

## **Programování a programovací jazyky PASCAL**

---

**ZÁVODY APLIKOVANÉ KYBERNETIKY**

# **Programování a programovací jazyky PASCAL**

Autor: Doc. ing. Zdeněk Havlíček, CSc.

## OBSAH

0. Úvod .....	7
1. Proč jazyk Pascal? .....	9
1.1 Historický vývoj jazyka .....	9
1.2 Kriteria při vyběru programovacího jazyka .....	10
1.2.1 Obecnost .....	10
1.2.2 Spolehlivost .....	11
1.2.3 Možnost údržby .....	11
1.2.4 Přenositelnost .....	11
1.2.5 Jednoduchost .....	12
1.2.6 Efektivnost .....	13
1.2.7 Přijateľnost .....	13
1.3 Základní charakteristiky Pascalu .....	14
1.3.1 Typy .....	14
1.3.2 Deklarace a pravidla viditelnosti .....	15
1.3.3 Možnosti strukturalizace .....	16
.. Základy jazyka Pascal .....	18
2.1 Základní prvky jazyka .....	18
2.1.1 Základní symboly .....	18
2.1.2 Rezervovaná slova .....	19
2.1.3 Identifikátory .....	19
2.1.4 Standardní identifikátory .....	20
2.1.5 Oddělovače .....	21
2.2 Struktura programu .....	22
2.2.1 Obecná organizace programu .....	22
2.2.2 Úvod do podprogramů .....	24
2.3 Jednoduché deklarace .....	26
2.3.1 Předdefinované typy dat .....	26

2.3.1.1 Číselné typy dat .....	27
2.3.1.2 Symbolické typy dat .....	32
2.3.2 Deklarace proměnných .....	35
2.3.3 Deklarace konstant .....	37
2.4 Výrazy .....	38
2.4.1 Aritmetické výrazy .....	38
2.4.2 Logické výrazy .....	40
2.5 Jednoduché příkazy .....	40
2.5.1 Přiřazovací příkaz .....	41
2.5.2 Příkaz procedury .....	42
2.5.3 Vstupy/výstupy .....	43
2.5.4 Prázdný příkaz .....	47
2.6 Základní strukturované příkazy .....	47
2.6.1 Složený příkaz .....	47
2.6.2 Příkaz If .....	49
2.6.3 Příkaz While .....	52
2.7 Ostatní strukturované příkazy .....	54
2.7.1 Příkaz For .....	54
2.7.2 Příkaz Repeat .....	57
2.7.3 Příkaz Case .....	59
2.8 Příkaz skoku GoTo .....	61
2.9 Syntetický příklad .....	63
3. Typy dat .....	65
3.1 Úvod .....	65
3.2 Jednoduché typy .....	67
3.2.1 Výčtový typ .....	68
3.2.2 Typ interval .....	74
3.3 Úvod do strukturovaných typů .....	78
3.4 Typ pole .....	79

3.4.1 Deklarace .....	79
3.4.2 Používání .....	81
3.4.3 Příklad .....	84
3.5 Typ záznam .....	86
3.5.1 Deklarace .....	86
3.5.2 Používání .....	87
3.5.3 Příkaz With .....	89
3.5.4 Variantní záznamy .....	91
3.6 Typ množina .....	95
3.6.1 Deklarace .....	95
3.6.2 Používání .....	96
3.7 Typ soubor .....	99
3.7.1 Typové soubory .....	99
3.7.2 Textové soubory .....	109
3.7.3 Netypové soubory .....	115
3.8 Typ řetězec .....	119
3.8.1 Deklarace .....	119
3.8.2 Používání .....	120
3.8.3 Souhrnný příklad .....	125
3.9 Typové konstanty .....	128
3.9.1 Deklarace .....	128
3.10 Identita a kompatibilita typů .....	130
4. Podprogramy .....	132
4.1 Pojem podprogram .....	132
4.2 Deklarace a vyvolávání podprogramů .....	133
4.2.1 Procedura .....	135
4.2.2 Funkce .....	136
4.3 Parametry .....	137
4.3.1 Úvod .....	137

4.3.2 Přenos hodnotou .....	139
4.3.3 Přenos odkazem .....	141
4.3.4 Další možnosti přenosu .....	143
4.3.5 Obecná pravidla pro přenos parametrů .....	144
4.4 Vztahy mezi podprogramy .....	145
4.4.1 Rozsah a viditelnost deklarací .....	145
4.4.2 Komunikace mezi podprogramy .....	148
4.5 Rekurzivní volání .....	156
4.5.1 Rekurze jednoduchá .....	156
4.5.2 Vzájemná rekurze - direktiva Forward .....	157
4.6 Předdeklarované podprogramy .....	158
4.6.1 Předdeklarované procedury .....	159
4.6.2 Předdeklarované funkce .....	160
5. Dynamické proměnné a typ ukazatel .....	162
5.1 Statické a dynamické proměnné .....	162
5.2 Deklarace .....	163
5.3 Přístup k dynamickým proměnným .....	164
5.4 Operace s dynamickými proměnnými a s proměnnými typu ukazatel .....	165
5.5 Příklad na používání dynamických proměnných ....	170
5.6 Význam dynamických proměnných a typu ukazatel ...	172
6. Několik rad k programování .....	174
6.1 Používání identifikátorů .....	174
6.1.1 Označování konstant .....	176
6.1.2 Označování typů .....	178
6.1.3 Označování proměnných .....	180
6.1.4 Označování podprogramů .....	180
6.2 Používání příkazu skoku .....	181
6.3 Organizace programů .....	182

7. Turbo Pascal .....	184
7.1 Charakteristika Turbo Pascalu .....	184
7.2 Editor .....	186
7.3 Kompilátor .....	188
7.3.1 Vkládané soubory .....	188
7.3.2 Systém segmentace .....	189
7.3.3 Režimy komplikace .....	190
7.4 Ladící prostředky .....	191
7.4.1 Režimy práce s komplikovaným souborem .....	192
7.4.2 Direktivy komplikátoru .....	192
7.4.3 Ošetření chyb .....	194
Literatura .....	197

## Úvod

Předkládaný učební text je zaměřen na programovací jazyk Pascal, o kterém je obecně známo, že přináší mnoho výhod. Proto také jako první je zařazena kapitola "Proč jazyk Pascal", ve které jsou uvedeny jak obecné požadavky na programovací jazyky, tak i tři základní charakteristiky Pascalu.

Hlavní a nejdůležitější je druhá kapitola "Základy Pascalu". Teprve po jejím úspěšném zvládnutí doporučujeme pokračovat ve studiu dalších kapitol.

V třetí a čtvrté kapitole tohoto učebního textu, tj. v kapitolách "Typy dat" a "Podprogramy" se vyskytuje celá řada příkladů. Všechny uváděné příklady byly odladěny v Turbo Pascalu, ve verzi 3.0 pod operačním systémem MS-DOS.

Pátá kapitola "Dynamické proměnné a typ ukazatel" stručně seznamuje s prostředky, které umožňují vytvářet složité datové struktury.

V šesté kapitole "Metodické rady k programování" jsou soustředěna některá doporučení pro praktické programování. Lze předpokládat, že tato kapitola bude mít největší význam při vytváření vlastních programů.

Na závěr učebního textu jsou uvedeny základní způsoby práce v Turbo Pascalu. Orientace na Turbo Pascal je záměrná, neboť tato implementace se stává postupně standardem jazyka Pascal na mikropočítačích. Se vzrůstajícím počtem mikropočítačů v zemědělských podnicích se dá očekávat větší využívání jazyka Pascal v implementaci Turbo či v některé implementaci obdobné (např. Mikro Pascal).

Celková logika jazyka Pascal je poměrně jednoduchá, ale v mnoha učebnicích je velmi složitě definována. Tím se zvyšují nároky na čtenáře a stává se, že někteří uživatelé raději programují v jazyku, který je méně přísně definován. Z tohoto důvodu jsou v tomto učebním textu některé definice zestrojeny a také syntaktické diagramy jsou zjednodušeny. Vše, co je zařazeno do poznámek, může být při prvním čtení ignorováno, neboť poznámky obsahují jen doplňkové informace k základnímu výkladu a konkrétní realizaci probírané látky v Turbo Pascalu. Čtenář, který má zájem o podrobnější studium, může využít literaturu, jejíž seznam je uveden v závěru.

Při studiu tohoto učebního textu jsou předpokládány jen minimální znalosti z algoritmizace úloh. Pro úspěšné zvládnutí probírané látky jsou k dispozici návazná skripta pro cvičení, kde jsou uvedeny jak další řešené příklady, tak úkoly k procvičení a opakování.

I když zpracování tohoto učebního textu byla věnována velká pozornost a maximálně byly využívány dostupné technické a programové prostředky (mikropočítač s textovým editorem), je možné, že v textu se vyskytnou určité nedostatky. Proto uvítáme všechny náměty a připomínky, které pomohou zlepšit kvalitu této učební pomůcky.

## 1. Proč jazyk Pascal?

Otázka, kterou pokládá název této první kapitoly, je velmi důležitá, neboť vytváří předpoklady k výběru a ke studiu nového programovacího jazyka.

Vzrůstající úloha programového vybavení při vytváření automatizovaných informačních systémů a velký počet programovacích jazyků disponibilních na většině počítačů přivádí uživatele k zamýšlení, který programovací jazyk je vhodnější pro vytváření aplikace. Je třeba zvážit různá kriteria, aby bylo přijato objektivní rozhodnutí.

Historický přehled vývoje jazyka Pascal, zhodnocení objektivních kriterií při výběru programovacího jazyka, jakož i uvedení tří základních charakteristik Pascalu jistě přispěje k poznání kvalit tohoto jazyka.

### 1.1 Historický vývoj jazyka Pascal

Když se v roce 1969 zrodil na Polytechnicke univerzitě v Curychu jazyk Pascal, byl to jen jeden z mnoha nových programovacích jazyků.

Jazyk Pascal se nejdříve prosadil ve výuce programování na vysokých školách.

V roce 1975 Pascal již široce pronikl do oblasti výzkumu. Některé velké počítačové firmy (např. Texas Instruments, ICL, aj.) zvolily jazyk Pascal pro vývoj nového programového vybavení. V této době existovaly komplátory Pascalu jen pro velké počítače. V uživatelské sféře se Pascal prosadil až později.

S příchodem mikropočítačů se objevují i nové

kompilátory Pascalu. Začátkem osmdesátých let největšího rozšíření dosáhly tyto implementace jazyka:

- Pascal UCSD - z kalifornské univerzity v San Diegu a
- Pascal MT+ - od společnosti Microsoft.

Zatím nejúspěšnější implementace jazyka Pascal se objevila v roce 1984, kdy společnost Borland International uvedla na trh Turbo Pascal. V současné době (listopad 1987) tato společnost uvádí, že prodala více než 600 000 kusů tohoto kompilátoru. Toto široké rozšíření je podmíněno jak vysokou kvalitou kompilátoru, tak i přehledným zpracováním dokumentace. Nejnovější verze Turbo Pascalu 4.0 přináší další prostředky pro snadnější vývoj programů (např. interaktivní help).

### 1.2 Kriteria při výběru programovacího jazyka

Při výběru programovacího jazyka můžeme zvažovat několik kriterií. Dosud neexistuje programovací jazyk, který by plně uspokojoval všechny požadavky a zároveň maximálně vyhovoval sedmi dále uvedeným kriteriím. Cílem dalšího výkladu není podrobné posouzení Pascalu, ale spíše jde o všeobecné zhodnocení nároků na programovací jazyky a o stručné uvedení některých možností Pascalu.

#### 1.2.1 Obecnost

Programovací jazyk musí být obecným, tzn., že má být minimálně spojen s určitou skupinou aplikaci nebo s určitým typem stroje.

Pascal je použitelný pro vyjádření různých algoritmů, jeho kompilátory jsou k disposici na většině počítačů.

#### 1.2.2 Spolehlivost

Vytváření spolehlivých programů je vlastně základním cílem programování. Dříve se uvažovalo, že spolehlivost programu závisí především na programátorevi a ne na použitém programovacím jazyku.

Pascal hnutí programátora k systematickému definování každého objektu. Tyto definice umožňují přesnou kontrolu správnosti programu jak během komplikace, tak v průběhu vykonávání programu.

#### 1.2.3 Možnost údržby

Skutečnost, že údržba programů tvoří často největší část nákladů při programování (až 60% pro složité systémy), způsobuje, že toto kriterium je jedním z nejpodstatnějších.

Možnost údržby programu je přímo spojena s jeho čitelností. Čitelnost je jak funkcí daného programu, tak programovacího jazyka.

Snadnost, s níž může kdokoliv opravit a rozšířit program, vytváří vhodné podmínky pro jeho údržbu. Tato vlastnost závisí na čitelnosti a zvláště pak na přítomnosti vhodných řídících struktur jazyka.

Možnosti ve strukturování dat i akcí na jedné straně a způsob zápisu programů na straně druhé vytvářejí předpoklady pro snadnou údržbu programů napsaných v Pascalu.

