initial a final vyplývá ze zápisu (jméno operace a jeden (případně složený) příkaz - neoddělované středníkem).

Pozn.: Uvedený zápis modulu je příkladem jak by mohla definice modulu vypadat. Syntax zápisu neodpovídá žádnému konkrétnímu systému pro modulární programování.

4.1.2. Obecnější formy modulu

Definice modulu, tak jak byla uvedena na příkladu v předcházejícím odstavci, vyhovuje většině praktických situací. Pro používání modularity na této úrovni, lze najít potřebné prostředky již u takových jazyků jako je Fortran, Cobol, PL/I apod. V některých případech však tato forma modulu již nestačí. Předpokládejme, že máme k dispozici modul LISTING, avšak v budovaném systému budeme souběžně vytvářet dvě různé tiskové sestavy. V tomto případě nelze pro dvě různé sestavy použít jednoho modulu LISTING, ale potřebujeme, aby existovaly dvě samostatné, nezávislé kopie modulu LISTING - tzv. instance modulu.

Dosavadní použití pojmu modul předpokládalo existenci jediné instance modulu. Je zřejmé, že vytváření nové instance modulu skutečným fyzickým zdvojením proměnných i operací modulu není výhodné, a v případě, že o počtu instancí modulu rozhoduje až průběh výpočtu, je to nemožné. Proto se hledá jiné řešení.

Chceme-li vytvořit novou instanci modulu, pak stačí, vytvoříme-li novou kopii proměnných modulu. Operace modulu mohou zůstat beze změny za předpokladu, že bude vždy určeno, se kterou kopií proměnných modulu se má v daném okamžiku pracovat. Lze to zajistit tak, že každá operace bude mít jeden [*] další povinný parametr nazývaný parametr instance. Aby bylo možné vytvářet nové kopie dat modulu je nutné, aby struktura všech proměnných byla definována vně modulu. Předmětem vývozu takového modulu bude místo konkrétních proměnných modulu datový typ modulu, který se skládá z typů všech potřebných proměnných modulu, a to jak z proměnných veřejných, které tvoří veřejnou část datového typu (public), tak ze soukromých proměnných, které tvoří soukromou část datového typu (private). Pokud je soukromá část datového typu neprázdná, hovoříme o abstraktním datovém typu modulu.

Je-li soukromá část datového typu prázdná, říkáme, že jde o konkrétní datový typ. Na základě těchto pojmů můžeme definovat obecnější formu modulu.

Programový multimodul (dále jen multimodul) je část programového systému, která má definováno jméno, rozhraní a skládá se z množiny operací. (V definici multimodulu chybí oproti definici modulu existence proměnných multimodulu. Obecnější definice multimodulu by mohla předpokládat existenci soukromých proměnných multimodulu, které jsou společné všem instancím modulu.) Operace otevření a uzavření jsou u multimodulu (na rozdíl od modulu) veřejné operace. Všechny veřejné

---

[*] Obecně lze použít i několik takových parametrů

operace specifikované v rozhraní multimodulu (tedy i otevření a uzavření) mají nejméně jeden povinný parametr - parametr instance modulu.

Před zahájením využívání multimodulu je nutno vytvořit příslušnou instanci modulu a to buď v době kompilace - jako důsledek deklarace proměnné datového typu modulu, nebo v době výpočtu - ustavením dynamické proměnné datového typu modulu. Na začátku výpočtu (nebo po dynamickém ustavení instance) se provede operace otevření aktuální instance multimodulu. Na závěr práce s danou instancí multimodulu se provede uzavření této instance multimodulu. Pro práci s instancemi multimodulu platí stejná pravidla, jaká jsme zavedli pro modul. Odkazovat se lze z vnějšku pouze do veřejné části instance a zde je možné pouze číst hodnoty veřejných proměnných, ale ne jejich hodnotu měnit.

Příklad 4.2.

Vytvořme modul, který sdružuje operace nad textovým řetězcem. Jde o typický příklad multimodulu, neboť požadujeme, abychom mohli pracovat s více řetězci - instancemi - současně. Popis multimodulu bude opět zapsán prostředky hypotetického (nekonkrétního) systému pro modulární programování tak, aby postihl charakteristické rysy popisu multimodulu.

```
(1)  module STRING;
(2)  interface
(3)      export      {část vývozu}
             type STR = public   {veřejná část typu STR}
(4)                       LEN:integer;   {délka řetězce}
                         private  {soukromá část typu STR}
(5)                       CONT:packed array [1..1ØØ] of char;
                                 {obsah textového řetězce}
                         end;
(6)          initial (var S:STR);  {otevření instance}
(7)          procedure CAT (var S:STR; CH:char);
               {Procedura přidá na konec textového řetězce S znak CH}
(8)          procedure WRIT (var S:STR; var F:text);
               {Procedura vytiskne obsah řetězce S do souboru F}
(9)      import       {část importu je prázdná}
     end; {rozhraní}
(1Ø) procedure CAT;
         begin with S do begin
                          LEN:=LEN+1;  {Překročení maximální délky se
                                        pro zjednodušení nehlídá}
                          CONT[LEN]:=CH;
                         end
         end;
(11) procedure WRIT;
         begin
             write (F, S.CONT:S.LEN);  {Zápis textového řetězce délky S.LEN do F}
         end
(12)     initial
         begin   S.LEN:=Ø
         end.
```

