2. VLASTNOSTI ZÁKLADNÍCH DATOVÝCH STRUKTUR A JEJICH VYUŽITÍ

2.1. K o n c e p c e d a t o v ý c h t y p ů

2.1.1. Úvod

Moderní číslicový počítač je vyvinut a používá se jako zařízení, které má umožnit či zrychlit složité a časově náročné výpočty. Ve většině aplikací hraje schopnost počítače uchovat a posléze umožnit přístup k velkému objemu informace nejdůležitější roli a považuje se za primární vlastnost počítače. To, že počítač může "počítat", t.zn. provádět aritmetické operace, se většinou chápe jako méně významná a někdy dokonce zanedbatelná okolnost.

Ve všech těchto případech representuje velké množství informace, které se má zpracovat, abstrakci určité části reálného světa. Tato informace sestává z vybrané množiny dat o reálném světě. Z takové množiny, která se vztahuje k řešenému problému a která se považuje za postačující pro dosažení požadovaných výsledků. Data representují abstrakci reality v tom smyslu, že neobsahují určité vlastnosti a údaje o reálných objektech, které lze považovat za okrajové nebo podružné. Abstrakce je tedy významným mechanismem zjednodušujícím skutečnost.

Ať už řešíme problém na počítači nebo bez něj, vždy musíme zvolit abstrakci reality, t.zn. definovat množinu dat, která representují skutečnou situaci. Tato volba závisí na řešeném problému. Potom následuje volba representace této informace. Tato volba závisí na prostředcích, kterými se problém řeší tzn. na vlastnostech počítače, který je k dispozici.

Volba representace dat je často obtížná a není vždy jednoznačně určena vlastnostmi počítače. V úvahu se musí vždy vzít dané operace, které se nad daty budou provádět. Dobrým příkladem je representace čísel, které samy o sobě jsou abstrakcí vlastností určitých objektů. Je-li sčítání jedinou (nebo alespoň nejdůležitější a nejčastější) operací, která se nad čísly bude provádět, pak dobrým způsobem, jak representovat číslo n je zapsat n čárek. Sčítací pravidlo pro takovou representaci je zcela zřejmé a jednoduché. Čísla v římské soustavě jsou založena na stejném principu jednoduchosti a pravidlo pro sčítání malých římských čísel je shodné. Na druhé straně representace čísel v arabské soustavě vyžaduje pravidla, která (pro malá čísla) nejsou zřejmá a musí se zapamatovat. Situace se ale změní, jakmile začneme uvažovat o sčítání velkých čísel, nebo o násobení a dělení. Dekompozice těchto operací na jednodušší operace je mnohem jednodušší při práci s arabskými čísly a to v důsledku systematického strukturovacího principu založeného na posiční váze číslic.

Ukažme si na malém příkladu několik úrovní representace údaje. Je dán problém representovat polohu objektu ve dvojrozměrném prostoru. První úroveň rozhodování nás může vést ke dvojici reálných čísel buď v kartézské nebo polární soustavě. Druhá úroveň může vést k zobrazení v pohyblivé řádové čárce, kde každé číslo sestává ze dvou celých čísel representujících mantisu a exponent. Třetí úroveň, daná skutečností, že pro výpočet použijeme počítač, určí, že číslo bude representováno pomocí dvojkové soustavy. Čtvrtá úroveň určí fyzikální princip representace dvojkových číslic 0 a 1. Zatímco první úroveň je ovlivněna především vlastním problémem, poslední úroveň je určována použitým nástrojem a jeho fyzikálními vlastnostmi. Nelze požadovat, aby programátor řešil otázku representace čísla v paměti, či dokonce otázky související s principem uchování informace

v paměti počítače. Tyto otázky "na nižší úrovni" řeší konstruktéři počítačů, kteří jsou nejlépe zasvěceni do současné technologie a kteří určí nejpřijatelnější volbu pro aplikace, v nichž se mají uchovat a zpracovat čísla.

Z tohoto příkladu jasně vyplývá význam programovacích jazyků. Programovací jazyk představuje abstraktní počítač, který je schopen "rozumět" termínům tohoto jazyka a který může obsahovat určitou úroveň abstrakce objektů užitých v reálném počítači. Programátor, který užívá "vyššího" jazyka je např. zbaven starosti o representaci čísel, je-li číslo základním objektem tohoto jazyka.

Význam užívání jazyků, které poskytují vhodný soubor základních abstrakcí, společných většině problémů zpracování dat, tkví především ve spolehlivosti výsledných programů. Je snazší navrhnout program sestavený z čísel zapsaných ve známé notaci, ze souborů, posloupností a cyklů než sestavený z "bitů", slov a ekoků. Je samozřejmé, že konkrétní representace programu a jeho částí bude opět vytvářet velké množství bitů. Tato skutečnost však programátora nemusí zajímat, pokud nechce detailně sledovat, zda volba representace jím vybrané abstrakce, kterou provedl počítač (kompilátor) je vhodná pro dané zadání.

2.1.2. Vlastnosti datových typů

Jedním ze základních stavebních kamenů programovacího jazyka je koncepce jeho datových typů. I v matematice bývá zvykem důsledně odlišovat jednotlivé proměnné podle svého typu, např. reálné, komplexní, logické proměnné aj. Vyspělé programovací jazyky dodržují princip, že <u>každá konstanta, proměnná, výraz nebo funkce je určitého typu.</u> Tento typ určuje množinu hodnot, do kterých patří jeho konstanta, kterých může nabýt jeho proměnná či výraz, nebo které může generovat funkce. Vyspělé programovací jazyky dodržují zásadu, že typ datového objektu či funkce se <u>explicitně</u> uvede v deklaraci konstanty, proměnné nebo funkce. Deklarace typu objektu textově předchází jeho použití. Tato skutečnost má praktický význam uvážíme-li, že kompilátor musí zvolit typ representace datového objektu v paměti. Kapacita paměti přidělená datovému objektu by měla být určena podle rozsahu hodnot, jichž může údaj nabýt. Je-li tato informace známa kompilátoru, lze se vyhnout tzv. dynamickému přidělování paměti, což často vede k účinné realizaci algoritmů.