#### 1.2.4 Přenositelnost

Úroveň přenositelnosti se měří úsilím, které je třeba vykonat při převodu programu z jednoho stroje na jiný.

Tento cíl vystupuje do popředí právě při velkém rozvoji

současné počítačové techniky. Přeče při každé změně počítače se nebudou přepisovat všechny programy.

Přenositelnost programů je úzce spojena s úrovní normalizace jazyka, s přesností jeho definice a se způsobem, s jakým výrobce kompilátoru zajistil danou implementaci.

I když neexistuje oficiální československá norma jazyka Pascal, přesto však existuje dobrá úroveň přenositelnosti programů, neboť rozdílné implementace jazyka se příliš neliší. Tento výsledek vzniká kombinací dvou jevů:

první spočívá v přesné definici jazyka Pascal, která byla vytvořena již jeho autorem N. Wirthem a později zpracována ve formě mezinárodní normy [viz lit. 6]. Pascal je lépe definován než jiné jazyky;

druhý jev je spojen s metodami implementace kompilátoru Pascalu. Je možno uvést např., že kompilátor Turbo-Pascalu je doplněn instalacním programem, který umožňuje instalaci kompilátoru na téměř libovolný typ mikropočítače.

#### 1.2.5 Jednoduchost

Jednoduchost programovacího jazyka se vyjadřuje:

- stupněm obtížnosti při studiu a používání a
- stupněm obtížnosti při vytváření implementace.

Programovací jazyk musí být snadno naučitelný, nesmí být určen jen pro elitní programátory. Studium Pascalu je poměrně snadné. I když při prvním seznámení vzniká pocit, že se jedná o složitý jazyk. Uvádí se, že během týdenního školení se dá dobře zvládnout.

#### 1.2.6 Efektivnost

Efektivnost programového vybavení se obvykle vyjadřuje mírou spotřeby zdrojů počítače, např. potřebným časem procesoru či obsazeným rozsahem operační paměti. Vzhledem ke stále se zvyšující výkonnosti počítačů je třeba brát v úvahu též produktivitu programátorské práce.

Každý vyšší programovací jazyk musí v dané implementaci umožnit tvorbu programů, které budou efektivní při vlastním zpracování. Z tohoto hlediska lze implementaci Turbo Pascal považovat za velmi zdařilou, neboť komplikací vzniká cílový program, který zabírá relativně malý prostor paměti.

Jazyk Pascal byl navržen tak, aby umožňoval psát programy systematicky, přehledně a poskytoval možnosti pro ověřování správnosti algoritmů. Lze říci, že Pascal obsahuje celou řadu prvků, které podporují tzv. strukturované programování.

Produktivita programátorské práce také závisí na kvalitě a rychlosti kompilátoru. Je možno poznamenat, že i z tohoto hlediska je Turbo Pascal velmi účinný; např. komplikace programu, který má 2 500 příkazů, vyžaduje na mikropočítači typu IBM/PC jen asi 50 sekund.

#### 1.2.7 Přijatelnost

Každý programovací jazyk má určité "dobré" vlastnosti. Avšak o jeho úspěšnosti nerozhoduje jen fakt, jak je přijat odbornou veřejností či rozšířen ve školství a výzkumu, ale především jak je podporován průmyslem (respektive výrobci počítačů).

Z více než 2 000 existujících programovacích jazyků jen

asi šest jich plně vyhovuje kriteriu přijatelnosti, tzn., že se jedná o jazyky rozšířené, uznávané a používané.

Na základě toho, co bylo uvedeno v oddíle o historii Pascalu, lze říci, že programovací jazyk Pascal byl přijat.

### 1.3 Základní charakteristiky Pascalu

Aby odpověď na otázku "Proč jazyk Pascal?" byla úplná, je třeba být přesnější a uvést nejdůležitější charakteristiky jazyka. Při porovnání s jinými jazyky se nejčastěji uvádějí tyto tři charakteristiky: typy, deklarace a viditelnost deklarací a možnosti strukturovaného programování.

#### 1.3.1 Typy

Typem proměnné označujeme množinu hodnot, kterých proměnná může nabývat, tzn., že typ vyjadřuje variační rozpětí proměnné.

Typ také poskytuje informaci, které operace jsou pro danou proměnnou uskutečnitelné.

Pojem typ není nový, např. v jazyku Fortran se využívají typy Integer a Real. Pascal přináší zobecnění pojmu typ a rozpracovává jeho výhody, které jsou podstatně především z hlediska spolehlivosti a údržby:

- typy zvyšují spolehlivost programu tím, že umožňují jak při komplikaci, tak při vykonávání programu kontrolu správnosti přiřazení. Tento mechanismus umožňuje automatickou detekci mnoha druhů různých chyb;

- typy zlepšují údržbu tím, že zvyšují čitelnost programu.

Povinné přiřazování nějakého typu každé proměnné a možnost

definice vlastních typů jsou dva významné faktory v dokumentaci programu.

#### 1.3.2 Deklarace a pravidla viditelnosti

Program slouží k popisu akcí s objekty. Při zpracovávání programu v určitém okamžiku jsou zapotřebí jen některé objekty. Odtud vychází myšlenka, aby text programu byl rozdělován na části, které odpovídají určité akci. V každé části textu programu celá množina objektů není potřebná a tedy některé z těchto objektů mohou být neviditelné. Z hlediska potřeby a spolehlivosti je žádoucí, když jsou viditelné jenom ty objekty, které jsou nutné pro zpracování dotyčné části algoritmu.

V programovacích jazycích jako je Basic všechny objekty jsou viditelné a tedy přístupné z kteréhokoliv místa programu. V jazyku Fortran i v jazyku Basic programátor nemusí definovat jednoduché proměnné. Stačí zapsat jméno proměnné v nějakém příkazu a typ proměnné je určen automaticky. Tato proměnná bude existovat po celou dobu životnosti programu. Tyto implicitní deklarace přinášejí řadu nevýhod, např.:

- údržba programu je komplikovaná a drahá, neboť čitelnost programu je obtížná;
- existuje nebezpečí závažných chyb tím, že se v programu omylem pozmění jméno proměnné. Např. záměna dvou písmen ve jméně proměnné (ALFA --> AFLA) způsobuje tvorbu nové proměnné.

Do Pascalu jsou zařazeny dvě podstatné charakteristiky:

- proměnná nemůže být zapsána v příkazu, pokud nebude

předem uvedena v deklaraci. Každá odchylka od tohoto pravidla zapříčiní chybu při kompliaci;

- text programu může být rozdělen na podprogramy, které mají vlastní objekty. Tyto objekty jsou nepřístupné z vnějšího podprogramu. Čtením programu ihned rozpoznáme dobu životnosti každé proměnné a tím se zjednodušuje údržba.

### 1.3.3 Možnosti strukturalizace

I když pojem strukturované programování se nyní používá při programování ve všech jazycích, je třeba se zmínit o podstatě tohoto pojmu. Mezi zásady strukturovaného programování patří především návrh programu v co nejabstraktnější podobě přistupem shora-dolů a přítomnost vhodných řídících struktur.

Pascal umožnuje a podporuje metodu shora-dolů. Tato metoda spočívá v rozdělování řešeného problému na dílčí části - podprogramy.

Pro zápis jednotlivých základních struktur algoritmů jsou v Pascalu vhodné příkazy:

- sekvenční struktura se vyjadřuje složeným příkazem;
- alternativní struktura se vyjadřuje příkazem If, případně příkazem Case;
- iterativní struktura se vyjadřuje příkazy:

While,

Repeat - Until a

For.

Poznamenejme, že tyto tzv. strukturované příkazy se mohou dle potřeby kombinovat a tak se vyvíjí předpoklady k zápisu přehledných a správných programů.

Na závěr této kapitoly je třeba podotknout, že Pascal nemá některé uzavřené příkazy typu IF - ENDIF, které jsou vyžadovány modernějšími programovacími jazyky.

## 2. Základy jazyka Pascal

V této kapitole jsou uvedeny základy jazyka Pascal. Čtenář se seznámí se základními charakteristikami jazyka i se strukturou programu v jazyce Pascal.

### 2.1 Základní prvky jazyka

Pascal, tak jako každý jiný programovací jazyk, musí být jasně definován. V tomto oddíle budou uvedeny základní prvky jazyka, z nichž se vytvářejí programy.

#### 2.1.1 Základní symboly

V Pascalu lze používat písmena, číslice a speciální symboly. Ve standardní verzi lze používat 26 písmen latinky, tzn. písmena A až Z, přičemž není rozdíl mezi velkými a malými písmeny. Např. jména ALFA a alfa jsou považována za stejná. Stejný význam jako písmeno má také znak \_ (podtržení).

Používá se deset arabských číslic 0 až 9.

Speciální symboly - syntaktické jednotky jazyka se specifickým významem. Jsou to tyto jednoduché znaky:

+ - \* / = ^ < > ( ) [ ] { } . , : ; ' # \$

Některé dvojice jednoduchých znaků mají jednoznačný význam:

Operátor přiřazení: :=

Relační operátory: <>    <=    >=

Oddělovač intervalu: ..

Závorky: ( . a . ) lze použít místo [ a ]

Poznámky: (\* a \*) lze použít místo { a }

### 2.1.2 Rezervovaná slova

Rezervovaná slova, někdy označovaná též jako klíčová, mají v Pascalu jednoznačný význam. Tato slova nelze jinak používat, než jak bude uvedeno v dalších oddilech:

and	array	begin	case	const	div
do	downto	else	end	file	for
forward	function	goto	if	in	label
mod	nil	not	of	or	packed
procedure	program	record	repeat	set	then
to	type	until	var	while	with

Poznámky: - v Turbo Pascalu jsou definována další rezervovaná slova:

absolute	external	inline	overlay	shl
shr	string	xor		

### 2.1.3 Identifikátory

Identifikátory jsou jména používaná k označení různých objektů, tj.: konstant, typů, proměnných, procedur a funkcí. Identifikátory začínají písmenem, za kterým může následovat libovolná kombinace písmen, číslic a znaku \_ (podtržení).

#### Příklady správných identifikátorů:

Alfa

Pocet

JmenoStudenta

Urok2Proc

Jmeno\_Zam

Příklady nesprávných identifikátorů:

STRIDA identifikátor musí začínat písmenem

Urok4% speciální symbol % není povolen

Jmeno-st speciální symbol - není povolen

Při programování je doporučováno používat identifikátory srozumitelné a snadno čitelné. Např. Identifikátor VymeraOrnePudy je čitelnější než identifikátor VYMERADORNEPUDY.

Poznámka: - délka identifikátoru bývá v některých implementacích omezena, např. na maximálně 8 znaků. V Turbo Pascalu omezení prakticky neexistuje, neboť identifikátor může mít délku až do max. délky jednoho řádku, tzn. 127 znaků.

2.1.4 Standardní identifikátory

V Pascalu je definována celá řada identifikátorů, které označují předdefinované typy dat, konstanty, proměnné, procedury a funkce. Teoreticky lze používat tyto identifikátory i k jinemu účelu, než pro který byly předem definovány. Novou definici těchto identifikátorů lze poměrně snadno zapříčinit různé omyly, proto tyto identifikátory používejme jen tak, jak byly definovány výrobcem kompilátoru.

Arctan	Boolean	Char	Chr	Cos	Dispose
EOF	EOLN	Exp	False	Get	Input
Integer	Ln	Maxint	New	Odd	Ord
Output	Pack	Page	Pred	Put	Read
Readln	Real	Reset	Rewrite	Round	Sin
Sqr	Sqrt	Succ	Text	True	Trunc
Unpack	Write	Writeln			

Poznámky: - Turbo Pascal obsahuje celou řadu dalších standardních identifikátorů, z nichž nejdůležitější jsou:

Assign	BlockRead	BlockWrite	Buflen	Byte
Chain	Close	ClrEOL	ClrScr	Con
Concat	Copy	Delay	Delete	Erase
Execute	Exit	FilePos	FileSize	FillChar
Flush	Frac	GotoXY	Halt	IDresult
Int	Kbd	KeyPressed	Length	Lst
Move	Pi	Pos	Ptr	Random
Randomize	Release	Rename	Seek	SizeOf
SeekEof	SeekEoln	Str	Upcase	Val

- naopak, Turbo Pascal neobsahuje standardní identifikátory definované v normě jazyka:

Get	Pack	Page	Put	Unpack
-----	------	------	-----	--------

#### 2.1.5 Oddělovače

Při zápisu programu se jednotlivé prvky jazyka oddělují nejméně jedním oddělovačem. Oddělovačem se rozumí mezera, konec řádku (klávesa Return) nebo poznámka.