Multimodul STRING má definováno jméno (1), rozhraní (2) a operace (10), (11). Rozhraní obsahuje pouze část vývozu (3); dovoz (9) je prázdný. Multimodul vyváží datový typ STR, který má část veřejnou (4) a soukromou (5) a proto je tento datový typ abstraktní. Dále se vyváží tři operace: standardní operace otevření multimodulu (6). (Pozn. operace uzavření není zapotřebí) a veřejné operace CAT a WRIT (7), (8). Ve specifikacích, je vždy jako první parametr instance. Implementace operací (10), (11), (12) tvoří vlastní obsah modulu.

Modul (abstraktní datový typ) STRING, lze využít např. takto :

```
(1)   var S:STR;
          CH:char;
      :
      begin
(2)    {initial (S)}
      :
      read (CH);
(3)   CAT (S,CH);
      :
(4)   WRIT (S,OUTPUT);
(5)    {final (S);}
      end;
```

Deklarací proměnné S typu STR se vytvoří jedna instance multimodulu nazvaná S (1). Před prvním použitím se automaticky zařadí standardní operace otevření instance (2). Potom se s instancí S pracuje v operacích daného typu (3), (4). Závěrečná operace uzavření instance (5) se automaticky provede jako prázdný příkaz.

Dalším zobecněním pojmu modul je zavedení parametrizace modulu, která rozšíří vlastnosti modulárního programování. Uvedením skutečných parametrů lze modul přizpůsobit konkrétním potřebám užití. Jako parametr modulu lze použít nejen obvyklé hodnoty či proměnné specifikovaných typů, ale parametrem může být i typ dat, operace a jméno modulu. Takto zobecněný pojem modulu představuje předpis pro generování celé třídy modulů. Zaveďme proto následující rozšíření :

Generický programový modul resp. generický programový multimodul (dále jen generický modul resp. generický multimodul) je parametrizovaný modul resp. multimodul.

Jeho formálními parametry mohou být :

- hodnoty určitého typu, kterými lze modifikovat konstanty modulu jak v rozhraní, tak v implementaci operací
- proměnné určitého typu, kterými lze modifikovat skutečný dovoz proměnných
- operace s konkrétní specifikací, kterými lze modifikovat skutečný dovoz proměnných
- jména datových typů, kterými lze modifikovat dovoz datových typů nebo typy použitých proměnných
- jména modulů, kterými lze modifikovat, odkud se objekty do modulu dováží

Parametrizuje-li se modul proměnnou, operací nebo datovým typem, není nutné znát v době definice modulu modul, ze kterého se proměnná, operace nebo datový

typ dováží. Tento modul se určí až při vytvoření generovaného modulu.

Generace modulu resp. multimodulu se zadává uvedením skutečných parametrů generického modulu resp. multimodulu v oblasti dovozu uživatelského modulu. Tím se definuje konkrétní podoba modulu resp. multimodulu.

Příklad 4.3.

Vytvořme modul TABLE, který realizuje abstraktní datový typ "vyhledávací tabulka". Tabulka obsahuje určitý počet (až do zadané maximální hodnoty) položek obecně libovolného typu, pokud nad hodnotami tohoto typu existuje operace ekvivalence. Pro takto obecně formulované zadání je nezbytné využít generický multimodul.

```
(1)   module TABLE (MAX:Ø..MAXINT;  {maximální počet položek}
                    type ITEM;  {typ položky uchovávané v tabulce}
(2)                 operation EQU   {operace ekvivalence nad položkami}
                    );
      interface
          export
(3)          type TAB = private
                          TT:array [1..MAX] of ITEM;
                          IDX:Ø..MAX;
                       end;
(3)          initial (var T:TAB);   {otevření instance}
(3)          procedure ENTER (var T:TAB; var I:ITEM; var ENTERED:Boolean);
              {funkce vloží do tabulky T novou položku I, není-li již
               v tabulce ekvivalentní položka obsažena. Vloží-li se do
               tabulky nová položka, má parametr ENTERED hodnotu true, jinak
               má hodnotu false}
(3)          function SEARCH (var T:TAB; var I:ITEM): Boolean;
              {funkce zjistí, zda se v tabulce T nachází položka ekvivalentní
               s položkou I. Je-li tam taková položka, má funkce hodnotu true,
               jinak má hodnotu false}
          import
(4)          type ITEM;
(4)          function EQU (var X,Y:ITEM): Boolean;
                {ekvivalence operandů X a Y}
          end;  {rozhraní}
(5)       initial
             begin T.IDX:=Ø
             end;
(5)       procedure ENTER;
             begin
                   with T do
                     begin if SEARCH (T,I)
                              then ENTERED:=false
                              else begin
                                      IDX:=IDX+1;
                                      TT[IDX]:=I;
                                      ENTERED:=true
                                   end;
```

```
                end;
            end;  {funkce}
(5)         function SEARCH;
            var B:Boolean;
                M:1..MAX;
            begin
                B:=false;
                M:=1;
                while not B and (M ≤ T.IDX) do
                      if EQU (T.TT[M],I)
                         then
                                B:=true
                         else M:=M+1
                SEARCH:=B;
            end;  {funkce}
      end. {Modulu}
```