Základní vlastnosti koncepce datových typů v jazyce PASCAL, který bude v celém učebním textu používán jako základní programovací jazyk, se dají shrnout do těchto bodů :

1. Datový typ určuje množinu hodnot do níž náleží konstanta tohoto typu, které mohou být přiřazeny proměnným tohoto typu, kterých může nabýt výraz s použitím operátoru tohoto typu nebo kterých může nabýt funkce tohoto typu.
2. Typ hodnoty, kterou má konstanta, proměnná, výraz či funkce, lze určit z deklarace bez nutnosti provedení výpočtu.
3. Každý operátor nebo funkce předpokládá operandy (argumenty) stanoveného typu a dává výsledek stanoveného typu. Jestliže operand připouští operandy několika typů (např. + použité pro sčítání jak čísel <u>real</u> tak <u>integer</u>), pak se typ výsledku určí podle dalších pravidel jazyka.

Tato pravidla může kompilátor použít ke kontrole kompatibility a legálnosti různých konstrukcí. Přiřazení hodnoty typu <u>Boolean</u> proměnné typu <u>real</u> lze zjistit

již v průběhu překladu, bez nutnosti výpočtu (běhu) programu. Tento druh redundance v textu programu je svrchovaně užitečný jako nástroj pro tvorbu programů a považuje se za nejvýznamnější přednost vyšších jazyků nad strojově orientovanými jazyky.

Při studiu programovacího jazyka PASCAL jsme se seznámili s určitými metodami definování datových typů. Ve většině případů se nový datový typ definuje pomocí dříve definovaných datových typů. Hodnoty takových typů jsou obvykle složeny z hodnot komponent dříve definovaných ustavujících (konstitučních) typů a proto těmto složeným hodnotám říkáme strukturované. Jsou-li všechny hodnoty téhož konstitučního typu, pak tomuto typu říkáme bázový typ. Počet různých hodnot příslušících typu T se nazývá kardinalita typu T. Kardinalita je mírou pro určení množství paměti potřebné k representaci proměnné x typu T.

Konstituční typy mohou být opět samy strukturovány a tak lze vytvářet hierarchicky uspořádané struktury. Komponenty na nejnižší úrovni však jsou již "atomické" - dále nedělitelné. Proto je nutné, aby byly zavedeny také primitivní (jednoduché), nestrukturované typy. Hodnoty takových typů se ustavují vyjmenováním (výčtem,enumerací). Mezi primitivními typy jsou tzv. standardní typy, které jsou předdefinovány. V PASCALU zahrnují čísla, logické hodnoty a znaky. Jestliže jsou hodnoty daného primitivního typu uspořádány, pak se typu říká skalární typ. Jestliže ke každé hodnotě skalárního typu je definována hodnota následníka (s výjimkou poslední hodnoty) a hodnotu předchůdce (s výjimkou první hodnoty) pak se typ nazývá ordinální. V PASCALU jsou všechny nestrukturované typy skalární. V případě deklarace vyjmenováním jsou jejich hodnoty seřazeny podle pořadí v posloupnosti vyjmenování.

S těmito nástroji lze definovat primitivní typy a vytvářet složité datové struktury do libovolné vnořené úrovně. Při praktickém programování však nestačí jediná universální metoda pro vytváření struktur. S ohledem na praktické problémy representace a využití je nutné, aby vyšší programovací jazyky měly několik metod strukturování. Z matematického hlediska mohou být všechny ekvivalentní, ale liší se operátory, jimiž se konstruují hodnoty struktur a operátory, jimiž se vybírají komponenty ze strukturované hodnoty. Mezi základní metody strukturování patří pole, záznam, množina a soubor.

Pro práci s proměnnými jednotlivých typů jsou zavedeny operátory, pomocí nichž programem sestavujeme předpis pro výpočet. Podobně jako u datových typů jsou i zde k dispozici primitivní (jednoduché), standardní (atomické) operátory a řada strukturovacích metod, pomocí nichž se dají vytvářet složené operace z jednodušších (komponentních) operací. Zatímco tradiční způsob programování kladl důraz především na kompozici struktury operací, my budeme neustále sledovat, že neméně důležitou je i kompozice struktury dat.

Nejdůležitějšími základními operátory je srovnání (ekvivalence) a přiřazení. (Ačkoliv to není vůbec šťastné, některé vyšší jazyky, jako FORTRAN a PL/I používají rovnítko jako operátoru pro přiřazení.) Operace přiřazení je definována nad všemi datovými typy; musíme si však uvědomit, že pro rozsáhlé a hluboce strukturované datové typy může taková operace představovat značné množství strojových instrukcí. Operace srovnání je definována nad primitivními typy, množinami a textovými řetězci (viz [2]).

Další třídou základních a implicitně definovaných operátorů jsou tzv. transformační operátory, které mapují jeden datový typ do druhého. Jsou zvláště

důležité při práci se strukturovanými typy. Strukturované hodnoty se vytvářejí ze svých komponent pomocí tzv. konstruktorů a naopak, hodnotu jedné komponenty lze ze strukturované hodnoty získat pomocí tzv. selektoru. Konstruktory a selektory jsou tedy transformační operátory, které provádějí vzájemné mapování mezi ustavujícími typy a strukturovaným typem. Každá strukturovací metoda má svůj vlastní konstruktor a selektor, kterými se liší od jiných strukturovacích metod.

Standardní datové typy vyžadují také standardní operátory. A tak jsou zavedeny obvyklé aritmetické operace nad čísly, logické operace nad logickými hodnotami atd.

Zkoumáme-li vlastnosti datové struktury, mezi prvními nás zajímá, zda je struktura homogenní či heterogenní a zda je statická či dynamická. Zvláště druhá vlastnost bude velmi často aktuální.