## 2.1 Struktura programu

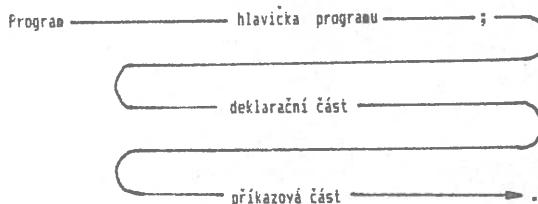
### 2.1.1 Obecná organizace programu

Programy v jazyce Pascal se sklídají ze tří částí: z hlavičky programu, deklarační části a z příkazové části:

- hlavička programu slouží především k označení programu;
- deklarační část se využívá k popisu objektů, se kterými se bude manipulovat v příkazové části;
- příkazová část popisuje akce a způsob, jakým budou vykonávány.

Aby mohly být uskutečněny akce, je nutné znát objekty, se kterými bude manipulováno. Proto deklarační část musí předcházet příkazové části.

Strukturu programu lze znázornit graficky. Diagram na obr. 2.1 začíná hlavičkou, za kterou se uvádí středník, pokračuje deklarační částí a končí příkazovou částí, za kterou musí být uvedena tečka.



Obr. 2.1 Syntaktický diagram programu v Pascalu

Tento diagram, ale i ostatní syntaktické diagramy v tomto učebním textu vycházejí z literatury [14] a jsou v podstatě jednodušší než běžně uváděné diagramy v jiných učebnicích.

Syntaktický diagram je orientovaný graf s jedním vstupem a jedním výstupem. Každý diagram má své jméno. Např. syntaktický diagram pro úplný program v Pascalu (viz obr. 2.1) je nazván jako "program". Jména diagramů mohou být použita i v jiných syntaktických diagramech.

Větvení čar v syntaktickém diagramu znázorňuje alternativní strukturu. Každá cesta definuje dovolenou strukturu. Iterativní struktura je znázorněna smyčkou (cyklem). Smyčkou lze projít jednou nebo vícekrát.

V syntaktických diagramech se používají jména a speciální symboly (viz 2.1.1). Aby se jasně rozlišila rezervovaná slova od ostatních jmen, je ostatní text v syntaktických diagramech uváděn drobným tiskem.

Syntaxe hlavičky programu se vyjádří pomocí syntaktického diagramu takto:

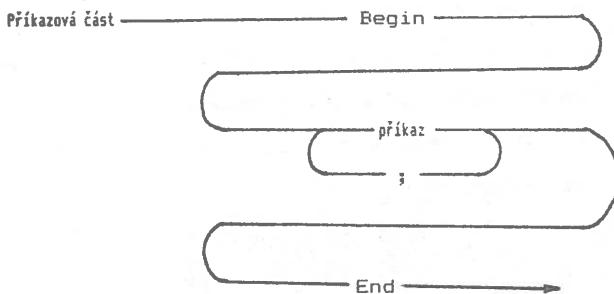


Obr. 2.2 Syntaktický diagram pro hlavičku programu

Je třeba poznamenat, že za rezervovaným slovem Program se obvykle uvádí jen jeden identifikátor. Další možnosti v zápisu hlavičky budou uvedeny v poznámkách na konci tohoto oddílu.

Výklad deklarační části je rozložen do celého textu (viz kapitola 3, oddíly 2.3 a 2.7).

Příkazovou část lze vyjádřit pomocí syntaktického diagramu takto:



Obr. 2.3 Syntaktický diagram příkazové části

#### Příklad

Nejjednodušší program, který je syntakticky správný, ale semanticky prázdný (neobsahuje popis žádné akce), je možno zapsat takto:

```
Program Prazdny;  
(* poznámka: deklarační část se uvádí před  
příkazovou částí *)
```

```
Begin (* zde začíná příkazová část *)
```

```
End.
```

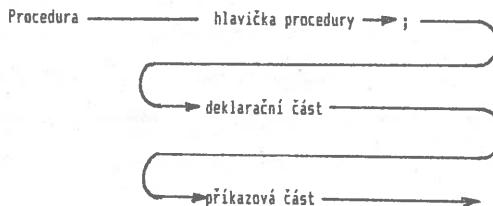
Každý program v Pascalu, i ten nejsložitější bude organizován stejným způsobem.

#### 2.2.2 Úvod do podprogramů

Procedura odpovídá tomu, co v jiných programovacích jazycích označujeme jako podprogram. Využívá se k definici složitějších akcí, přičemž popisuje jednodušší akce, z nichž je složena.

Abychom respektovali pravidlo "definujme dříve, než budeme používat", musíme v deklarační části uvést popis procedur, které budou používány v programu.

Procedura je vlastně jen zvláštní program (podprogram). Struktura procedury je podobná struktuře programu:



Obr. 2.4 Syntaktický diagram procedury

Následující příklad ukazuje strukturu programu, který obsahuje proceduru.

Příklad

```
Program Ukazka;  
(* deklarace objektu programu *)
```

```
Procedure P1;  
(* deklarace objektu P1 *)
```

```
Begin  
(* příkazy P1 *)  
End;
```

```
Begin  
(* příkazy programu *)  
End.
```

Procedura se používá pro popis akce, která může být složitá. Jestliže tato akce vytváří jednu hodnotu, potom používáme pojem funkce "function". Protože hodnota je vždy určitého typu, funkce musí být provázena popisem typu výsledku. Zatímco používání procedury se zabezpečuje pomocí speciálního příkazu (tzv. příkazu procedury), funkce, která vždy přináší výsledek, se vyvolává v aritmetickém výrazu.

V souladu s jinými autory se pro společné označení

deklarační a příkazové části používá termín blok.

Funkce i procedury budou podrobně vysvětleny ve čtvrté kapitole.

Poznámka: - ve standardní verzi Pascalu hlavička programu může obsahovat další identifikátory. Tyto identifikátory definují logická vstupní a výstupní zařízení, např. Input, Output. Uvádění těchto identifikátorů v Turbo Pascalu je možné, ale není nutné.

### 2.3. Jednoduché deklarace

#### 2.3.1 Předdefinované typy dat

Při řešení jakéhokoliv problému zjišťujeme, že údaje mohou být různé povahy, např. reálná čísla, celá čísla, alfanumerické znaky apod.

V programovacích jazycích pojem povahy dat hraje důležitou roli, umožňuje nejen prezentovat jasným způsobem manipulované objekty, ale také zajistit účinnou kontrolu programu jak při komplaci, tak při zpracování. V tomto kontextu je třeba chápát typy dat.

Jak bude uvedeno v třetí kapitole, lze konstruovat typy dat velmi složité. Ale ať se jedná o typ jakkoli složitý, je vždy vytvářen na základě jednoduchých typů. Jednoduchý typ je charakterizován množinou přípustných hodnot. Existují dvě kategorie jednoduchých typů:

- typy předdefinované, tzn., že tyto typy dat jsou definovány již výrobcem kompilátoru. Každý kompilátor Pascalu by podle normy měl obsahovat minimálně tyto čtyři typy: typ Integer, typ Real, typ Boolean a typ Char. Tyto typy budou popsány v tomto oddíle;

- typy jednoduché, definované programátorem. Mezi tyto typy patří výčtový typ a typ interval. Tyto dva typy budou vysvětleny v oddílech 3.2.1 a 3.2.2.

Protože typ definuje nejen množinu hodnot objektu tohoto typu, ale i operace vykonatelné s tímto objektem, tak jsou v dalším výkladu pro každý předdefinovaný typ uvedeny přípustné operace.

#### 2.3.1.1 Číselné typy dat

Existují dva předdefinované typy dat, které umožňují manipulaci s objekty, jejichž hodnotami jsou buď čísla celá nebo čísla reálná. Jsou to typy Integer a Real.

##### Typ Integer

Objekt typu Integer může nabývat hodnot celých čísel. V operační paměti počítače objekt tohoto typu je zobrazen jako číslo v pevné řádové čárce. V závislosti na konkrétních technických vlastnostech počítače a na způsobu řešení určité implementace Pascalu se setkáváme s různými délkami typu Integer; nejčastěji typ Integer zabírá délku 2B. Potom celé číslo je limitováno od -32768 do 32767.

Horní hranice je označována jako MaxInt (MaxInt = předdefinovaná konstanta).

Hodnota celého čísla se zapisuje v programu ve formě konstanty. Zapisuje se obvyklým způsobem pomocí posloupnosti desítkových číslic, kterým může předcházet znamenka.

Příklad

10            +5            -127

Základní operace pro typ Integer jsou:

sčítání	+
odčítání	-
násobení	*
celočíselné dělení	DIV
zbytek po dělení	MOD

Operace MOD (modulo) je definována jako:

$X \text{ MOD } Y = X - ((X \text{ DIV } Y) * Y)$  a přináší zbytek celočíselného dělení čísla X číslem Y, když X i Y je kladné nebo rovno nule.

Příklad

$$4 \text{ MOD } 3 = 1$$

$$12 \text{ MOD } 5 = 2$$

- Poznámky:
- každá výše uvedená operace přináší celočíselný výsledek;
  - přetečení v aritmetických operacích není určováno. Výsledky operací musí být v přípustném intervalu. Např. výraz  $1000 * 100 \text{ DIV } 50$  nedá výsledek 2000, neboť násobení způsobi přetečení;
  - operaci MOD pro záporné operandy je vhodné předem ověřit;
  - existují standardní funkce, které při použití na

celočíselné hodnoty dávají celočíselný výsledek:

Abs ( X )	absolutní hodnota X
Sqr ( X )	druhá mocnina X
Pred ( X )	předchůdce X, tj. X-1
Succ ( X )	následník X, tj. X+1;

- v Turbo Pascalu je definován typ `Byte`, který je podmnožinou typu `Integer`. Zabírá 1 B paměti a má rozsah 0..255. Pro typ `Byte` platí stejná pravidla jako pro typ `Integer`. Používání typu `Byte` je často výhodné, neboť umožnuje šetřit operační i vnější paměť.

#### Typ Real

Objekt typu `Real` může nabývat jakýchkoli numerických hodnot. V operační paměti počítače objekt tohoto typu je zobrazen jako číslo v pohyblivé řádové čárce. Tyto objekty jsou tedy zobrazovány v jednom útvaru jako dvě čísla, tj. exponent a mantisa. V závislosti na konkrétních technických vlastnostech počítače a na způsobu řešení určité implementace Pascalu tento útvar může mít délku 4, 6, 8 i více B. Celkový počet bitů mantisy obvykle určuje přesnost objektů typu `Real`.

Reálné číslo se v programu zapisuje s desetinnou tečkou.

#### Příklad

3.141      151.0      -2.2

Je přípustný též zápis v semilogaritmickém tvaru:

3E-7	je ekvivalentní	0.0000003
-5.29E3	je ekvivalentní	5290

Pro typ Real jsou definovány tyto operace:

sčítání	+
odčítání	-
násobení	*
dělení	/

Tyto čtyři operace dávají výsledek typu Real. Tyto operace lze použít i tehdy, je-li jeden operand typu Integer, výsledek operace je potom typu Real.

Operace reálného dělení ( / ) může být použita i pro operandy typu Integer, výsledek operace je potom typu Real.

- Poznámky:
- v Turbo Pascalu množina reálných čísel je od  $1E-38$  do  $1E+38$ . Každé číslo zabírá 6 B paměti a jeho přesnost dosahuje 11 dekadických číslic;
  - přetečení při aritmetických operacích s typy Real zastavuje zpracování programu a na obrazovce se objevuje chybové hlášení. Podtečení způsobí, že výsledek bude roven nule;
  - existují dvě standardní funkce, které při použití na typ Real dávají výsledek typu Integer. Jsou to:

Trunc ( X ) celočíselná část X

např. Trunc ( 5.67 ) = 5

Round ( X ) zaokrouhlování X

Jestliže  $X \geq 0$

Round ( X ) = Trunc(X+0.5)

Jestliže  $X < 0$

Round ( X ) = Trunc(X-0.5)

### Příklad

Round (5.67) = 6

Round (3.14) = 3

Round (-5.67) = -6

Round (-3.14) = -3

- existují dvě standardní funkce, které při použití na typy Real dávají výsledky typu Real.

J'sou to;

Abs ( x )               absolutní hodnota x

Sqr ( X )            X \* X (druhá mocnina):

- existuje šest standardních funkcí, které dávají výsledek typu Real, i když jsou použity na jakýkoli typ:

Sin ( x )

Cos ( x )

ArcTan ( x )

X je v radiánech,  
výsledek v radiánech

Ln ( x )

**Exp** ( x )

Sqrt ( x )

#### **BĚROZENÝ LEGARITMUS**

#### exponenciální funkce

DEUBÁ, edoceania

- tipo Real má tato amazônico -

- funkce Pred a Succ nemohou být použity

- proměnná typu Real nemůže být použita pro indexování polí

- proměnné typu Real nemohou být použity pro řízení příkazů For a Case.

- definování intervalů není dovoleno

- proměnná typu Real se nesmí použít při definování typu množina

### 2.3.1.2 Symbolické typy dat

Existují dva předdefinované typy, které umožňují definovat buď množinu logických hodnot a nebo množinu hodnot, které jsou reprezentovány znaky. Tyto typy se nazývají "Boolean" a "Char".