Generický multimodul TABLE (1) je parametrizován (2) třemi parametry: Hodnotou MAX, kterou se určuje maximální počet položek v tabulce, datovým typem ITEM, kterým se určuje typ položek v tabulce a operací EQU, která určuje operaci ekvivalence nad dvěmi položkami typu ITEM. V rozhraní modulu jsou pro výrazy (3) specifikovány abstraktní datový typ TAB a operacemi initial (otevření), ENTER a SEARCH. Dováží se datový typ ITEM a operace EQU (4). Protože jde o parametry generického multimodulu, není nutné uvádět jméno modulu, ze kterého se ITEM a EQU dováží. Zbývající část modulu tvoří implementace specifikovaných operací (5).

Modulu TABLE lze v jiném modulu využít např. takto :

```
import
      from module OBJECT;
           type OBJ;
           function OBJEQU (var X,Y:OBJ):Boolean;
(6)  6    from module TABLE (100, OBJ, OBJEQU);
           type TAB;
           :
           :
```

V rozhraní modulu, který bude využívat abstraktního typu dat "vyhledávací tabulka" se provede generace instance multimodulu (6) zadáním skutečných parametrů modulu.

4.2. P r o g r a m o v á n í v e v e l k é m

Výstavba rozsáhlých programovacích systémů přináší celou řadu kvalitativně nových problémů. Metody tvorby, které vyhovují u malých programů nejsou účinné u rozsáhlých programových systémů. Důležitým momentem se stává fakt, že na rozsáhlém programovém projektu se musí podílet více programátorů. A tak se do popředí dostává problematika dělby práce mezi jednotlivé programátory, případně mezi skupiny programátorů a zajištění časové i věcné návaznosti produktů jejich práce.

Pro odlišení této oblasti od programování v dosud obvyklém smyslu byly zavedeny pojmy programování ve velkém (angl. programming in the large) a programování v malém (angl. programming in the small).

Základním úkolem programování ve velkém je dekompozice projektu na jednotlivé části - moduly. Vycházíme-li z definice modulu, pak můžeme modul rozložit na dvě části :

- specifikace modulu, která určuje jméno a rozhraní modulu
- implementace modulu, která definuje všechny soukromé proměnné, implementaci veřejných operací a specifikaci i implementaci soukromých operací.

Výsledkem etapy programování ve velkém je specifikace všech modulů daného programového projektu. Úkolem následující etapy - programování v malém - je implementace těchto modulů ve shodě s jejich specifikací.

Mezi jednotlivými moduly projektu se na základě vztahů dovoz-vývoz vytváří relace nadřazenosti. Říkáme, že modul A je nadřazený modulu B, když modul A dováží z modulu B operace nebo proměnné. Reprezentujeme-li relaci nadřazenosti jako jednoduchý orientovaný graf, získáme graf nadřazenosti modulární struktury. Tomuto grafu se také říká graf viditelnosti. Na rozdíl od blokové struktury, v níž podřízený blok "vidí" objekty nadřazeného bloku, v modulární struktuře jsou viditelné objekty podřízeného modulu. Z hlediska vlastností modulární struktury je výhodné, aby relace nadřazenosti byla částečným uspořádáním, tzn. aby byl graf nadřazenosti hierarchický. *) Jednotlivé úrovně ("patra") hierarchické struktury se nazývají virtuální stroj. Platí zásada, že každá nadřazená úroveň - každý abstraktnější virtuální stroj - je implementován pouze prostředky nižšího virtuálního stroje. Za základní (nejnižší) virtuální stroj považujeme zvolený programovací jazyk "v malém" - např. Pascal.

Pro programování ve velkém existují speciální systémy. Specifikace modulů se vyjadřuje v jazyce programování ve velkém. Může to být speciální jazyk, odlišný od jazyků programování v malém, nebo mohou oba typy jazyků vytvářet jeden celek. Systémy programování ve velkém zajišťují tyto funkce :

1. Kontrola správnosti modulární struktury, která provádí kontrolu uspokojení každého dovozu vývozem a analýzu grafu nadřazenosti.
2. Kontroly při samostatném překladu implementací modulu v jazyce pro programování v malém. Tyto kontroly zjišťují shodu mezi definicí objektů a využíváním těchto objektů.
3. Automatická dokumentace struktury programového projektu.
4. Automatizace údržby programového systému. Zajišťuje kontroly při sestavování modulů, údržbu jednotlivých verzí modulů a generování magnetických pásek pro distribuci projektu mezi jeho uživatele.