Jsou-li všechny komponenty dané struktury téhož typu, je struktura homogenní, v jiném případě je heterogenní. Homogenní je i tehdy, jsou-li komponenty struktury samy o sobě nehomogenní, za předpokladu, že jsou všechny téhož typu. Např. pole záznamů je vždy struktura homogenní, zatímco záznam může být heterogenní.

Jestliže ustavená datová struktura nemůže měnit v průběhu výpočtu počet svých komponent ani způsob uspořádání struktury říkáme, že struktura je statická. V jiném případě hovoříme o dynamické struktuře. (Schopnost měnit svou hodnotu v průběhu zpracování nepřipisujeme dynamičnosti datové struktury. Je to základní vlastnost proměnných!) Např. pole je v jazyce Pascal statické. V Algolu 60 však mohlo v rámci blokové struktury měnit rozsah jednotlivých dimenzí (nikoliv však jejich počet!) a tudíž i počet svých komponent a proto je pole v Algolu 60 dynamickou strukturou. Datové struktury vytvořené v Pascalu pomocí ukazatelů mohou měnit nejen počet svých komponent, ale také způsob uspořádání struktury. Binární strom lze transformovat např. na oboustranně vázaný lineární seznam a naopak. Proto je taková struktura dynamická.

Vždy si však musíme uvědomit, že jak homogennost/heterogennost, tak statičnost/dynamičnost se posuzuje vždy jako vlastnost na jisté úrovni abstrakce. Snížíme-li se na úroveň stroje, pak jsou data vždy homogenní a statická.

2.2. Jednoduché datové typy

2.2.1. Typ definovaný výčtem

Tradiční způsoby zpracování údajů často používají celých čísel jako representace hodnot, které nemají číselný význam. S těmito celými čísly se také neprovádějí žádné aritmetické operace. Typickým příkladem je representace měsíce v roce pořadovým číslem, která je běžná i v každodenním životě. Závažnější je příklad, v němž pro hodnoty typu "stav osoby" (svobodný, ženatý, rozvedený, ovdovělý) zavedeme celá čísla 0, 1, 2, 3. V praxi pak snadno může dojít k chybám, zejména dochází-li ke změnám v kódovacím číselníku.

To jsou typické případy, pro které je v jazyce Pascal k dispozici typ definovaný výčtem. S tímto typem jsme se podrobně seznámili v předmětu Počítače a programování v [2], a proto shrňme jen některé nejdůležitější vlastnosti a zásady :

1. Typ definovaný výčtem je jednoduchý, ordinární typ. Jeho definice má tvar :

$$\underline{\text{type}}\ T = (C_1, C_2, \ldots, C_n)$$

kde C_i je vždy identifikátor i-té hodnoty typu. (Objektem C_i nikdy nemůže být číslo, znak (char) či textový řetězec!)

n je kardinalita typu T.

2. Nad typem definovaným výčtem jsou definovány operace pořadí (ord), hodnota následníka (succ), hodnota předchůdce (pred), predikát rovnosti a nerovnosti (=, <>), predikáty uspořádání (<, >, <=, >=) a operace přiřazení (:=).

3. Stojí za povšimnutí, že uspořádání hodnot některých typů nemusí mít v reálném prostředí žádný význam (např. typ "stav osoby", typ barva, typ tvar atd.). V tom případě nezáleží na pořadí, v jakém uvedeme jednotlivé hodnoty v definici typu.

4. Začátečníci velmi často opomíjejí řadu užitečných aplikací typu definovaného výčtem. Za zmínku stojí použití tohoto typu pro indexování pole, při práci s cyklem for, při práci se strukturou case či pro konstrukci množin set.

5. Zjevné potíže dělá začátečníkům vstup/výstup hodnot typu def.výčtem. Často si neuvědomují, že jazyk nemá a nemůže mít standardní prostředky pro oboustrannou konverzi mezi textovým řetězcem, jako vnější representací hodnoty typu definovaného výčtem a vnitřní representací této hodnoty v paměti počítače (již proto, že textový řetězec vnější representace se nemusí shodovat s identifikátorem odpovídající hodnoty; hodnotě LEDEN může na vstupu odpovídat textový řetězec 'LED' - třeba jen proto, aby textové řetězce všech hodnot typu měsíc byly stejně dlouhé, což má praktický význam pro zjednodušení vstupu/výstupu). Vstup/výstup textového řetězce a následující/předcházející konverzi mezi množinou textových řetězců a množinou hodnot typu definovaného výčtem si musí zajistit programátor ve vlastní režii!

2.2.2. Standardní jednoduché typy

Standardní jednoduché typy jsou typy integer, Boolean, real a char. Podrobně jsme se s těmito typy seznámili v [2] a proto jen shrneme :

1. Nad všemi typy jsou definovány operace přiřazení, relace ekvivalence a uspořádání.

2. Typy integer, Boolean a char jsou ordinární typy. Použití funkcí ord, succ a pred je však u těchto typů v některých případech diskutabilní.
   - Funkce ord(i) pro i typu integer má hodnotu i.
   - Funkce succ(i) resp. pred(i) pro i typu integer by se měla používat tam, kde jde o určení následníka resp. předchůdce a neměly by suplovat aritmetický přírůstek, pro který je vhodnější tradiční zápis i+1 resp. i-1. (Např. určení indexu následníka, ale nikoli ke zvýšení počtu o jedničku.)
   - Použití uvedených funkcí, ale také relace uspořádání nad typem Boolean je nevhodné a vede k málo srozumitelnému, kryptogramnímu zápisu.
   - Použití funkcí pred a succ nad typem char s výjimkou podmnožiny číslic je nevhodné, protože s výjimkou číslic není pro všechny znaky Z zaručeno, že

```
succ(Z) = chr(ord(Z)+1)
```

jako důsledek nezávislosti jazyka na použitém kódu (viz [2]).