#### Typ Boolean

Typ Boolean specifikuje množinu logických hodnot, které se označují předdefinovanými identifikátory True a False.

Promenna typu Boolean zabírá 1 B paměti.

Pro hodnoty typu Boolean jsou definovány operace, které přinášejí výsledky typu Boolean. Jsou to:

logický součin AND

logický součet OR

logická negace NOT

Poznámky: - je definováno, že hodnota False je menší než True;

- existují relační operace, které mohou být použity pro všechny jednoduché typy, včetně typu Boolean. Tyto operace dávají výsledek typu Boolean. Jsou to:

rovno =

menší než <

větší než >

menší nebo rovno <=

větší nebo rovno >=

nerovno <> ;

- existuje standardní funkce, která při použití na

typ Integer prináší výsledek typu Boolean. Je to funkce:

Odd ( X ) ,

která dává výsledek True, je-li X liché. Je-li X sudé, potom výsledek je False. Tento test na liché číslo je ekvivalentní  $\text{Abs}(X) \bmod 2 = 1$ .

### Typ Char

Typ Char definuje uspořádanou množinu hodnot, které jsou reprezentovány znaky; definuje tedy abecedu, která umožňuje komunikovat s vnějším světem (vstupy dat a výstupy výsledků na periferních zařízeních).

Konstanta typu Char se v programu zapisuje jako znak uzavřený v apostrofech. Např.:

'A'      ';'      'x'

Poznámky: - v Turbo Pascalu hodnota typu Char je jeden znak z množiny kódu ASCII. Znaky jsou seřazeny podle hodnot jejich ASCII kódu, např. 'A' < 'B'.  
- Proměnná typu Char zabírá 1 B paměti;  
- existují dvě standardní funkce, které umožňují přechod z celočíselné hodnoty na hodnotu typu Char a naopak. (Celočíselná hodnota určitého znaku je jen ordinální (pořadové) číslo tohoto znaku v množině znaků).

Ord ( C ) dává ordinální číslo znaku, jehož hodnota je dána znakem C. Výsledkem je taková celočíselná hodnota, že platí:

$0 \leq \text{Ord} (C) \leq \text{počet znaků} - 1$

`Chr (X)` přináší znak, jehož ordinální číslo je  
dáno `X`. `X` je typu Integer.

### Příklady

`Chr (65)` přináší znak 'A'

`Ord ('B')` dává ordinální číslo 66

(Porovnejte uvedené příklady s tabulkou kódu ASCII.)

Funkce `Ord` a `Char` jsou inverzní:

`Ord (Chr (Y)) = Y`

`Chr (Ord (X)) = X ;`

- standardní funkce `Pred` a `Succ` nejsou definovány,  
jestliže se použijí pro první a poslední znak z  
množiny.

Přehled standardních funkcí použitelných pro  
předdefinované typy je uveden v následující tabulce:

funkce	typ operandů	typ výsledku
Abs	Integer Real	Integer Real
Sqr	Integer Real	Integer Real
Pred/Succ	Integer Char	Integer Char
Trunc/Round	Real	Integer
Sin/Cos/ArcTan Ln/Exp/Sqrt	Real/Integer	Real

funkce	typ operandů	typ výsledku
Odd	Integer	Boolean
Chr	Integer (0 až 127)	Char
Ord	Char	Integer

Tab. 2.1 Přehled standardních funkcí

### 2.3.2 Deklarace proměnných

Aby v programu mohl existovat určitý objekt, je třeba jej deklarovat. V Pascalu deklarace proměnných se uvádějí v úseku deklarací proměnných, který začíná rezervovaným slovem "Var". Deklarace proměnné se skládá z identifikátoru (jména proměnné), za kterým následuje ":" (dvojtečka), typ proměnné a oddělovač ";" (středník), který ukončuje konec deklarace.

#### Příklad:

```
Program Deklarace;
(* deklarace proměnných *)

var
  I : Integer;    (* proměnná I je celočíselného typu *)
  X : Real;       (* proměnná X je reálného typu *)
  OK : Boolean;   (* proměnná OK je typu Boolean *)
  Znak,
  Kod : Char      (* proměnné Znak a Kod jsou typu Char *)

Begin
  (* příkazová část *)
End.
```

Jestliže několik proměnných je stejného typu, je možné vytvořit jednu společnou deklaraci, přičemž jména proměnných se oddělují čárkami. Tzn., že deklarace

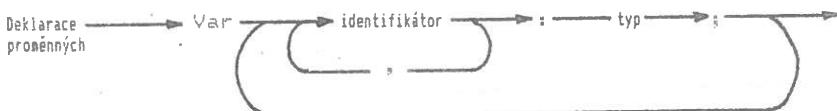
Znak, Kod : Char;

Znak : Char;

Kod : Char;

jsou dvě možné formy zápisu, které jsou rovnocenné.

Syntaktický diagram na obr. 2. 5 znázorňuje úsek deklarace proměnných.



Obr. 2.5 Syntaktický diagram úseku deklarace proměnných

Poznamenejme, že slovo Var se v deklaraci proměnných objevuje jen jednou.

### 2.3.3 Deklarace konstant

Pro zvýšení čitelnosti programů a zvláště pro usnadnění údržby programu Pascal nabízí možnost používání konstant, které jsou přiřazovány symbolickým jménům. Deklarace konstant se uskutečňuje v úseku deklarace konstant, který začíná rezervovaným slovem "Const". Syntaxe tohoto úseku je tato:



Obr. 2.6 Syntaktický diagram úseku deklarace konstant

Konstanta musí být předdeklarovaného typu. Pokud se v programu použije úsek deklarace konstant "Const", tak musí předcházet úseku deklaraci proměnných.

#### Příklad

```
Const
  Jedna = 1;      (* Jedna bude mít hodnotu 1 *)
  Max = 100;     (* Max bude mít hodnotu 100 *)
  Pravda = True;  (* Pravda bude mít hodnotu True *)
```

Poznámky: - v Turbo Pascalu jsou předdefinovány čtyři konstanty, které můžeme využívat, aniž bychom je deklarovali:

Pi	Typ Real	(3.1415926536E+00)
False	Typ Boolean	(nepravda)
True	Typ Boolean	(pravda)
Maxint	Typ Integer	(32767)

- v úseku deklarace konstant lze též definovat tzv. typové konstanty (viz oddíl 3.9).

#### 2.4 Výrazy

Základní příkaz, který umožnuje změnit či přiřadit hodnotu proměnné, se nazývá přiřazovací příkaz. Hodnota, která je přiřazena proměnné, je vždy výsledkem více či méně

složitého výpočtu, který je popsán výrazem. V Pascalu se rozlišují dva typy výrazů: aritmetické a logické.

#### 2.4.1 Aritmetické výrazy

Aritmetický výraz popisuje výpočet, který dává číselný výsledek. Výraz se skládá z operandů (proměnné, konstanty a jména funkcí) a z operátorů, které byly uvedeny pro typy Integer a Real.

#### Příklady

Cena + Dan	potřebné deklarace:
Počet Mod 2	Const
Pi * Sqr (Polomer)	Dan = 10.0;
	Var
	Cena : Real;
	Pocet, Polomer : Integer;

Poznámky: - priority

- pokud výraz obsahuje několik operátorů, vzniká problém, v jakém pořadí operace budou vykonány. Tak jako ve většině programovacích jazyků existují tzv. pravidla priority. Operátory násobení-dělení mají vyšší prioritu než operátory sčítání-odčítání,
- pokud pravidla priority nevyhovují je možno použít závorky,
- pokud několik operátorů má stejnou prioritu, tak jsou vykonávány v pořadí zleva doprava.

#### Příklady

$A + B * C$  je stejně jako  $A + (B * C)$

aby se uskutečnilo dříve sčítání,

je třeba zapsat  $(A + B) * C$

W \* X - Y \* Z je ekvivalentní  $(W * X) - (Y * Z)$   
X \* Y / Z je ekvivalentní  $(X * Y) / Z$   
X Mod Y \* Z je ekvivalentní  $(X \text{ mod } Y) * Z$   
-A + B je ekvivalentní  $(-A) + B$ ;

- typy

- typ výsledku aritmetického výrazu závisí jak na použité operaci, tak na typu operandu.  
Využívají se tato pravidla:
  - výsledek Integer: Jestliže operandy jsou celočíselné a není použito reálné dělení (/).
  - výsledek Real: Jestliže jeden z operátorů (+ - \* ) je reálný. Jestliže se použije reálné dělení (/).

#### 2.4.2 Logické výrazy

Logický výraz popisuje výpočet, který dává výsledek typu Boolean.

V logických výrazech se mohou použít jen operátory, které byly popsány v oddíle 2.3.1.2 (Not, Or, And a relační operátory).

#### Příklad

Výsledek výrazu  $(9 < 8) \text{ And } (10 > 7)$   
je False.

Poznámky: - priority

- pokud se nepoužijí závorky, tak platí toto sestupné pořadí: Not, And, Or a relační

### **operatory.**

### Příklad

Předpokládejme, že proměnné X, Y, Z jsou typu Integer a proměnné A, B typu Boolean

$X < Y \text{ Or } X < Z$  je ekvivalentní  $(X < Y) \text{ Or } (X < Z)$

$\text{Not } A \text{ Or } B$  je ekvivalentní  $(\text{Not } A) \text{ Or } B$

- typy
    - logický výraz je vždy typu Boolean.

## 2.5. Jednoduché příkazy

Příkazy se dělí na dvě velké skupiny: na příkazy jednoduché a na příkazy strukturované. Jednoduché příkazy jsou takové příkazy, které se neodvolávají na jiné příkazy. V Pascalu mezi ně je možno zařadit přiřazovací příkaz, příkaz procedury, prázdný příkaz a příkaz skoku. Příkazy vstupu/výstupu jsou zvláštním příkazem procedury. Příkaz skoku bude vysvětlen až v oddíle 2.8.

#### 2.5.1 Přiřazovací příkaz

Přiřazovací příkaz umožnuje přiřadit vypočtenou hodnotu ve výrazu do zadанé proměnné:



Obr. 2.7 Syntaktický diagram přiřazovacího příkazu

Výraz popisuje způsob výpočtu zjištované hodnoty. Typ hodnoty musí být kompatibilní s typem proměnné, která přijímá hodnotu. Proměnné typu Real může být přiřazena hodnota typu Integer.

### Příklady

Předpokládejme tyto deklarace:

```
var  
  D, F : Boolean;  
  I, J, K : Integer;  
  C : Char;  
  S : Real;
```

Správné přiřazovací příkazy:

```
S := 0;  
D := I > J;  
F := Not D;  
J := I + K Mod 5;  
C := 'g';  
S := Pi * 2 * I;  
S := J;
```

Nesprávné přiřazovací příkazy:

```
I := 5.6;  
F := 0;  
C := D;  
J := S;  
I := K + C;
```

Nyní již můžeme zapsat kompletní program, ke kterému přidáme pouze jeden dosud neznámý příkaz (Writeln) pro vytištění výsledku.

### Příklad

Vypočteme plochu kruhu, pokud jeho polomér je 50 cm.

```
Program PlochaKruhu;  
var  
  Plocha : Real;  
  Polomer : Real;  
Begin  
  Polomer := 50.0;  
  Plocha := Pi * Polomer * Polomer;  
  Writeln (Plocha)  
End.
```

### 2.5.2 Příkaz procedury

V oddíle 2.2.2 bylo ukázáno, že je možno zkonstruovat část programu pomocí deklarace procedury. Pro zajištění akce, která je vyjádřena v proceduře, používáme tzv. příkaz procedury. Tento příkaz se zapisuje pomocí identifikátoru procedury.

#### Příklad

```
Program Akce;
Procedure AkceZ;
Begin
    Writeln ('V y s l e d k y');
End;
Begin
    AkceZ;
End.
```

Při nalezení příkazu AkceZ se vykonají všechny příkazy obsažené v proceduře tak, jako kdyby byly zapsány místo příkazu procedury.

Poznámky:

- procedura musí být deklarována dříve než bude použita;
- předdefinované procedury (např. Writeln) mohou být použity, aniž budou deklarovány v programu.

### 2.5.3 Vstupy/výstupy

Oba dva výše uvedené programy způsobují výstup výsledků. Proto byl použit příkaz pro vyvolání předdefinované procedury Writeln (zkratka z "write line"). Nyní budou vysvětleny základní předdefinované procedury vstupu/výstupu pro čtení vstupních dat a pro zobrazení výsledků na obrazovce.

## VÝSTUPY

Výstupní procedura Write má tuto formu zápisu:

Write ( p1, p2, ... , pn )

kde p1, p2, ... , pn jsou parametry, které mohou nabývat jedné z těchto forem:

e

e:w

e:w.d

Výraz e představuje výstupní hodnotu, která musí být typu Integer, Real, Boolean, Char, případně to může být řetězec znaků uzavřený v apostrofech. w je nepovinný parametr, který určuje minimální délku (počet pozic) výstupní hodnoty. Výraz d je také nepovinný parametr, používá se jen u reálných hodnot a potom udává počet pozic za desetinnou tečkou.