---

*) Je možné požadovat i slabší podmínku - a to, aby graf byl částečně hierarchický. T.zn., že na jednotlivých úrovních hierarchického uspořádání mohou být libovolné vazby.

4.3. Modulární programování v jazyce PASCAL - EC

4.3.1. Popis rozšíření jazyka PASCAL - EC

Koncepce jazyka PASCAL se vytvářela na počátku sedmdesátých let, tedy dříve, než se objevily první práce v oblasti modulárního programování. Proto nejsou přímo v definici jazyka obsaženy žádné prostředky podporující tuto technologii vytváření programového vybavení počítačů. Hlavní omezení se týkají nevyřešené otázky samostatné kompilace procedur nebo funkcí a nutnosti uvádět deklarace v pevném pořadí (návěští, konstanty, ...) bez možnosti rozdělit jednotlivé typy deklarací na více částí. Většina překladačů jazyka PASCAL se snaží vyřešit tyto nedostatky. Řešení, které nabízí překladač PASCAL - EC, umožňuje využití metody modulárního programování i pro programovací jazyk PASCAL.

4.3.1.1. Vkládání úseků programů ze zdrojové knihovny

Překladač PASCAL - EC umožňuje vkládat při překladu do zdrojového textu programu úseky uložené před překladem do zdrojové knihovny operačního systému. Rozlišují se přitom dvě formy :

- příkaz COPY
- příkaz INCLUDE

Jejich syntaktická definice je následující :

```
příkaz-COPY = "¤COPY" jméno-knihy    [ parametry ] [ ";" ] .
příkaz-INCLUDE = "¤INCLUDE" jméno-knihy [ parametry ] [ ";" ] .
jméno-knihy = identifikátor .
parametry = "(" parametr { "," parametr } ")".
parametr = posloupnost-znaků.
příkaz-vložení-textu = příkaz-COPY | příkaz-INCLUDE .
```

Pro obě formy příkazu-vložení-textu platí tato společná pravidla :

1. Příkaz-vložení-textu způsobí nahrazení znaků příkazu (nikoliv celého vstupního řádku obsahem knihy ze zdrojové knihovny zadaného jména-knihy za předpokladu, že se ve zdrojové knihovně nachází kniha udaného jména. Jinak je signalizovaná chyba.
2. Délka jména-knihy, která má vliv na hledání v knihovně je maximálně 8 znaků. Zbývající znaky jména-knihy se ignorují. Pro konstrukci jména-knihy platí stejná pravidla jako pro každý jiný identifikátor jazyka PASCAL - EC.
3. Text vkládané knihy může opět obsahovat příkazy vložení textu. Maximální úroveň zanoření je 32. Úroveň zanoření a odpovídající jméno-knihy je tisknuto v protokolu o překladu ve sloupcích N a SOURCE. Úroveň hlavního textu má N=0.
4. Každý parametr se skládá z posloupnosti libovolných znaků. Všechny znaky (včetně mezery) jsou prvky parametru. Obsahuje-li parametr čárky, musí být uzavřeny mezi dvojicí shodných závorek. Za dvojici závorek se považují :
   ( ),[ ], '', (**), (..), { }
5. Parametr může obsahovat příkaz-vložení-textu, který se ale neuplatní v okamžiku analýzy parametru, ale teprve při modifikaci textu (viz dále).
6. Parametry slouží k modifikaci vkládaného textu. Požadujeme-li modifikaci, je nutné, aby jako první příkaz vkládané knihy byl příkaz-MACRO.

```
příkaz-MACRO = "¤MACRO" "(" formální-parametr { "," formální-
                    parametr } ")" [ ";" ] .
formální-parametr = "& identifikátor".
```

7. Počet formálních-parametrů příkazu-MACRO musí být shodný s počtem parametrů příkazu vložení textu. Každému formálnímu-parametru je takto přiřazena určitá textová hodnota parametru podle odpovídajícího pořadí.
8. Text vkládané knihy může obsahovat libovolný počet výskytů formálních parametrů. Každý takto použitý formální-parametr musí být uveden v příkazu-MACRO. Při vkládání takovéto knihy budou na všechna místa formálních-parametrů dosazeny jejich textové hodnoty z příkazu-vložení-textu. Dosazený text už není znovu prohlížen pro hledání formálních-parametrů.
9. Má-li za formálním-parametrem bezprostředně následovat písmeno nebo číslice, musí být mezi ně vložena tečka. Má-li následovat tečka, musíme vložit dvě tečky.

Pro ilustraci těchto vlastností předpokládejme, že ve zdrojové knihovně je pod jménem KNIHA1 uložen následující text:

```
¤MACRO (&TYPE,&MIN,&P,&R);
&TYPE;
   RG = &MIN...256;
  &P.x = array&R of char;
```

a pod jménem KNIHA2 text :

```
  type
```

Potom úsek překladu části programu může vypadat takto :

```
LINE TEXT                                                    N SOURCE
 1 ¤COPY KNIHA1 (¤COPY KNIHA2,1,MM,[RG,RG]);                 0
 2 ¤COPY KNIHA2;                                             1 KNIHA1
 3  type                                                     2 KNIHA2
 4    RG = 1..256;                                           1 KNIHA1
 5    MMx = array [RG,RG] of char;                           1 KNIHA1
```

Řádky 1 a 2 se pouze protokolují, ale nepřekládají se.