3. Nad typem integer jsou definovány aritmetické operace sečítání (+), odečítání (-), násobení (*), celočíselné dělení (div) a operace pro získání zbytku po celočíselném dělení - modulo (mod).
Celočíselné dělení má za výsledek celé číslo, vzniklé zanedbáním kladného zbytku. Pro kladné m a n typu integer tedy platí

$$m-n < (m \text{ div } n)*n \leq m$$

Operace modulo se definuje pomocí operace celočíselného dělení vztahem

$$(m \text{ div } n)*n + (m \text{ mod } n) = m$$

Z matematického hlediska je tato operace definována pouze pro kladné argumenty.
Úlohy z oblasti nenumerického zpracování jen zřídka vyžadují operace celočíselného dělení v níž by některý argument byl záporný a pro operaci modulo by to mělo být vyloučeno (i když strojové provedení instrukce to nevylučuje).

4. Nad typem real jsou definovány aritmetické operace sečítání (+), odečítání (-), násobení (*) a dělení (/). Typ real se zobrazuje v pohyblivé řádové čárce a u některých překladačů je možnost volby délky mantisy umožňující za cenu zpomalení aritmetických operací rozšířit jejich přesnost. Nad tímto typem je definována řada standardních funkcí (viz [2]), z nichž je vhodné upozornit na transformační funkce trunc a round provádějící konverzi typu real na integer. Z pravidel kompatibility pro přiřazení vyplývá, že proměnné typu real lze přiřadit výraz typu integer, ale nelze provést přiřazení výrazu typu real proměnné typu integer. Pro typ operace složené ze smíšených operandů real a integer platí zvláštní pravidla ([2]).

5. Nad typem Boolean jsou definovány operace konjunkce (and), disjunkce (or) a negace (not). Všechny relace dávají rovněž za výsledek výraz typu Boolean.

6. Nad typem char je na rozdíl od ostatních ordinálních typů definována také funkce chr, která je inverzní k funkci ord a tudíž pro každé Z typu char platí

$$chr(ord(Z))=Z$$

2.2.3. Typ interval

V praxi se často stává, že proměnná jistého typu může nabýt v průběhu výpočtu pouze hodnoty z jistého, předem známého intervalu. Je-li tento typ ordinální, pak lze definovat typ interval uvedením dolní a horní hranice intervalu jeho hodnotami ve formě konstant takto :

```
type T = min..max
```

Např.

```
type
    ROK=1900..1999
    CISLICE='0'.. '9'
```

Základním smyslem typu interval je poskytnout programátorovi diagnostický prostředek pro kontrolu správnosti probíhajícího výpočtu. Dojde-li k pokusu o přiřazení hodnoty vně intervalu proměnné typu interval, hlásí se chyba. Využití

redundantní informace skýtané typem interval pro detekci chyb je jedním ze základních vlastností vyššího programovacího jazyka. Pro vyspělého programátora by mělo být v reálných programech samozřejmostí běžné používání tohoto typu zejména ve spojitosti s proměnnými, které se v tradičních programech deklarovaly jako integer.

2.3. Strukturované datové typy

2.3.1. Pole

Pole je homogenní, v Pascalu statická datová struktura. Všechny komponenty jsou téhož (bázového) typu. Tímto typem může být kterýkoli definovaný typ. Jednotlivé komponenty pole jsou označeny indexem, což je hodnota typu definovaného jako "typ indexu". Typem indexu může být libovolný ordinální typ.

Pole je struktura s náhodným přístupem k jednotlivým komponentám, což znamená, že v daném okamžiku je každá komponenta stejně dosažitelná.

Pascal neobsahuje operátor typu "konstruktor pole"takový, který by umožnil zápis resp. definici strukturované hodnoty typu pole výčtem hodnot jeho komponent. Hodnota pole se může tudíž ustavit pouze ustavením hodnot všech jeho komponent (např. v cyklu for).*) Operátor typu "selektor složky pole" sestává ze jména pole a z indexu dané složky uzavřeného do hranatých závorek. Nad složkou pole jsou definovány všechny operace daného bázového typu.

Skutečnost, že pole je v Pascalu principiálně jednorozměrné, je významná. N-rozměrnosti se dosáhne tím, že bázovým typem pole je N-1 rozměrné pole. Pomocí selektoru s K indexy prvních (levých) rozměrů (kde K< N) lze zapsat N-k rozměrnou složku struktury pole. Např. je-li definována proměnná POLE typu

```
type TYPPOLE=array[1..5,1..10,1..20] of integer
```

pak zápis

POLE[3] představuje 200 prvků "třetí" matice 10x20 celých čísel

a zápis POLE[3,7] představuje "sedmý" řádek této matice o 20 prvcích

Skutečnost, že index, jako součást selektoru, musí být definovaného ordinálního typu, má významný důsledek, kterým se pole liší od jiných strukturovaných typů. Hodnotu indexu lze tudíž vypočítat. Na místě indexu v selektoru může být nejen konstanta, ale také výraz. Výraz se vyčíslí v okamžiku zpracování selektoru, určí hodnotu indexu a tím i odpovídající komponentu pole. Tato vlastnost je jedním z nejvýznamnějších a nejmocnějších nástrojů programování, ale také zdrojem velmi častých chyb v programu. Jestliže hodnota takového výrazu je vně intervalu specifikovaného typem indexu, specifikuje selektor s tímto argumentem složku pole, která podle definice neexistuje. Vyspělé programovací jazyky indikují chybný přístup k neexistující složce pole. U jazyků, které možnost takové indikace nemají, může docházet k závažným, obtížně zjistitelným chybám.

Podrobnosti o definici typu pole a o práci s polem jsou uvedeny v [2].
K volbě abstraktní datové struktury pole, jako representace údajů o objektu reálného problému v programu, vedou tyto předpoklady :

1. Všechny údaje jsou téhož typu
2. Všechny údaje je nutné současně uchovat v paměti

---

*) Výjimku tvoří textový řetězec - packed array[I] of char

3. Je nezbytný náhodný přístup k jednotlivým údajům

4. Je explicitně znám celkový resp. maximální počet těchto údajů.

Nejsou-li splněny všechny předpoklady, není pole vhodnou representací údajů o objektech reálného problému.