### Příklady

Výstup hodnoty typu Integer,  
proměnná I1 má hodnotu 133.

Write ( I1 );	123456789	= pozice
Write ( I1:8 );	133	
Write ( I1:2 );	133	

Výstup hodnoty typu Real,  
proměnná R1 má hodnotu 98.133.

Write ( R1:8:2 );	123456789	= pozice
Write ( R1:3:1 );	98.13	
Write ( R1:9:4 );	98.1330	
Write ( R1:9 );	9.813E+01	
Write ( R1 );	9.8133000000E+01	

Výstup hodnoty typu Char,  
proměnná Znak má hodnotu 'a'.

```
123456789      = pozice  
Write ( Znak );      a  
Write ( Znak:5 );      a
```

Výstup řetězce znaků, např. 'Vymera'.

```
123456789      = pozice  
Write ( 'Vymera' );      Vymera  
Write ( 'Vymera':10 );      Vymera  
Write ( 'Vymera':3 );      Vymera
```

Výstupní procedura Writeln je téměř identická s procedurou Write, jediný rozdíl spočívá v tom, že po výstupu hodnot prostřednictvím procedury Writeln se za výstupními hodnotami uskuteční posloupnost řídicích znaků CR/LF (návrat vozíku a posun na další řádek). Kurzor se posouvá na začátek dalšího řádku.

Poznámky:

- při práci v Turbo Pascalu výstupní procedury jsou velmi rychle osvojitelné, i když je nutno poznamenat, že procedury Write a Writeln poskytují ještě další možnosti výstupu, např. tištění výsledku na tiskárnu (viz oddíl 3.7.2);
- procedura Writeln bez parametru způsobí výstup prázdné řádky. Kurzor se přesune na začátek dalšího řádku.

## VSTUPY

Vstupní procedura Read má tuto formu zápisu:

```
Read (v1, ... , vn )
```

kde, v1, ... , vn jsou identifikátory pro proměnné typu Integer, Real a Char.

## Příklady

Předpokládejme deklaraci:

```
Var  
    I1, I2 : Integer;  
    Polomer: Real;
```

Potom můžeme zapsat např:

```
Read (I1);
```

Program bude připraven číst celé číslo.

Po napsání čísla musíme pro ukončení  
čtení stisknout klávesu Return.

Pro snadnější práci s počítačem je vhodné,  
když požadavky na vstup zapisujeme takto:

```
Write ('Zadej polomer:');  
Readln (Polomer);
```

Vstupní procedura Readln je téměř identická s  
procedurou Read, jediný rozdíl spočívá v tom, že po vstupu  
hodnot prostřednictvím procedury Readln se za vstupními  
hodnotami uskuteční posloupnost řídicích znaků CR/LF. Kurzor  
se přesouvá na začátek dalšího řádku.

Poznámky:

- v interaktivním režimu práce s mikropočítačem  
vstupní procedury jsou snadno osvojitelné.
- Procedury Read a Readln se používají např. pro  
čtení z disket. (viz oddíl 3.7.2);
- pro práci s obrazovkou je v Turbo Pascalu  
předdefinováno několik praktických procedur.

Zatím uvedeme tyto dvě:

ClrScr,

která zabezpečuje výmaz obrazovky a nastavení kurzoru do pozice 1,1

GotoXY(Xpoz,Ypoz),

která přesouvá kurzor na obrazovce do pozice, která je určena výrazy Xpoz (vodorovná hodnota, tj. pozice v řádku) a Ypoz (svislá hodnota, tj. pozice v sloupci). Levá horní pozice na obrazovce má hodnotu (1,1).

Na základě tohoto výkladu můžeme příklad na výpočet plochy kruhu z oddílu 2.5.1 upravit takto:

```
Program PlochaKruhu;
var Plocha : Real;
    R : Real;

Begin
    ClrScr;
    GotoXY(10,2);
    Write(' Program pro vypocet plochy kruhu');
    GotoXY(5,4); Write ('Zadej polomer:');
    Read ( R );
    Plocha := Pi * R * R;
    GotoXY(5,5);
    Write('Pro R ',R:6:1,' Plocha =',Plocha:9:1);
    GotoXY(10,7);
    Write (' Konec programu ')
End.
```

Po zpracování programu bude na obrazovce tento stav:

Program pro vypocet plochy kruhu

Zadej polomer:2  
Pro R 2.0 Plocha = 12.6

Konec programu

Obr. 2.8 Výstup programu PlochaKruhu

#### 2.5.4 Prázdný příkaz

Prázdný příkaz je příkaz, který neobsahuje žádné symboly a také nemá žádný vliv na zpracování. Může se objevit tam, kde syntaxe jazyka Pascal vyžaduje příkaz, ale přitom se neprovádí žádná akce.

### 2.6. Základní strukturované příkazy

Mnoha teoretickými pracemi bylo dokázáno, že existují v podstatě jen tři základní struktury, ze kterých se skladají algoritmy. Tyto struktury umožňují vyjádření pořadí, podle kterého budou rozdílné akce vytvářející program, seřazeny. Jsou to tyto struktury:

- sekvenční struktura označovaná iako posloupnost příkazů;
- alternativní struktura, označovaná jako podmíněná struktura nebo zkráceně jen jako větvení a
- iterativní struktura, označovaná jen jako opakování nebo cyklus.

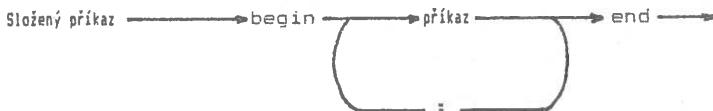
Pro zápis těchto tří struktur jsou v jazyku Pascal k disposici především tyto tři příkazy: složený příkaz, příkaz If a příkaz While.

#### 2.6.1 Složený příkaz

Jednoduchá akce je taková akce, která může být vyjádřena pomocí jednoho příkazu, jako je např. přiřazovací příkaz.

Jestliže akce vyžaduje nejméně dva jednoduché příkazy, potom můžeme využít mechanismus, který umožnuje seskupení rozdílných příkazů do jednoho celku. Toto seskupení se

nazývá složený příkaz a lze jej vyjádřit tímto diagramem:

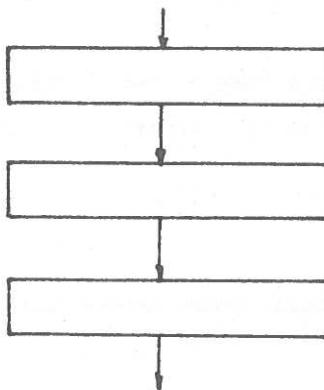


Obr. 2.9 Syntaktický diagram pro složený příkaz

příkaz může označovat jakýkoliv příkaz definovaný v Pascalu (tedy jak jednoduchý tak strukturovaný).

Ve složeném příkazu skupina příkazů je ohrazena rezervovanými slovy begin a end. Je možno poznamenat, že celý program je vlastně zapsán pomocí složeného příkazu.

Pomocí vývojových diagramů se složený příkaz dá vyjádřit takto:



Obr. 2.10 Vývojový diagram sekvenční struktury

Složený příkaz umožnuje vyjadřovat i velmi složité zpracování pomocí jednoduchých dílčích částí. Definice příkazu je totiž rekursivní, tzn., že příkaz může obsahovat soubor příkazů seskupených do jiných složených příkazů. Tzn., že v obr. 2.10 každá znáčka pro zpracování může být reprezentována libovolnou strukturou.

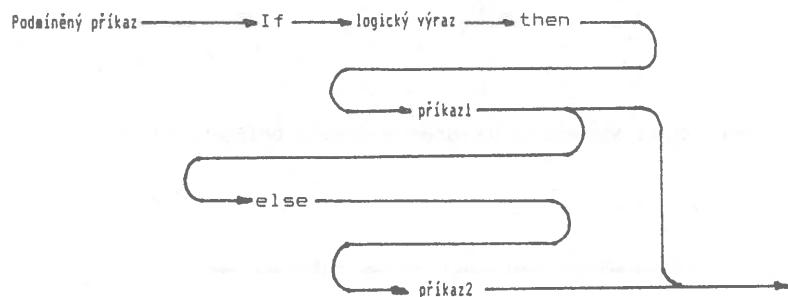
Poznámky: - středník je oddělovač příkazu a ne znaménko konce příkazu, proto zapisujeme:

```
begin  
    A := B;  
    C := D  
end
```

- posloupnost dvou středníků definuje prázdný příkaz stejně jako posloupnost ";end".

#### 2.6.2 Příkaz If

Podmíněný příkaz If umožnuje alternativní výběr akce v závislosti na podmínce. Tento příkaz má tuto syntaxi:



Obr. 2.11 Syntaktický diagram příkazu If

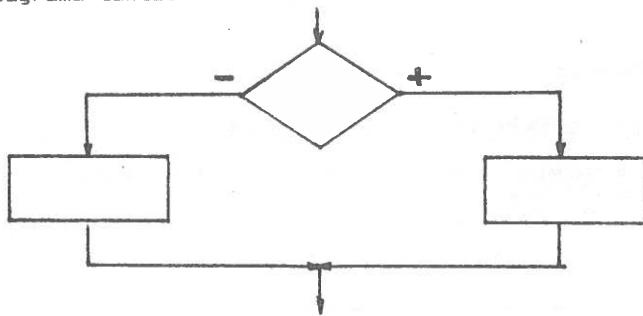
Jak vyplývá z diagramu existují dvě možné formy tohoto příkazu:

1. If → logický výraz → then → příkaz1 → else → příkaz2 →

Obr. 2.12 Syntaktický diagram úplného příkazu If

Jestliže hodnota logického výrazu je True, potom se vykoná

příkaz1; když ne, tak se vykoná příkaz2. Po vykonání jednoho nebo druhého příkazu program pokračuje normálně. Tuto situaci lze znázornit pomocí vývojového diagramu takto:

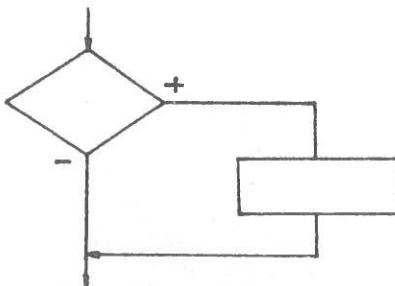


Obr. 2.13 Vývojový diagram úplného příkazu If

2. If → logický výraz → then → příkaz1 →

Obr. 2.14 Syntaktický diagram neúplného příkazu If

Jestliže hodnota logického výrazu je True, potom se vykoná příkaz; když ne, tak program bude pokračovat normálně dál. Tuto situaci lze znázornit pomocí vývojového diagramu takto:



Obr. 2.15 Vývojový diagram neúplného příkazu If

#### Příklad

```
Program Prodavac;
var
    Cena, Placeno, Vratit : Integer;

Begin
    Read ( Cena, Placeno );
    Vratit := Placeno - Cena;
    If Vratit < 0 then
        Write(' Bylo zaplaceno malo')
    else
        Write('Je trsba vratit:',Vratit)
End.
```

V tomto příkladu je znázorněn běžný způsob rozhodování prodavače. Příkazem If se určuje obě možné alternativy. Situace, kdy se vynechává část else, bude znázorněna v dalším příkladu. Potom pomocí příkazu if lze vybírat pouze akce, když je logický výraz True.

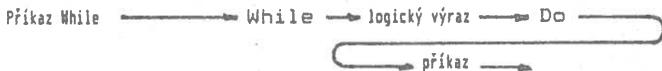
### Příklad

```
Program Maximum;
var
  Cislo1, Cislo2, Maximum : Integer;

Begin
  Read (Cislo1, Cislo2);
  Maximum := Cislo1;
  If Cislo1 < Cislo2 then
    Maximum := Cislo2;
  Write ('Maximum = ',Maximum)
End.
```

### 2.6.3 Příkaz While

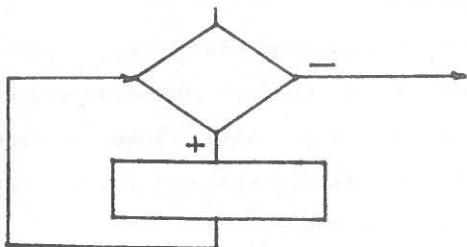
Příkaz cyklu While umožňuje opakovat zpracování tak dlouho, pokud podmínka je správná. Syntaxe tohoto příkazu je tato:



Obr. 2.16 Syntaktický diagram příkazu While

Jestliže podmínka, definovaná logickým výrazem, není splněna již od začátku, potom se příkaz nevykonává.