Mimo výše uvedené společné vlastnosti příkazů-vložení-textů existují i tyto rozdíly :

1. Příkaz-COPY je možné uvést na libovolném místě v programu a chápe se jako prostředek pro textovou úpravu programu. Naproti tomu lze použít příkaz-INCLUDE pouze jako rozšíření úseku deklarací bloku. Rozšířená definice je

```
úsek-deklarací = úsek-deklarací-návěští
                 úsek-deklarací-konstant
                  úsek-deklarací-proměnných
                   úsek-deklarací-procedur-a-funkcí
                    příkazy-INCLUDE.
příkazy-INCLUDE = { příkaz-INCLUDE } .
blok = úsek-deklarací příkazová-část .
```

   Text vložený každým příkazem-INCLUDE musí mít strukturu úseku-deklarací. V rámci knihy vkládané příkazem-INCLUDE musí být rozřešeny všechny definice typů ukazatel. Takto umožní příkaz-INCLUDE vložit do programu samostatný úsek deklarací bez ohledu na porušení pevného pořadí jednotlivých úseků deklarací.
2. Mezi navzájem zanořenými knihami vloženými příkazy-COPY se nesmí vyskytovat dvě stejného jména. Naproti tomu, vyskytnou-li se ve stejném nebo nadřazeném úseku-deklarací dva stejné příkazy-INCLUDE, je druhý interpretován

jako prázdný příkaz. Dva příkazy-INCLUDE jsou stejné tehdy, když se shodují ve jméně-knihy a v hodnotě všech parametrů.

4.3.1.2. Samostatné kompilace modulu

Požadavek samostatně kompilovat určitou část výsledného programového systému se zejména při větším rozsahu systému stává nezbytností. Zkracují se tím výrazně doby kompilací, zjednodušuje se manipulace při opravách a na jednom programu se může podílet bez problémů i více programátorů. Překladač PASCAL - EC umožňuje samostatně překládat skupinu procedur a funkcí. Takto samostatně překládanou proceduru nebo funkci budeme nazývat vnější operací. Překladač při jednom svém běhu může buďto přeložit jeden program (tak jak je obvyklé) nebo jeden fyzický modul. Syntaktická skladba obou překládaných jednotek je shodná. O fyzickém modulu hovoříme tehdy, když definuje aspoň jednu vnější operaci. Spustitelný celek tvoří právě jeden program, který může být spojený (spojovacím programem operačního systému) s libovolným počtem modulů.

Na úrovni programu nebo fyzického modulu lze využívat vnějších operací určitého fyzického modulu za těchto předpokladů :

1. Pro každou vnější operaci fyzického modulu bude uvedena specifikace vnější operace ve tvaru hlavičky procedury či funkce doplněné direktivou OPER.
2. Pro každý fyzický modul, jehož vnější operace se využívají, je zapotřebí definovat užití fyzického modulu ve tvaru

   procedure jméno-modulu; MODULE;

   mezi deklaracemi procedur a funkcí toho bloku, kde budou vnější operace využívány.

Při definici vnější operace je třeba použít postup podobný použití direktivy FORWARD. Nejprve se uvede specifikace vnější operace stejným způsobem jako na straně použití. Vlastní realizace následuje dále v textu stejným způsobem, jako by se jednalo o direktivu FORWARD (tj. formální parametry se neopakují). Podle toho, zda překladač nalezne definici těla vnější operace se rozhoduje, jde-li o specifikaci nebo o definici. Schematicky lze tuto situaci znázornit takto :

```
(1) program P; { program }          (2) program M; { fyzický modul }
      :                                   :
(3) procedure OP(N: integer);       (3) procedure OP(N: integer);
      OPER; { specifikace operace OP }        OPER; { deklarace operace OP }
      :                                   :
(4) procedure M; MODULE;            (5) procedure OP; { Tělo operace OP }
      :                                   :
    begin                           (6) begin ... end;
      :                                 begin
    end.                                  :
                                        end.
```

Samostatně se překládají dvě jednotky: program P (1) a fyzický modul M (2). Na straně použití se uvede specifikace vnější operace OP (3) podobně jako při její definici. Na straně definice je ale doplněno její tělo (5) (6). Užití fyzického modulu ze strany programu P je signalizováno v (4). Tato signalizace se

uplatní při sestavování programu a jeho modulů (vazba typu EXTRN). Pouhá vazba přes jméno vnější operace (vazba typu WXTRN) nedostačuje a všechny vnější operace, které při spojování se neuspokojily (tj. chybí jejich definice) se interpretují jako prázdné operace.