Pro práci s celým polem, t.zn. postupně se všemi jeho komponentami, se používá nejvhodněji explicitní cyklus typu for, v němž řídicí proměnná postupně nabývá všech hodnot typu index daného pole.

2.3.2. Záznam

Záznam je statická datová struktura. Jako jediná struktura Pascalu může být heterogenní, t.zn., že jeho komponenty nemusí být téhož typu. Jednotlivé komponenty záznamu jsou označeny tzv. identifikátorem složky. Na rozdíl od pole, jehož index má spočitatelnou hodnotu, nelze identifikátor složky nahradit žádným jiným objektem. Identifikátory složky záznamu nepřísluší žádnému datovému typu. Není tudíž možné např. použít identifikátor složky záznamu jako parametr v typickém případě universální řadicí procedury, která by měla být schopná řadit pole záznamů o osobách jednou podle složky JMENO a jindy podle složky ROKNAROZENI. Takový případ se musí řešit jinými prostředky.

Pascal neobsahuje operátor typu "konstruktor záznamu" takový, který by umožnil zápis resp. definici strukturované hodnoty typu záznam výčtem hodnot jeho komponent. Hodnota záznamu se může ustavit pouze ustavením hodnot všech jeho komponent (např. posloupností přiřazovacích příkazů). Operátor typu "selektor složky záznamu" sestává z identifikátoru záznamu a z identifikátoru dané složky oddělených tečkou. Nad složkami záznamu jsou definovány všechny operace odpovídajících datových typů.

Složkou záznamu může být opět záznam, a toto rekursivní pravidlo dovoluje vytvářet hierarchické vnořené záznamové struktury neomezené úrovně. Existuje-li záznam o N-té úrovni vnoření záznamů, pak selektor s K identifikátory vyšších úrovní (kde K < N) označuje vnořenou složku na úrovni N-K. Nechť existuje proměnná OSOBA typu TYPOSOBA

```
type TYPOSOBA=record
               JMENO:packed array[1..20]of char;
               ROK_NAROZENI:1850..2000;
               ADRESA:record
                        MISTO:record
                                NAZEV:packed array[1..20] of char;
                                PSC:1..99999
                                end;
                        ULICE:packed array[1..20] of char;
                        CISLO:1..999999
                      end
               end
```

Pak zápis

OSOBA.ADRESA představuje složku na 1.úrovni, typu záznam se složkami MISTO, ULICE, CISLO

OSOBA.ADRESA.MISTO představuje složku na 2.úrovni, typu záznam se složkami NAZEV, PSC

a zápis

OSOBA.ADRESA.MISTO.PSC představuje složku na 3.úrovni, typu interval.

Zvláštní vlastností záznamu je možnost definovat tzv. variantní složku. Je to složka, kterou se proměnné téhož typu záznam mohou vzájemně odlišovat. Tato vlastnost je významná proto, že umožňuje vytvořit strukturu záznamů, která zůstává navenek homogenní přesto, že se její záznamy mohou odlišovat alternativou své variantní složky.

Podrobnosti o definici typu záznam a o práci se záznamy jsou uvedeny v [2]. Proto v tomto odstavci dále shrneme jen některé okolnosti, které souvisí s programovacími technikami.

Při práci se záznamy, lze použít příkazu with. Zápis with r do s znamená, že identifikátory složek proměnné typu záznam r lze uvnitř příkazu s používat bez předpony "r", a předpokládá se její implicitní platnost. Příkaz with slouží ke zjednodušení a zpřehlednění textu programu, ale může také zabránit nadbytečnému vyčíslování při vyhodnocování adresy indexované složky, jak ilustruje tento příklad pole A prvků TYPOSOBA

```
for I:=1 to N do
  with A[I] do
     if (ROKNAROZENI=1965) and (ADRESA.MISTO.PSC=73801)
       then POCET:=POCET+1;
```

Pro zvýšení spolehlivosti programu při práci se záznamy s variantní složkou, se doporučuje používat tzv. rozlišovací složku varianty ve spojení s příkazem case rozlišujícím alternativní zpracování různých alternativ variantní složky, jak to ilustruje příklad výpočtu vzdálenosti dvou bodů zadaných v kartézských nebo polárních souřadnicích (viz [2]). I při nevyužití rozlišovací složky varianty v záznamu by Pascal (podlé své normy-viz [2]) neměl umožnit záměnu alternativ variantní složky. Současné kompilátory však takovou kontrolu v průběhu výpočtu nedělají. Tato skutečnost umožňuje realizovat určité programátorské obraty (triky), které sice nejsou zcela v souladu se zásadami strukturovaného programování, ale vedou k efektivnímu řešení některých problémů. Tyto obraty jsou založeny na znalosti implementace variantní složky záznamu, z níž vyplývá, že všechny alternativy jsou uloženy ve stejné oblasti paměti a vzájemně se překrývají podle své aktuálnosti. Začátek oblasti je pro všechny stejný a délka je rovna velikosti požadované nejrozsáhlejší alternativou varianty. Neexistuje-li ochrana záměny alternativ, pak do určitého prostoru může jedna alternativa vložit údaj jistého typu a druhá alternativa z téhož prostoru (nebo z jeho části) vybrat údaj jiného typu. Tento mechanismus se podobá funkci oblasti COMMON ve Fortranu a může sloužit k neformální konverzi datových typů. Je nutno vážně upozornit, že podobné obraty jsou implementačně závislé a jsou založeny na důkladné znalosti representace datových typů u daného kompilátoru na daném počítači! Mohou zabránit přenositelnosti (portabilitě) programu z jednoho počítače na druhý, která se považuje za zásadní vlastnost moderních programů. Proto se považuje za nutné omezit takové obraty jen na nejdůležitější (nejčastěji systémové) problémy a dokonalou dokumen-

tací upozornit na jejich existenci a funkci. Implementačně závislý úsek programu je nejlépe vymezit ve formě procedury nebo funkce, která se musí při přenosu na jiný počítač přepracovat. Podobné obraty ilustrují dva následující příklady.