Vývojový diagram pro příkaz While lze zapsat takto:



Obr. 2.17 Vývojový diagram příkazu While

Příkaz je vykonáván tak dlouho, pokud hodnota logického výrazu zůstává rovna True. Je třeba poznamenat, že pokud logický výraz je False již od začátku vykonávání, potom se příkaz nevykoná ani jedenkrát. V rámci cyklu se musí vyskytnout akce, která umožní změnit prvek v logickém výrazu, aby se zabránilo nekonečnému cyklu.

Příklad nekonečného cyklu:

```
X := 0;  
While X = 0 do  
    A := B;
```

Příklad ilustruje použití příkazu While.

```
Program Prumer;  
var  
    Cislo, Suma, Pr : Real;  
    Pocet : Integer;  
  
Begin  
    ClrScr;  
    GotoXY(5,1); Write ('Program na vypocet prumeru');  
    GotoXY(2,4);  
    Writeln ('Zadavej cisla, zaporne cislo => konec');  
    Suma := 0.0; Pocet := 0;  
    Readln ( Cislo );  
    While Cislo >= 0.0 do  
        begin  
            Suma := Suma + Cislo;  
            Pocet := Pocet + 1;  
            Readln ( Cislo )  
        end; (* konec prikazu While *)  
    If Pocet > 0 then  
        begin  
            Pr := Suma / Pocet;  
            Write (' Prumer = ', Pr:8:2)  
        end  
    else  
        Write('Nebyly zadany vstupni hodnoty');  
End.
```

Program Prumer poskytuje např. tyto výsledky:

Program na vypocet prumeru

```
Zadavej cisla, zaporne cislo => konec  
6  
8  
4  
-1
```

Prumer = 6.00

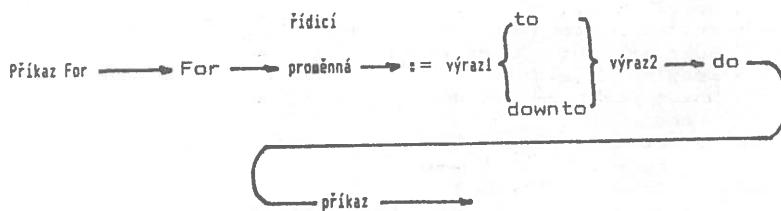
Obr. 2.18 Výstup programu Průměr

## 2.7. Ostatní strukturované příkazy

Existují další příkazy, které již nejsou nezbytné, ale jsou velmi praktické pro zápis některých algoritmů. Patří sem další dva příkazy cyklu (For a Repeat) a podmíněný příkaz Case.

### 2.7.1 Příkaz For

Tento příkaz umožňuje vykonávání příkazu pro všechny hodnoty řídící proměnné od hodnoty počáteční až do koncové, přičemž nastavá zvyšování nebo zmenšování řídící proměnné o jedničku v každé iteraci.



Obr. 2.19 Syntaktický diagram příkazu For

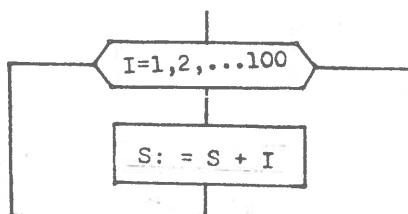
Příkaz je opakován pro všechny následující hodnoty z uzavřeného intervalu ohraničeného hodnotami výraz1 a výraz2. Tyto dva výrazy musí být stejněho ordinálního typu (předdefinované typy vyjma typu Real jsou typy ordinální) a kompatibilní s typem proměnné.

Příklad

```
Program Cyklus;
var
  S,I : Integer;

Begin
  S := 0;
  For I := 1 to 100 do
    S := S + I;
  Writeln('Součet cisel 1,...,100 = ', S)
End.
```

Příkaz For lze pomocí značek vývojových diagramů zapsat takto:



Obr. 2.20 Vývojový diagram pro příkaz For - příklad Cyklus

Poznámky: - pojem krok v jazyku Pascal není explicitně zaveden, na rozdíl od mnoha jiných programovacích jazyků. Krok je implicitně definován jako Succ (běžná hodnota);

Příklad

```
For Vel := 'A' to 'Z' do ...
For Log := False to True do ...
For Ses := 'z' downto 'a' do ...
```

Proměnná *Vel* nabývá nejdříve hodnoty 'A', potom 'B', atd. až do 'Z'; proměnná *Log* nabývá hodnoty False a potom True; proměnná *Ses* nabývá sestupně hodnot 'z', 'y', ... , 'b', 'a' :

- konstrukce For *V* := *E1* to *E2* do *Akce*

je ekvivalentní:

```
begin
  T1 := E1;
  T2 := E2;
  If T1 <= T2 then
    begin
      V := T1;
      Akce;
      While V <> T2 do
        begin
          V := Succ (V);
          Akce
        end
    end
  end
```

Toto dokazuje, že:

- koncová hodnota se určuje před cyklem. Všechny změny koncové hodnoty uvnitř cyklu nemají vliv na počet opakování Akce uvnitř cyklu,
- jestliže hodnota koncová je nižší než počáteční, potom Akce se vůbec nevykoná;
- řídící proměnná cyklu musí být deklarována v deklarační části úrovně, kde se příkaz For objevuje. Tzn. jako lokální proměnná, jestliže příkaz For se nachází v příkazové části procedury nebo funkce nebo jako globální proměnná, jestliže příkaz For se nachází v hlavním programu;
- řídící proměnná cyklu nesmí být změněna uvnitř cyklu. Tzn., že výsledek následujícího příkladu

je nesprávný:

```
For I := 1 to 10 do
begin
  I := I + 1;           chyba
  S := S + 1
end
```

- hodnota řídící proměnné je na konci cyklu neurčena (kromě předčasného východu pomocí příkazu GoTo). Tzn., že po vykonání cyklu:

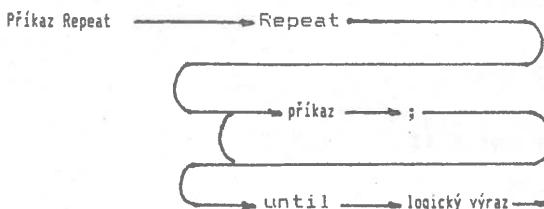
```
For I := 1 to 10 do Akce;
```

příkaz Writeln (I); zajišťuje výstup neurčené hodnoty;

- typ řídící proměnné musí být kompatibilní s typy počátečních a koncových hodnot. Musí být jednoduchého typu, ale ne typu Real.

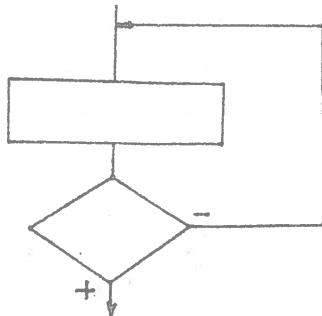
#### 2.7.2 Příkaz Repeat

Příkaz Repeat umožňuje opakovat akci nejméně jednou, případně vícekrát, tzn. až do splnění podmínky. Jeho syntaxi lze vyjádřit takto:



Obr. 2.21 Syntaktický diagram příkazu Repeat

Všechny příkazy obsažené mezi rezervovanými slovy Repeat a Until jsou opakovány až do té doby, pokud logický výraz nebude mít hodnotu True. Vývojový diagram pro příkaz Repeat lze zapsat takto:



Obr. 2.22 Vývojový diagram příkazu Repeat

#### Příklad

```
Program Prumer;
var
    Cislo, Suma, Pr : Real;
    Pocet : Integer;

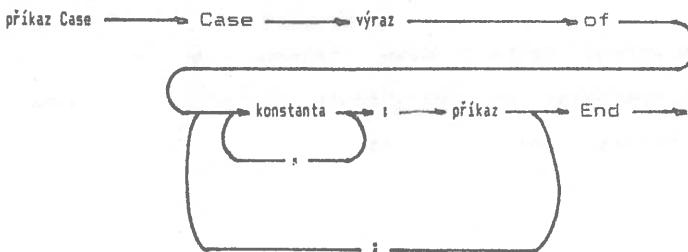
Begin
    ClrScr;
    GotoXY(5,1); Write ('Program na vypocet prumera');
    GotoXY(2,3);
    Writeln ('Zadavej libovolna cisla,',
             'pro ukonceni zadej cislo zaporne');
    Suma := 0.0; Pocet := 0;
Repeat
    Readln(Cislo);
    If Cislo > 0.0 then
        begin
            Suma := Suma + Cislo;
            Pocet := Pocet + 1;
        end;
    until cislo < 0.0;
    If Pocet > 0 then
        begin
            Pr := Suma / Pocet;
            Write (' Prumer = ', Pr:8:2);
        end
    else
        Write('Nebyly zadany vstupni hodnoty')
End.
```

Tento příklad představuje jen modifikaci příkladu, který byl uveden při výkladu příkazu While. Porovnejte obě řešení.

- Poznámky:
- v protikladu vůči příkazu While, příkazy zařazené v rámci tohoto cyklu se vykonají nejméně jednou;
  - jestliže v příkazech cyklu While a For musí být několik příkazů sestupeno do složeného příkazu, potom zde tomu tak není, neboť rezervovaná slova Repeat a Until ohraňují příkazy, které se mají opakovat.

#### 2.7.3 Příkaz Case

Při zápisu algoritmů se často stává, že je třeba předepsat větvení výpočtu do několika alternativ. Příkaz Case odpovídá tomuto cíli, neboť umožňuje aplikovat množinu hodnot výrazu na množinu akcí, které mohou nastat. Syntaxe příkazu je tato:



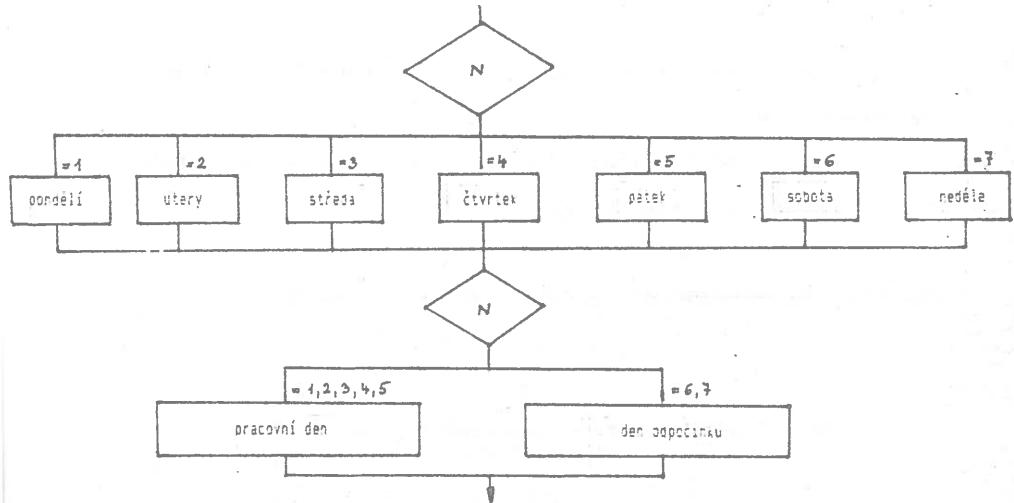
Obr. 2.22 Syntaktický diagram příkazu Case

Příkaz Case, který zajišťuje tzv. vícenásobné větvení, pracuje takto: výraz (selektor) je vyhodnocen a hodnota výsledku umožňuje výběr příkazů k vykonání. Tak jak ukazuje následující příklad, jednomu příkazu může být přiřazeno několik konstant. Selektor musí být ordinálního typu, tzn. jednoduchého, ale ne typu Real.

Příklad:

```
Program Den;
(* precteni celociselne hodnoty z intervalu 1 az 7
   umoznuje jasne popsat jmeno dne v tydnu
   a urcit zda den je pracovni ci ne *)
var
  N : Integer;
Begin
  Write (' Zadej cislo 1 az 7:');
  Readln (N);
  Case N of
    1 : Writeln ('pondeli');
    2 : Writeln ('utery');
    3 : Writeln ('streda');
    4 : Writeln ('ctvrtek');
    5 : Writeln ('patek');
    6 : Writeln ('sobota');
    7 : Writeln ('nedele');
  end;
  (* konec prikazu Case *)
  Case N of
    1,2,3,4,5 : Writeln ('pracovni den');
    6,7         : Writeln ('den odpocinku');
  end
  (* konec prikazu Case *)
End.
```

Pro vyjádření příkazu Case nejsou ve výjovových diagramech speciální značky. Přesto lze pro výše uvedený příklad sestrojit vývojový diagram:



Obr. 2.23 Vývojový diagram pro program Den

Poznámky:

- hodnoty, které představují konstanty, musí být kompatibilní s typem použitým pro výběr každe akce;
- hodnota, která vzniká při vyhodnocení selektoru, má být rovna jedné ze selekčních konstant. Pokud tomu tak není, nastává chyba. V Turbo Pascalu je pro tyto situace definována volitelná možnost vykonání příkazu za rezervovaným slovem `else` (viz procedura `InputStr` v oddile 3.8.3).