Pro používání této koncepce samostatné kompilace platí následující pravidla:

1. Procedura s direktivou MODULE se může vyskytnout na libovolné úrovni blokové struktury uvnitř jak programu, tak i fyzického modulu a způsobí vygenerování dvou implicitních příkazů. Příkaz inicializace modulu se provede vždy před prvním uživatelským příkazem daného bloku a příkaz finalizace se provede po jeho posledním příkazu. Oba příkazy mohou obsahovat mimo systémových akcí také uživatelské příkazy. Pro příkaz inicializace jsou umístěny uvnitř těla hlavního bloku příslušného fyzického modulu. Požadujeme-li uživatelské příkazy při finalizaci fyzického modulu je zapotřebí rozdělit tělo hlavního bloku fyzického modulu na dvě části

```
begin
  [příkazy inicializace]
[final
  [příkazy finalizace] ]
end.
```

Hranaté závorky označují části, které mohou být při definici těla hlavního bloku fyzického modulu vynechány.

Příkaz inicializace zahrnuje tyto akce (v uvedeném pořadí) :

a) vytvoření bloku dat na vrcholu zásobníku s proměnnými fyzického modulu deklarovanými na hlavní úrovni
b) vytvoření řídících bloků všech souborů deklarovaných na hlavní úrovni
c) implicitní příkazy inicializace všech užívaných fyzických modulů
d) uživatelské příkazy inicializace

Příkaz finalizace provádí postupně tyto akce :

a) uživatelské příkazy finalizace
b) implicitní příkazy finalizace všech užívaných fyzických modulů
c) uzavření všech souborů deklarovaných na hlavní úrovni
d) zrušení bloku dat modulu (proměnné + řídící bloky souborů)

2. Je-li jméno fyzického modulu definované při užití shodné se jménem fyzického modulu v němž je použito, je toto užití ignorováno.

3. Dojde-li k příkazu inicializace nebo finalizace nad fyzickým modulem, který už je inicializován (resp. finalizován) případně inicializace (resp. finalizace) probíhá, je proveden jako prázdný příkaz.

4. Proměnné deklarované na hlavní úrovni fyzického modulu jsou přístupné v době mezi příkazem inicializace a finalizace tohoto fyzického modulu, tj. v době, kdy je možné využívat jeho vnější operace. Každá z těchto proměnných může být definovaná jako externí nezávisle na ostatních modulech. V případě, že se jedná o soubory, musí si jejich vícenásobnou využitelnost zajistit uživatel. Výjimku tvoří standardní soubor output, který je sdílen mezi všemi fyzickými moduly i programem.

5. Přístup k proměnným jiného fyzického modulu (nikoliv programu) lze zajistit tehdy, definujeme-li užití fyzického modulu takto :

function jméno-modulu:@TYP; MODULE; [*]

kde TYP popisuje proměnné jméno-modulu počínaje první proměnnou. V tomto případě musí uživatel explicitně zavolat danou funkci, která mu vrátí ukazatel na proměnné požadovaného fyzického modulu. Použití této funkce a její hodnoty musí být v úseku kódu bez prověřování správnosti typu ukazatel (¤¤T-¤), neboť tyto údaje neleží v dynamické paměti, ale uvnitř zásobníku.

4.3.1.3. Standardní datový typ ANY

Silná kontrola typové čistoty je jedním ze základních rysů jazyka PASCAL. Přestože jde o progresivní rys, dochází zejména při používání parametrických uživatelských datových typů ke komplikacím. Např. modul organizující seznam dynamických objektů manipuluje pouze s adresami objektů, nikoliv s objekty samotnými. Kontrola typové čistoty pak způsobuje nižší obecnost modulu, nebo vede ke generování více kopií téhož modulu.

Tyto nevýhody řeší použití standardního datového typu ANY, pro který platí tyto pravidla :

1. není přípustné definovat proměnnou či složku strukturované proměnné ani parametrizovat jiný typ (mimo typu ukazatel) nebo proceduru či funkci hodnotou pomocí ANY. Tj. lze používat pouze @ANY resp. parametr proměnná typu ANY.
2. s údaji typu ANY nelze manipulovat nijak jinak, než formou předávání si jejich adres. Tj. nelze je dosazovat, porovnávat ani dynamicky vytvářet (new). Je možné je předat jako parametr proměnné. S údaji typu @ANY je možné manipulovat libovolně.

4.3.2. Modulární programování

V jazyce PASCAL - EC je nutné realizovat každý modul, resp. multimodul jedním fyzickým modulem uloženým ve dvou samostatných knihách v knihovně operačního systému. Jedna kniha bude obsahovat definici rozhraní modulu, tj. jeho specifikaci a druhá definici implementace modulu. Protože je zapotřebí uchovávat obě knihy v jediné knihovně (resp. podknihovně), musí se pojmenovat různě. Přijmeme konvenci, že kniha obsahující definici rozhraní se bude jmenovat tak, jako modul a kniha s implementací modulu se bude jmenovat jméno-moduluM. Obecná struktura implementace modulu je :

```
program jméno-moduluM (seznam-externích-objektů);
¤INCLUDE jméno-modulu1;  }
    :                    }      dovoz modulu
¤INCLUDE jméno-modulun;  }
¤INCLUDE jméno-modulu;   }      vývoz modulu
lokální-deklarace-modulu
implementace-vyvážených-operací
begin
   otevření-modulu
final
   uzavření-modulu
end.
```

---

[*] Znak " @ " je v Pascalu synonymem znaku " ↑ " a zde se používá pro zápis typu ukazatel.