Funkce EXP typu integer má za výsledek celočíselnou hodnotu exponentu čísla typu real zobrazeného podle zásad zobrazení na počítačích JSEP, kde v 1.bytu zobrazení je v 1.bitu znaménko mantisy a ve zbývajících 7 bitech exponent zobrazený v kódu transformované nuly.

```
function EXP (X:real):integer;
{Výsledná hodnota funkce je hodnota exponentu čísla X typu real. Funkce je
 implementačně závislá! Předpokládá exponent v 1.bytu zobrazení čísla typu
 real a zobrazení exponentu v kódu transformované nuly.}

type TYPZ=record
          case Boolean of
            true:(ZNAKY:packed array[1..4] of char);
            false:(CISLO:real)
          end;
{Program uloží do alternativy CISLO zpracovávanou hodnotu. Z prvního znaku
 alternativy ZNAKY určí hodnotu exponentu.}

var Z:TYPZ;
    POM:Ø..257;

begin
   Z.CISLO:=X;  {Vložení hodnoty do alternativy CISLO}
   POM:=ord(Z.ZNAKY[1]);
           {konverze 1.znaku na celé číslo}

   if POM>=128
          then {mantisa je záporná, 1.bit je jedničkový}
               EXP:=POM-128-64  {Odstranění znam.bitu a transformace nuly}
          else  {mantisa je kladná}
               EXP:=POM-64     {transformace nuly}
end {funkce EXP}
```

Procedura PREVOD, na základě znalosti implementace datového typu množina, rychle a jednoduše zobrazí 32 bitů čísla typu integer (zobrazení JSEP) ve formě znaků 'Ø' a '1' do pole znaků.

```
type ALTERNATIVA=(PRVNI,DRUHA);
type TYPPOLE=array [Ø..31] of char;
  :
  :
procedure PREVOD (Y:integer; var ZNAKY:TYPPOLE);
 {Implementačně závislá konverze celého čísla Y v zobrazení JSEP do znakové-
  ho pole dvojkových číslic dvojkového zobrazení čísla}

type TYPPOM=record
             case ALTERNATIVA of
             PRVNI:(CELECISLO:integer);
             DRUHA:(MNOZINA:set of Ø..31)
             end;
```

```
{Celé číslo se vloží do alternativy CELECISLO. Stejný prostor se potom
 interpretuje jako množina, v níž existence jednotlivých prvků je dána
 jedničkovou hodnotou odpovídajícího bitu }

var POM:TYPPOM;
    I:Ø..31;

begin
    POM.CELECISLO:=Y;  { Vložení hodnoty do alternativy CELECISLO }
    for I:=Ø to 31 do  { Generování všech prvků množiny }
        if I in POM.MNOZINA
                then ZNAKY[I] :='1'    { Ustavení pole znaků }
                else ZNAKY[I] :='Ø'
end  {procedury PREVOD }
```

Datový typ záznam svou strukturou odpovídá řídicí struktuře "složený příkaz", kde příkazové závorky begin a end uzavírající sekvenci různých příkazů vytvářejí jeden (složený) příkaz. Záznam, jako samostatný objekt, slouží převážně jako pomocný datový prostor. Většinou se záznam používá jako konstituční typ jiných datových struktur(pole, souboru, dynamicky sřetězených struktur aj.).

2.3.3. Množina

Množina je statická homogenní struktura postavená nad bázovým typem, jímž může být vhodný ordinální typ. Množina, jako datová struktura, neobsahuje hodnoty prvků bázového typu, ale pouze informaci o jejich existenci či neexistenci v dané množině. Pro zobrazení této informace stačí jeden bit pro každý prvek.

Množina je v Pascalu jediným ze strukturovaných datových typů, který má definován konstruktor pro ustavení hodnoty výčtem jednotlivých ustavujících komponent. Konstruktor má tvar [p,q,...,r] , kde p,q,...,r je seznam hodnot bázového typu množiny. Zápis [ ] představuje pro všechny typy množin množinu prázdnou. Nad množinou není definován operátor typu selektor, ale existence jednotlivých prvků se může ustavit Booleovskou hodnotou existenční operace in.

Operátor přiřazení nad množinami je definován pro operandy kompatibilní vzhledem k přiřazení (viz [2]). Nad vzájemně kompatibilními typy množin jsou definovány tyto relační operátory: ekvivalence(=), nonekvivalence(<>), inkluze množiny K v množině L (K<=L) a inkluze množiny L v množině K (K>=L). Dále jsou pro tyto operandy definovány tyto množinové operace: průnik množin (*), sjednocení množin (+), rozdíl množin (-) a operátor existence prvku v množině (in).

Pro konkrétní aplikace je významný způsob implementace množiny v paměti počítače. Typickým způsobem je zobrazení existence prvku v množině jedničkovou hodnotou bitu, jehož pořadí je v prostoru zaujímaném množinou shodné s ordinálním číslem prvku bázového typu. Operace nad množinami mohou být velmi rychlé, protože je lze přímo realizovat základními strojovými instrukcemi. Pro účinnost těchto operací je ale nutné, aby prostor potřebný pro reprezentaci množiny nepřesáhl velikost, se kterou pracují tyto strojové operace. Tato velikost bývá nejčastěji určena velikostí (nebo jejím násobkem) registrů daného počítače. Jestliže má tento prostor velikost např. 64 bitů (2 spojené univerzální registry počítače JSEP), je možné definovat množinu obsahující pouze prvky, jejichž ordinální čísla jsou v rozsahu Ø..63. Z toho vyplývá skutečnost, že v běžně provozo-

vaných implementacích jazyka Pascal na počítačích JSEP nelze vytvářet množinu nad typem char, ani nad intervalem tohoto typu, vymezujícím běžně používané tisknutelné znaky.*)

V praktických problémech se jako bázový typ při práci s množinami nejčastěji uplatňuje typ definovaný výčtem.