## 2.8 Příkaz skoku GoTo

Tento příkaz je stále předmětem častých diskusí, především pokud jde o jeho nutnost a užitečnost. Mnoha výzkumnými pracemi bylo dokázáno, že tento příkaz je teoreticky neužitečný. Přestože v Pascalu existuje několik učinných strukturovaných příkazů, je použití příkazu `GoTo` v některých výjimečných případech vhodné, neboť umožňuje jak

zjednodušení konstrukce programu, tak zvýšení čitelnosti vytvářeného programu.

Příkaz GoTo slouží k přerušení normálního běhu programu. Jeho syntaxe je tato:

příkaz GoTo → GoTo → návěští →

Obr. 2.24 Syntaktický diagram příkazu GoTo

Návěští je standardně definováno jako celé kladné číslo bez znaménka (maximálně čtyřmístné) a používá se k označení příkazu, na který se provede skok.

Pro zvýšení čitelnosti programu musí být všechna návěští použitá v jedné příkazové části předtím deklarována v odpovídající deklarační části, a to v úseku deklarace návěští:

Deklarace návěští → Label → návěští →

Obr. 2.25 Syntaktický diagram úseku deklarace návěští

- Poznámky:
- v Turbo Pascalu je možno, aby návěští bylo vyjádřeno také pomocí identifikátoru;
  - rozsah platnosti návěští je limitován blokem, proto není možný skok dovnitř nebo vně funkci a

procedur

příklad

```
Program Alera;
Label Chyba, Konec;
Const MaxCis = 20000;
var
    I, Soucet, Cislo : Integer;

Begin
    Soucet := 0;
    For I := 1 to 5 do
        begin
            Write('Zadej cislo:');
            Readln (Cislo);
            if Soucet + Cislo > MaxCis then
                Goto Chyba
            else
                Soucet := Soucet + Cislo
        end;
    Writeln('Soucet cini ?', Soucet);
    Goto Konec;
    Chyba : Writeln(' Chyba - prilis velka cisla');
    Konec :
End.
```

Tento program vypočítává součet 5 celých čísel.

Jestliže součet je větší než 20000, potom příkaz

GoTo Chyba umožňuje vytisknout chybové hlášení;

- jak vyplývá z předchozího příkladu, příkaz GoTo snižuje čitelnost programu. Příkaz GoTo tedy nedoporučujeme, možnosti jeho výjimečného používání jsou uvedeny v oddíle 6.2.

2.9 Syntetický příklad

Dále uvedený příklad shrnuje vše podstatné, co bylo uvedeno v této kapitole:

```
Program PrevodNaSI;
(* Tento program prevadi udaje v palcích
do soustavy SI, pro hodnoty
ze zadavaného intervalu se vypočtou
odpovídající hodnoty v mm *)
var
    Dolni, Horni, Palec : Integer;
    mm : Real;
    Odpoved : Char;
```

```
OK : Boolean;

Begin
    Writeln ('Program na prevod hodnot z palcu na mm');
    repeat
        Write ('Zadej dolni a horni hodnotu intervalu:');
        Dolni := 1;
        Horni := 0;
        While Horni < Dolni do
            begin
                Readln (Dolni, Horni);
                if Horni < Dolni then
                    Writeln ('Nespravne udaje, napiste je znova:');
                end;
        for Palec := Dolni to Horni do
            begin
                mm := 25.4 * Palec;
                Writeln (Palec, ' ==> ', mm:7:2, ' mm');
            end;
        repeat
            Write ('Dalsi udaje? odpovezte A nebo N:');
            Readln (Odpoved);
            OK := (Odpoved = 'A') or (Odpoved = 'N');
        until OK;
        until Odpoved = 'N';
End.
```

Tento program dává např. takovéto výsledky:

```
Program na prevod hodnot z palcu na mm
Zadej dolni a horni hodnotu intervalu:1
Nespravne udaje, napiste je znova:
+10
==> 25.40 mm
==> 50.00 mm
==> 75.20 mm
==> 100.40 mm
==> 125.60 mm
==> 150.80 mm
==> 176.00 mm
==> 201.20 mm
==> 226.40 mm
10 ==> 254.00 mm
Dalsi udaje? odpovezte A nebo N:n
Dalsi udaje? odpovezte A nebo N:N
```

Obr. 2.26 Výstup programu PrevodNaSI

Poznámka: Pokud uvedený příklad je pro Vás nepochopitelný, prostudujte znovu tuto kapitolu, případně vypracujte odpovídající cvičení v lit. [4]. Téprve po důkladném osvojení látky z druhé kapitoly pokračujte dále.

### 3. Typy dat

#### 3.1 Úvod

Pojem typ, tak jak byl dosud uváděn, umožňuje jen definování struktur jednoduchých dat, tzn. takových struktur, které jsou dále nedělitelné. Tzn., že např. výraz typu Integer může nabývat jen celočíselných hodnot.

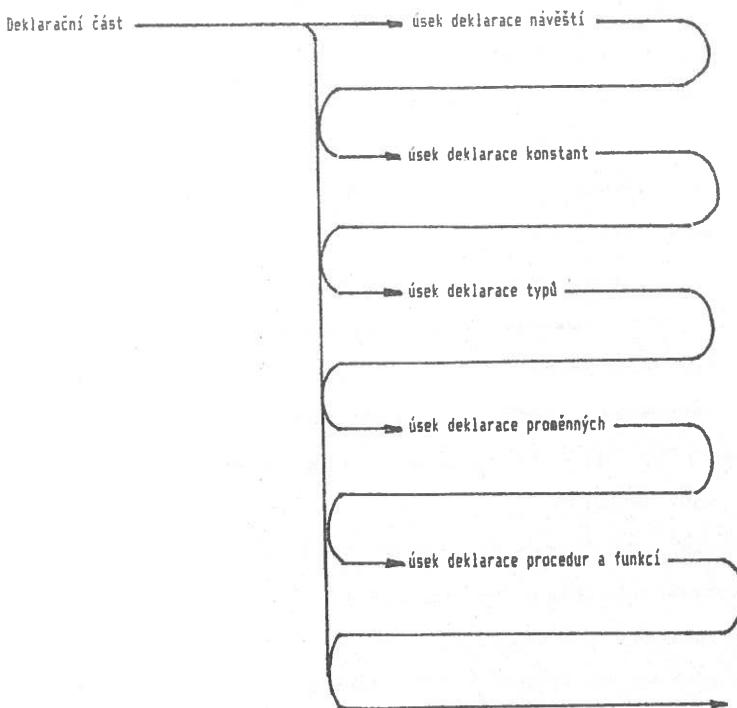
Pro zpracování v počítači je třeba, aby bylo možno definovat složitější struktury. Např. při zpracování osobní evidence struktura informací zahrnuje údaje různé povahy, jako je jméno, věk, pohlaví, vzdělání atd.

Pojem typ dat umožňuje programátorovi strukturování údajů dle potřeb řešeného problému. Typ popisující složitou datovou strukturu může být zpracován na základě jiných složitých struktur a nebo na základě jednoduchých struktur.

Pojem typ dat je v Pascalu široce rozpracován, a tak přináší řadu výhod:

- objekt jako konstanta, proměnná, funkce nebo výraz je vždy jednoznačného typu. Typ určuje množinu hodnot, které mohou být přiřazeny v určitém okamžiku. Proto lze vždy určit typ objektu tím, že zkонтrolujeme deklarační část programu;
- programátor může pro každý typ rozšířit množinu standardních operací tím, že definuje procedury nebo funkce pro nové operace;
- pojem typ umožňuje, aby kompilátor určoval řadu chyb jak při komplikaci, tak při výpočtu. Např. objeví se chyba, když proměnné typu Integer se má přiradit výsledná hodnota reálného výrazu.

Pascal dává programátorovi k disposici prostředky pro definování typů. Typy lze definovat v úseku deklarace typů, který se uvádí v deklarační části, ježíž komplétní struktura je tato:



Obr. 3.1 Syntaktický diagram deklarační části

Pořadí jednotlivých úseků je dáné pravidlem, že deklarace objektů se uvádějí dříve, než se tyto objekty používají.

Úsek deklarace typů má tuto syntaxi:



Obr. 3.2 Syntaktický diagram úseku deklarace typů

Identifikátor představuje jméno vytvářeného typu, deklarace typu představuje požadovanou definici pro vytvářený typ.

Pascal umožňuje definovat tři kategorie typů: typy jednoduché, typy strukturované a typy ukazatel. První dvě kategorie typů budou vysvětleny v této kapitole.

### 3.2 Jednoduché typy

Kromě jednoduchých typů, které jsou předdefinovány a byly již uvedeny v oddíle 2.3, existují další dva jednoduché typy, jejichž definici zajišťuje programátor.

Všechny jednoduché typy jsou množiny, které mají tyto vlastnosti:

- hodnoty těchto typů jsou "primitivní", tzn., že nemohou být rozloženy na jednodušší hodnoty;
- hodnoty každého typu vytvářejí uspořádanou množinu;
- obecně použitelné operace pro tyto hodnoty jsou přiřazení a porovnávání.

Explicitní deklarace typů je jedna ze základních výhod jazyka Pascal. Možnost definování (případně i redefinování předdefinovaných typů) zvyšuje čitelnost programu a obecné použití jazyka.

Pascal nabízí dvě možnosti pro definování jednoduchých typů, a to výčtový typ a typ interval.

### 3.2.1 Výčtový typ

Jak vyplývá z názvu, výčtový typ umožňuje pojmenovat množinu hodnot nějakého objektu, přičemž se zajišťuje výčet (vyjmenování) všech hodnot, kterých tento objekt bude nabývat.

Aby program byl čitelnější, každá hodnota ve výčtu se označuje symbolickým jménem (identifikátorem).

#### DEKLARACE

Způsob deklarace výčtového typu je uveden v syntaktickém diagramu:



Obr. 3.3 Syntaktický diagram deklarace výčtového typu

#### Příklad

Předpokládejme, že je třeba vypracovat program pro automatické řízení dopravních signálů. V tomto programu se bude určitě pracovat s barvami světelných signálů. Je přirozené pracovat s hodnotami, které přesně odpovídají objektům, a znát, jakých barev může nabývat světlo na semaforu.

```
Program Doprava;  
type  
    Signal = (zelena, zluta, cervena);  
var  
    Semafor1, Semafor2 : Signal;
```

Hodnoty výčtového typu jsou uspořádány vzestupně, v pořadí dle zápisu v seznamu identifikátorů. Tzn., že v uvedeném příkladu hodnota zelená je menší než žlutá a žlutá je menší než červená.

Proto v programu lze např. zapsat:

```
if Semafor1 = Cervena then  
    Semafor2 := Zelena;  
pro vyjádření, že na semaforu 2 bude zelená, když na  
semaforu 1 bude červená.
```

Je zřejmé, že uvedený zápis je čitelnější, než zápis:

```
if Semafor1 = 2 then  
    Semafor2 := 0;
```

#### Jiný příklad

```
Program Agenda;  
type  
    Den = (Pondeli, Uttery, Streda, Ctvrtok,  
            Patek, Sobota, Nedele);  
var  
    Vcera, Dnes, Zitra : Den;
```

Jména proměnných pro typ Den jsou závislá na pořadí v deklaraci výčtového typu a je užitečné srovnávat, jestli den je předchozí či následující vzhledem k nějakém dni.

Poznámky: - identifikátory ve výčtovém typu, které představují rozdílné vyjmenované hodnoty, jsou vlastně identifikátory konstant. Takto je zapsán předdefinovaný typ Boolean:

type

Boolean = {True, False};

- není možno deklarovat stejný identifikátor pro dva typy, neboť kompilátor by tuto situaci nedokázal vyřešit:

Program Agenda;

type

Den = {Po, Ut, St, Ct, Pa, So, Ne};  
Vikend = {So, Ne};

neboť to znamená, jako kdybychom chtěli zapsat:

Program

Const

Po = 0; Ut = 1; St = 2;

Ct := 3; Pa = 4;

So := 5; Ne = 6;

(\* tato deklarace konstant odpovídá typu Den \*)

So = 0; Ne = 1;

(\* tato deklarace konstant odpovídá typu Vikend\*);

- v programu jsou dvě chyby, neboť konstanty So a Ne jsou dvakrát deklarovány. Tento rozpor lze snadno vyřešit pomocí typu interval (viz oddíl 3.2.2).

## POUŽÍVÁNÍ

Na základě uvedených charakteristik a omezení je možno poměrně snadno určit operace, které Pascal povoluje pro výčtový typ. Přípustné jsou tyto operace:

- přiřazení hodnoty;
- předdefinované standardní funkce: Ord, Succ, Pred a
- porovnávání.

### Přiřazení

Proměnné výčtového typu lze přiřadit hodnotu symbolicky označenou pomocí identifikátoru v deklaraci typu.