Každá kniha s definicí rozhraní obsahuje deklarace vyvážených objektů. Pokud je nutné při konstrukci deklarace vyvážených objektů použít některý z dovážených objektů (jedná se zpravidla o datový typ) je nutné uvést i zde definici dovozu. Všechny operace jsou uvedeny s direktivou OPER a je doplněno užití fyzického modulu:

```
¤INCLUDE jméno-modulu1;   }
    :                     }   specifikace částí dovozu
¤INCLUDE jméno-modulun;   }   modulu
const
   definice-vyvážených konstant
type
   definice-vyvážených typů
procedure OPERACE1 ...; OPER;
    :
procedure OPERACEm ...; OPER;
procedure jméno-moduluM; MODULE;
```

Při vývozu proměnných se rozšíří definice rozhraní takto

```
type
    jméno-moduluT = record
                      definice-vyvážených-proměnných
                    end;
```

a nahradí se užití fyzického modulu

```
function jméno-moduluM:@jméno-moduluT; MODULE;
```

Při otevření modulu, který danou definici rozhraní dováží, je nutné definovat proměnnou

```
var
    jméno-moduluV:@jméno-moduluT;
```

a mezi příkazy otevření modulu uvést

```
jméno-moduluV:=jméno-moduluM;
```

Příklad 4.4.

Předpokládejme zadání modulu LISTING z příkladu 1. Pro zápis v jazyce PASCAL - - EC je zapotřebí založit dvě knihy s tímto obsahem : kniha LISTING - definice rozhraní modulu

```
type LISTINGT = record
                  PAGENUM: integer;
                end;
procedure LINE; OPER;
function LISTINGM:@LISTINGT; MODULE;
```

kniha LISTINGM - definice implementace modulu

```
program LISTING(output);
¤INCLUDE TITLE;
¤INCLUDE LISTING;
var PUBLIC: LISTINGT;
    LINENUM: integer;
procedure LINE;
begin
    LINENUM := LINENUM+1;
    if LINENUM = 55 then begin
```

```
            LINENUM := Ø;
            PUBLIC.PAGENUM := PUBLIC.PAGENUM+1;
            HEADER;
        end;
    end;
    begin  {inicializace}
        LINENUM := Ø;
        PUBLIC.PAGENUM := 1;
        HEADER;
    final
        writeln ('END OF REPORT');
    end.
```

Na straně uživatele tohoto modulu bude použití vypadat

```
    program XX ... ;
       :
    ¤INCLUDE LISTING;
       :
    var LISTINGV:@LISTINGT;
       :
    begin
        (*¤T-*)[*]  {úsek programu bez kontroly správnosti typu ukazatel}
        LISTINGV := LISTINGM;
        with LISTINGV@do begin
           :
           LINE;
           if PAGENUM > 1Ø then
           :
        end;
    end.
```

V případě multimodulu se místo s proměnnými pracuje s datovým typem modulu. Jeho obecný tvar - část veřejná a soukromá se realizuje

```
    record
        veřejná-část
        PRIVATE: record
                    soukromá-část
                 end;
    end;
```

Tím se výrazně ztíží přístup k soukromým proměnným. Není to sice dokonalé řešení, ale pro disciplinované programátory je postačující.

Příklad 4.5.

Předpokládejme zadání modulu STRING z příkladu 2. Pro zápis v jazyce PASCAL-EC je zapotřebí založit dvě knihy s tímto obsahem :
kniha STRING - definice rozhraní modulu

---

[*] Řetězec '(*' je synonymum znaku '{' a řetězec '*)' je synonymum znaku '}' .

```
type STR = record
              LEN: integer;
              PRIVATE: record
                         CON: packed array [1..100] of char;
                       end;
           end;

    procedure STRINIT (var S:STR); OPER;
    procedure CAT (var S: STR; CH: char); OPER;
    procedure WRIT (var S: STR; var F: text); OPER;
    procedure STRINGM; MODULE;
```

kniha STRINGM - implementace modulu

```
program STRINGM;
¤INCLUDE STRING;
procedure CAT;
begin
    with S,PRIVATE do begin
       LEN:=LEN+1;
       CON[LEN]:=CH;
    end;
end;
procedure WRIT;
begin
   with S,PRIVATE do
     write (F,CON:LEN);
end;
procedure STRINIT;
begin
    S.LEN:=0;
end;
begin
end.
```

Takto realizovaná definice modulů vyhovuje obecně definovaným vlastnostem modularity. Překladač nezajišťuje pouze kontrolu, zda se do dovážených proměnných nedosazuje a negeneruje automaticky operace otevření a uzavření multimodulu. Poznamenejme, že je možné realizovat i obecnější schéma modularity, kdy se nemusí explicitně definovat ze kterých modulů se dováží a je možné, aby měl multimodul proměnné společné všem instancím modulu.