2.3.4. Soubor

Společnou vlastností strukturovaných datových typů, o nichž se až dosud hovořilo, je konečná hodnota jejich mohutnosti (kardinality). Řada vyšších datových struktur jako je lineární seznam, strom, graf, jsou charakteristické tím, že jejich mohutnost je nekonečná. Tato skutečnost má, na rozdíl od předcházejících strukturovaných typů, některé významné praktické důsledky.

Jedním z nejdůležitějších důsledků je skutečnost, že velikost paměťového prostoru pro reprezentaci takových struktur není známá v době překladu a může se měnit v průběhu zpracování programu, tzn., že tyto struktury jsou dynamické a vyžadují mechanismus dynamického přidělování paměti. Volba reprezentace těchto struktur se dá dělat pouze na základě znalosti jednoduchých operací nad těmito strukturami a na základě znalosti četnosti použití takových operací. Jednotlivé alternativy reprezentace se totiž mohou výrazně lišit účinností alternativních implementací jednotlivých operací. V důsledku toho, že o potřebných operacích ani o typické četnosti jejich použití nelze udělat obecný závěr, nejsou tyto struktury většinou zahrnuty mezi standardní struktury vyšších jazyků. Z uvedených skutečností vyplývá i doporučení, že je účelné vyhnout se dynamickým strukturám tam, kde lze problém řešit pomocí základních datových struktur. Poskytuje-li se vyšší strukturovaný typ standardně, platí často omezení, že jeho složkami mohou být pouze jednoduché nebo základní strukturované datové typy.

V Pascalu je takovým typem, reprezentujícím lineární seznam, sekvenční soubor typu file. Vhodná volba operací nad lineárními seznamy, které lze použít nad typem file, umožňuje účinnou reprezentaci datové struktury na daném paměťovém zařízení a zajišťuje dostatečně jednoduchý mechanismus dynamického přidělování paměti, zbavující programátora nutnosti zabývat se detailními problémy implementace.

Podstatou sekvenčního přístupu k lineárnímu seznamu je skutečnost, že v daném okamžiku je možný přístup pouze k jedné složce struktury. Aktuální poloha přístupového mechanismu určuje, která složka je zpřístupněna. Polohu přístupového mechanismu (přístupové proměnné) lze měnit operacemi nad souborem obvykle tak, že se zpřístupní následující nebo první složka souboru. Důsledkem sekvenčního přístupu ke složkám souboru je skutečnost, že proces vytváření souboru zápisem jeho složek (jejich přidáváním na konec souboru) a proces průchodu souborem a pasivní zpřístupňování jeho složek (čtení složek souboru) jsou navzájem různé procesy a nemohou se směšovat. Proto může být soubor v jednom ze dvou stavů: vytváří se (zápis), nebo se jím prochází (čtení).

---

*) Na počítačích ADT pracují implementace jazyka PASCAL, které umožňují definovat množinu o mohutnosti 128. Lze tedy pracovat s ordinálními čísly 0..127 a také s typem char, protože je implementován 7-bitovým kódem ASCII (ISO 7).

Výhoda sekvenčního přístupu se projeví zejména v případě, kdy se pro representaci souborů použijí vnější paměťová zařízení. Z jejich využití vyplývá nezbytnost přenosu dat mezi různými paměťovými médii. Mechanismus vyrovnávání (buffering) úspěšně skryje mnohé obtíže vyvstávající s fyzickým přenosem dat a zaručí optimální využití datových "zdrojů" použitých v počítačovém systému. Sekvenční přístup je charakteristickou vlastností všech paměťových zařízení, která využívají pohybujícího se mechanismu (především paměti na magnetických páskách, také na discích a bubnech), ale i jiných principů.

Definice souborů, operace nad nimi i způsob práce se soubory jsou detailně popsány v [2]. Proto jen shrňme nejzávažnější skutečnosti :

1. Nad souborem jsou v Pascalu definovány operátory pro ustavení do režimu zápisu (rewrite), pro ustavení do režimu čtení (reset), pro vložení prvku na konec souboru (put), pro čtení běžného prvku souboru (get) a predikát polohy přístupové proměnné (eof), který je pravdivý, jestliže je polohou určen konec souboru.

2. Nad souborem jsou v Pascalu definovány také operátory read a write.
   Zápis

   read(x,v) je ekvivalentní s v:=x↑ ; get(x)

   a zápis

   write(x,e) je ekvivalentní s x↑:=e; put(x)

   Smyslem těchto příkazů je jejich jednoduchost a možnost ignorovat existenci přístupové proměnné x↑ , jejíž hodnota je v některých případech nedefinované. Operace read (x,v) se smí provést při splnění podmínky not eof(x) a operace write(x,e) při splnění podmínky eof(x).

3. Zvláštním případem souboru je typ file of char - tzv. textový soubor, který je v Pascalu označen standardním identifikátorem typu "text". Pro práci se soubory typu text je definován oddělovač řádků v textu - znak "eoln" a funkce eoln(f), která má při přečtení znaku "eoln" hodnotu true a při čtení jiných znaků false.

4. Operace read a write umožňují automatickou konverzi mezi typu integer, Boolean a real a souborem typu text (viz [2]).

5. V Pascalu jsou definovány dva standardní soubory typu text, označené identifikátory input a output, kde input je standardní vstupní soubor určený výhradně pro čtení a output je standardní výstupní soubor určený výhradně pro výstup. Ustavení režimu těchto souborů se provádí automaticky, při provedení prvního příkazu programu.

6. Komponentami souboru mohou být i strukturované typy s neomezenou hladinou vnoření. Současně provozované překladače však neumožňují, aby komponentou souboru byl opět soubor, ani jiný typ obsahující uvnitř sebe soubor.

2.3.5. Zhušťování

Zhušťování zobrazení strukturovaných údajů je takový způsob zobrazení, který vede k úspornějšímu využití paměťového prostoru, nejčastěji za cenu prodloužení času potřebného pro přístup ke složkám datové struktury.

Mechanismus zhušťování lze charakterizovat takto. Každý kompilátor na daném počítači určuje standardní velikost paměti pro zobrazení jednotlivých primitivních

datových typů. Jestliže je kardinalita definovaného typu menší, než největší číslo zobrazitelné ve standardně přidělovaném prostoru, lze za jistých okolností zobrazit tento typ na menším prostoru za podmínky, že je tento typ komponentou strukturovaného typu. Mechanismus zhušťování přitom závisí také na velikosti nejmenší adresovatelné jednotky paměti, na možnostech adresování a vlastnostech instrukcí daného počítače, především ale na implementovaných vlastnostech daného kompilátoru. Mechanismus zhušťování se týká strukturovaných typů, jejichž vnitřními komponentami jsou typy: char, Boolean, interval a typ definovaný výčtem.

Pascal umožňuje definici všech strukturovaných typů ve zhuštěné podobě tak, že se před identifikátor typu uvede standardní slovní symbol packed. Z hlediska programu je způsob práce i zápis operací se zhuštěnou datovou strukturou shodný s nezhuštěnou formou odpovídající struktury.

Při práci s typem packed array nastává nepříznivá situace při nutnosti postupného přístupu ke všem prvkům zhuštěného pole, protože opakované vyhodnocování složité přístupové (mapovací) funkce způsobí výraznou časovou ztrátu ve srovnání s opakovaným přístupem do nezhuštěného pole. Tuto nevýhodu mohou kompenzovat standardní procedury "pack" a "unpack", které provádějí vzájemnou transformaci mezi zhuštěným a nezhuštěným polem téhož bázového typu. Bližší podrobnosti viz [2].

Zvláštním případem zhuštěného pole je packed array jehož bázovým typem je typ char. Takové struktuře se říká také "textový řetězec" a na rozdíl od ostatních polí jsou nad tímto typem definovány další operátory. Operátory srovnání ($=, <>, <=, <, >=, >$) definují lexikografickou relaci dvou textových řetězců, kterou lze popsat takto :

> Jestliže se dva textové řetězce o n znacích shodují v K prvních znacích a liší se na K+1 znaku (kde $K \in \langle \emptyset, n+1 \rangle$), pak "větší" z obou je ten řetězec, jehož k-tý znak je větší. V jiném případě jsou oba řetězce shodné.

Pro datovou strukturu textový řetězec je definován konstruktor tvaru $'ch_1\ ch_2\ \ldots\ ch_n'$, kde ch_i je konstanta typu char. Tento konstruktor umožňuje např. ustavení hodnoty proměnné S typu packed array [1..4] of char zápisem

```
S:='TEXT'
```

Textové řetězce je možno přímo zapisovat do textového souboru (viz příkaz write v [2]). Některé implementace Pascalu připouštějí i čtení textových řetězců z textových souborů.

Mechanismus zhušťování záznamu - typu packed record - se vztahuje na ty strukturované složky záznamu, které samy nejsou označeny jako packed, na všech vnitřních úrovních.*) Kromě toho může tento mechanismus účelným uspořádáním jednotlivých (nestrukturovaných) složek dosáhnout úspornějšího uložení dat v paměti (např. uložit několik "malých" složek do jednoho slova paměti). Tato vlastnost nevede k výraznému prodloužení přístupové doby ke složce záznamu; běžně provozované kompilátory ji však nevyužívají. Využití této vlastnosti by bránilo neformální, implementačně závislé transformaci typů pomocí variantních složek záznamu, ilustrované na příkladech v odst.2.3.2.

---

*) FEL PASCAL ADT zhušťuje jen složky záznamu na nejbližší úrovni

2.4. S h r n u t í

Struktura údajů a algoritmus, který nad ní pracuje, tvoří dialektickou jednotu vytvářející program, jak je zdůrazněno i v titulu významné Wirthovy publikace [1]. Skutečnost, že datové i řídicí struktury vycházejí z téhož kompozičního principu, znázorňuje tab. 2.1.

kompoziční princip	řídicí struktura	datová struktura
Atomický prvek	Přiřazovací příkaz	Skalární typ
Sekvence	Složený příkaz	Záznam
Explicitní počet opakování	Příkaz for	Pole
Alternativa	Příkaz if Příkaz case	Záznam s variantní složkou
Implicitní počet opakování	Cykly write,repeat	Soubor
Rekurze	Příkaz procedury	Rekurzivní datový typ
Obecný graf	Příkaz skoku	Struktura sřetězená pomocí ukazatelů

Tab. 2.1. Souvislost mezi řídicími a datovými strukturami

Hlavní vlastnosti základních datových struktur shrnuje tab. 2.2.

Struktura	Pole	Záznam	Množina
Deklarace	a:array[I] of T_o	r:record S_1:T_1; ⋮ S_n:T_n end	s:set of T_o
Selektor	a[i] (i ∈ I)	r.S(S∈ {S_1..S_n})	-
Přístup ke složce	Selektor s vypočitatelným indexem i	Selektor s označovačem složky	Test na přítomnost operátorem in
Typ složek	Všechny téhož typu T_o (homogenní)	Mohou se vzájemně lišit (heterogenní)	Všechny téhož bázového ordinálního typu T_o
Kardinalita	$card(T_o)^{card(I)}$	$\prod_{i=1}^{n} card(T_i)$	$2^{card(T_o)}$

Tab. 2.2. Shrnutí vlastností základních datových struktur

2.5. L i t e r a t u r a

[1] Wirth,N. : Algoritmus + Data Structures = Programs.
Prentice-Hall, Englewood Cliffs, New York, USA 1973

[2] Rábová,Z., Češka,M., Honzík,J., Hruška,T. : Počítače a programování,
skriptum FE VUT, SNTL 1982