V případě, že potřebujeme definovat nejobecnější formu modulu - generický modul, musíme kromě dvou definičních knih, které se pak samostatně nepřekládají, založit pro každou vygenerovanou variantu generického modulu jednu knihu obsahující pouze

¤COPY jméno-moduluM (parametry);

Mezi parametry bude udané i jméno vygenerované varianty. Stejné parametry je nutné zadat i při definici dovozu.

Příklad 4.6.

Předpokládejme zadání modulu TABLE z příkladu 4.3. Pro zápis v jazyce PASCAL-EC je zapotřebí založit knihy :

```
      ¤MACRO(& VARIANTA, & MAX, & ITEM, & EQU, & IMPORT);
      ¤INCLUDE & IMPORT;
      type TAB = record
                    PRIVATE: record
                                TT: array [1..&MAX] of &ITEM;
                                IDX: ∅.. &MAX;
                             end;
                 end;
      procedure TABINIT (var T: TAB); OPER;
      procedure ENTER (var T: TAB; var I: &ITEM; var ENTERED: Boolean); OPER;
      function SEARCH (var T: TAB; var I: &ITEM):Boolean; OPER;
      procedure &VARIANTA.M; MODULE;
kniha TABLEM - implementace generického modulu
      ¤MACRO (& VARIANTA, & MAX, & ITEM, & EQU, & IMPORT);
      program &VARIANTA.M;
      ¤INCLUDE TABLE (& VARIANTA, & MAX, & ITEM, & EQU, & IMPORT);
      procedure TABINIT;
      begin
          T.PRIVATE.IDX:=∅;
      end;
      function ENTER;
      begin
          with T,PRIVATE do begin
             if SEARCH(T,I) then
                 ENTERED:=false
             else begin
                 IDX:=IDX+1;
                 TT[IDX]:=I;
                 ENTERED:=true;
             end;
          end;
      end;
      function SEARCH;
      var B: Boolean;
          M: 1..&MAX;
      begin
          B:=false;
          M:=1;
          with T,PRIVATE do begin
            while not B and (M<=IDX) do
               if &EQU(TT[M],I) then begin
                  I:=TT[M];
                  B:=true;
               end else
                  M:=M+1;
            end;
          SEARCH:=B;
      end;
      begin
      end.
```

Při použití tohoto modulu musíme nejdříve zvolit konkrétní variantu a vygenerovat ji; tj. přeložit knihu TABLE1

```
¤COPY TABLEM(TABLE1,1ØØ,OBJ,OBJEQU,OBJECT);
```

Ne straně uživatele této varianty modulu pak bude :

```
   ⋮
¤INCLUDE TABLE(TABLE1,1ØØ,OBJ,OBJEQU,OBJECT);
   ⋮
var T: TAB;
    O: OBJ;
    VLOZIL:Boolean;
begin
    TABINIT(T);
       ⋮
       ENTER(T,O,VLOZIL);
         if VLOZIL then ...
end.
```

Příklad použití :

Vytvořme tabulku záznamů o osobách, v níž klíčem bude jméno osoby. Ekvivalentní jsou dva záznamy se shodným klíčem.

Kniha OSOBA definuje rozhraní :

```
type TYPOSOBA = record
                JMENO:packed array [1..2Ø] of char;
                DATA:record
                       EVIDCISL: integer;
                       ROKNAR: integer
                     end;
                end;
function EQUOSOBA (OSOB1,OSOB2:TYPOSOBA):BOOLEAN; OPER;
procedure OSOBA; MODULE;
```

Kniha OSOBAM definuje implementaci :

```
program OSOBAM;
¤INCLUDE OSOBA;
function EQUOSOBA;
  begin
      EQUOSOBA:=OSOB1.JMENO=OSOB2.JMENO
  end;
begin
end.
```

Pak lze v uživatelském programu :

```
   ⋮
¤INCLUDE TABLE (TABLE1,1ØØ,TYPOSOBA,EQUOSOBA,OSOBA);
   ⋮
var T:TAB;
    O:TYPOSOBA;
begin
     TABINIT(T)
```

Pozn.: Aby bylo možné použít v jednom uživatelském modulu dvě varianty tabulky s různými typy např.

```
(TABLE1, 1ØØ, OBJ1, OBJEQU1, OBJECT1)
a    (TABLE2, 5Ø, OBJ2, OBJEQU2, OBJECT2)
```

musel by parametrem makroinstrukce být i typ &TYP.

4.4. Literatura

[1] Ghezzi,C., Jazayeri,M.: Programming Language concepts, New York, John Wiley & Sons 1982.

[2] Rajlich,V. a kol.: Datové abstrakce, Sborník přednášek konference SOFSEM, Bratislava 1977

[3] DeRemer,F., Kron,H.: Programming in the Large Versus Programming in the Small, IEEE Transactions on Software Engineering, June 1976, str. 80-86.