### Příklad

```
Program Doprava;  
  
type  
  
    Signal = (zelena, zluta, cervena);  
  
var  
  
    Semafor1, Semafor2, Semafor3 : Signal;
```

povolené přiřazení	nesprávné přiřazení
<pre>begin      Semafor1 := zelena;      Semafor2 := zluta;      Semafor3 := Semafor1  end</pre>	<pre>begin      Semafor1 := zeleny;      Semafor2 := 2;      Semafor3 := modra</pre>

Pro výčtový typ aritmetické operace nemají smysl. Např. příkaz `Semafor3 := Semafor2 + Semafor1;` bude komplátorem označen jako chybný. Ale příkaz `Semafor3 := Semafor1;` bude považován za správný, neboť se jedná jen o přiřazení.

### Standardní funkce

Protože výčtový typ je založen na pořadí, je možno s proměnnými výčtového typu manipulovat pomocí tří

standardních funkcí, které již byly vysvětleny (viz oddíl 2.3). Jsou to:

- pořadí prvku v seřazené množině: Ord
- následník prvku v seřazené množině: Succ
- předchůdce prvku v seřazené množině: Pred.

Funkce Ord použitá na hodnotu výčtového typu přináší celočíselný výsledek, který odpovídá pořadí této hodnoty ve výčtu hodnot při definici typu.

Příklad

```
Program Doprava;
type
  Signal = (zelena, zluta, cervena);
var
  Semafor1, Semafor2, Semafor3 : Signal;

Begin
  ClrScr;
  Writeln (' Ordinalni cislo zelene: ', Ord (zelena) );
  Writeln (' Ordinalni cislo zlute: ', Ord (zluta) );
  Writeln (' Ordinalni cislo cervene: ', Ord (cervena) );
  Semafor1 := zelena;
  Writeln (' Zelene svetlo --> ', Ord (Semafor1) );
  Semafor2 := zluta;
  Writeln (' Zlute svetlo --> ', Ord (Semafor2) );
  Semafor3 := cervena;
  Writeln (' Cervene svetlo --> ', Ord (Semafor3) )
End.
```

```
Ordinalni cislo zelene: 0
Ordinalni cislo zlute: 1
Ordinalni cislo cervene: 2
Zelene svetlo --> 0
Zlute svetlo --> 1
Cervene svetlo --> 2
```

Obr. 3.4 Výstup programu Doprava

Je třeba podotknout, že první ordinální číslo v množině výčtového typu je 0.

Jestliže se na hodnoty výčtového typu použijí funkce

Pred a Succ, získá se hodnota předchozí, respektive následující.

Pred(zluta) zelena

Succ(zluta) cervena

Funkce Pred a Succ se nemohou použít pro první, respektive poslední hodnotu množiny, neboť dojde k chybě.

### Porovnávání

Pro výčtový typ lze použít již definované relační operace ( "=", "<>", ">", "<", "<=" ). Výsledek porovnávání závisí na pořadí hodnot ve výčtu. Když uvažujeme, že proměnné D1 a D2 jsou typu Den (viz předchozí příklad), potom při porovnávání dostaneme tyto výsledky:

hodnota D1	hodnota D2	porovnání	výsledek
sobota	sobota	D1 = D2	true
		D1 <> D2	false
pondeli	pátek	D1 < D2	true
		D2 > D1	true
		D1 <> D2	true
		D1 = D2	false

Poznámka: - obvyklá chyba začátečníka spočívá v tom, že zaměňuje hodnotu výčtového typu se symbolickým zobrazením:

Příklad

budíž Samohláska a proměnná L definována takto:  
type

Samohlaska = (A, E, I, O, U);

var

L : Samohlaska;

v příkazové části nelze zapsat:

L := I; Writeln(L);

neboť hodnota L je definována pořadím identifikátoru I v seznamu (A,E, I,O,U). Tato hodnota nemá nic společného s hodnotou typu Char 'I'!

3.2.2 Typ interval

Aby programy byly přesnější a čitelnější, je vhodné, když v programovacím jazyku jsou možnosti pro omezení rozsahu proměnných na podmnožinu určených hodnot daného jednoduchého typu.

Předpokládejme, že v programu se manipuluje s teplotami, které vzhledem k řešenému problému jsou celá čísla mezi -20 a +30 stupni C. Tím, že je možno deklarovat nový typ, který označuje dolní a horní hranici, se zvýší spolehlivost programu (díky detekci chyb při překročení hranice) a usnadní se čtení programu.

### DEKLARACE

Typ interval má tuto syntaxi:

Typ interval → dolní hranice → . . . → horní hranice →

Obr. 3.5 Syntaktický diagram pro deklaraci intervalu

Hodnoty, které vytvářejí interval jsou konstanty stejného typu a označují nejmenší a největší hodnotu intervalu (dolní hranice musí být menší nebo rovna horní hranici). Interval je chápán jako uzavřený.

#### Příklad

```
type  
    Cislice = 0..9;  
var  
    Cislo1 : Cislice;
```

Proměnná Cislo1 může nabývat jen hodnot mezi 0 a 9 (včetně 0 a 9). Hodnoty proměnných typu interval mohou být předdefinovány, pokud hranice jsou konstanty předdefinovaného typu.

#### Příklad

```
Program Interval;  
Const  
    Leto = 30;  
    Zima = -20;  
Type  
    Teplota = Zima..Leto;  
    Pismeno = 'a'..'t';  
var  
    Stupen : Teplota;
```

Znak : Písmeno;

Je samozřejmě, že konstrukce typu interval není možná pro typ Real. Konstrukce tohoto typu kompilátor signalizuje jako chybné.

Hranice intervalu mohou být symbolické hodnoty výčtového typu.

Příklad

```
Program Agenda;
Type
  Den = (Po, Ut, St, Ct, Pa, So, Ne);
  Tyden = Po..Pa;
  Vikend = So..Po; (* Chyba *)
```

Deklarace Vikend je nespojitá; typ interval vyžaduje, aby dolní hranice byla menší nebo rovna horní!

Používání

Jednou z podstatných výhod typu interval je, že umožnuje kontrolu správnosti hodnot. V okamžiku přiřazení nějaké hodnoty proměnné typu interval se zajišťuje ověření správnosti. V případě, že hodnota nepatří do daného intervalu, je signalizována chyba (viz též poznámky na konci oddílu).

Po typ interval lze používat operace nebo funkce, které jsou použitelné pro proměnné stejného typu jako jsou hranice daného intervalu. Např. proměnná typu interval s celočíselnými hranicemi, bude chápána jako podmnožina typu Integer a všechny operace a funkce použitelné pro typ Integer budou také použitelné.

Stejný princip se využívá, když hranice jsou typu Char nebo Boolean.

Pro proměnné typu interval, které jsou zkonstruovány na základě výčtového typu, platí stejná pravidla jako pro proměnné výčtového typu.

Příklad

```
Program Test;
Type
  Mesic = 1..12;
  Den = (Po, Ut, St, Ct, Pa, So, Ne);
  Tyden = Po..Pa;
Var
  OK : Boolean;
  Zari, Rijen : Mesic;
  Zitra : Tyden;
```

správné použití	nesprávné použití
Zari := 3; Rijen := Zari + 1; Zitra := Ut; OK := Ut < Pa;	Zari := 15; Rijen := Zari * 2; Zitra := So;

Poznámky:

- předdefinovaný typ Byte v Turbo Pascalu je vlastně definován takto: `Byte = 0..255;`
- používání typu interval se doporučuje, neboť kromě zvýšené čitelnosti přináší i úsporu paměti. Interval typu Integer, kde obě hranice jsou mezi 0 a 255, zabírá pouze 1 Byte paměti;
- pro detekci chyb při překročení intervalu je třeba, aby byla aktivní direktiva komplikátoru `R+`. Do neodladěných programů se tato direktiva vkládá zapisem `($R+)`, podrobněji viz oddíl 7.4.

### 3.3 Úvod do strukturovaných typů

Dosud probírané typy umožňují definovat jen množiny jednoduchých hodnot, ale mnohé problémy vyžadují manipulaci s proměnnými, které mohou nabývat složitějších hodnot; např. při práci s komplexními čísly je třeba, aby komplexní číslo bylo vyjádřeno pomocí dvou čísel typu Real, první pro reálnou část, druhé pro imaginární část.

Obdobně je výhodné, když existuje možnost přímé manipulace s proměnnými, které mají rozdílnou povahu. Např. pro výpočet mzdy zaměstnance potřebujeme celou osobní kartu a z ní vyjímáme údaje jako je stáří, pohlaví, kvalifikace apod.

Tyto úkoly a mnohé další zajišťujeme v Pascalu pomocí strukturovaných typů. Strukturovaným typem se nazývá každý typ, jehož definice se uskutečňuje odkazem na jiný typ, tzn., že strukturovaný typ je zkonstruován na základě jednoho nebo několika typů. V Pascalu jsou definovány tyto čtyři strukturované typy:

- typ pole (Array),
- typ záznam (Record),
- typ množina (Set),
- typ soubor (File).

Pro každý strukturovaný typ budou uvedeny předdefinované procedury a funkce použitelné pro proměnné, které jsou deklarovány pro tyto typy.

### 3.4 Typ pole

Pole (Array) je homogenní datová struktura, která se skládá z konečného počtu složek (prvků) stejného typu. Každé složce může být připojena hodnota (index), která umožňuje individuální přístup ke každé složce.

#### 3.4.1 Deklarace

Pole je možno definovat takto:

Deklarace pole —————→ Array —————→ [→typ indexu →] —————→ of —————→ typ složek —————→

Obr. 3.6 Syntaktický diagram pro jednoduchou deklaraci pole

typ indexu musí být ordinálního typu. Typ složek může být libovolného jednoduchého či strukturovaného typu, dokonce to může být i typ ukazateł.

#### Příklady deklarací

##### Type

```
Vektor = array [1..10] of Real;
```

```
Cislice = 0..9;
```

```
TabBarva = array [Cislice] of Boolean;
```

```
Den = (Po, Ut, St, Ct, Pa, So, Ne);
```

```
Prezence = array [Den] of Integer;
```

```
Kod = array ['A'..'Z'] of Cislice;
```

Typ složky pole může být také typu pole, tzn., že lze konstruovat 2,3,...n rozměrná pole.

Příklad

Type

```
tMat = array [1..3] of array [1..5] of Integer;
```

Typ tMat definuje pole o 3 složkách, přičemž každá tato složka

je pole o 5 složkách typu Integer.

```
tProstor = array [Boolean] of array [1..20]  
                 of array ['A'..'V'] of Integer;
```

Typ tProstor je definován jako třírozměrné pole, první rozměr nabývá hodnot True a False, druhý rozměr hodnot 1 až 20 a třetí rozměr hodnot 'A' až 'V'; složky jsou typu Integer. Existuje ekvivalentní zápis, který zjednodušuje popis deklarací.

Předchozí deklaraci pro typ tProstor lze zapsat takto:

Type

```
tProstor = array [Boolean, 1..20, 'A'..'V'] of  
Integer;
```

Obdobně lze zapisovat deklaraci pro matice:

Type

```
Radky, Sloupce = 1..20;
```

```
tMatice = array [Radky, Sloupce] of Real;
```

Tzn., že syntaktický diagram na obr. 3.6 je třeba zobecnit:



Obr. 3.7 Syntaktický diagram pro deklaraci pole

### 3.4.2 Používání

Abychom mohli použít složku pole z n-rozměrného pole, je třeba nejdříve označit proměnnou typu pole a potom vybrat nějakou složku pomocí indexů. Proto jedna složka pole se nazývá jako "indexovaná proměnná". Jestliže jeden, nebo několik indexů se získává vyhodnocením jednoho nebo několika výrazů, potom kompilátor ověřuje kompatibilitu mezi typem výsledku každého výrazu a typem odpovídajícího indexu.

### Jednorozměrná pole

13

```
var  
  PReal : array [1..10] of Real;  
  PBool : array ['A'..'D'] of Boolean;  
  PChar : array [Boolean] of Char;
```

Typ PReal definuje jednorozměrné pole, které obsahuje 10 složek, každá složka je typu Real. Pro označení složky s indexem 3 je možno zapsat:

PReal [3] nebo

PReal [I] je-li I celočíselná proměnná  
o hodnotě 3

PReal [(I-J) div 2+1] jsou-li I a J celocíselné proměnné  
a mají hodnoty 10 a 6.

Typ PBool definuje pole o čtyřech složkách (hodnoty indexů jsou čtyři: 'A', 'B', 'C', 'D'), každá složka je typu Boolean. Zápis PBool['C'] umožňuje přístup ke složce s indexem 'C'.

Typ PChar definuje pole o dvou složkách (hodnoty indexů jsou dvě: True a False), každá složka je typu Char. PChar [True] umožňuje přístup ke složce s indexem True, podobně jako PChar [I=5], jestliže proměnná I je celočíselná a má hodnotu 5.

#### Vícerozměrná pole

##### Příklad

```
var  
    Tab1 : array [-2..+1, Boolean] of Char;
```

Pole Tab1 lze znázornit takto:

-2, False  
-2, True  
-1, False  
-1, True  
0, False  
0, True  
1, False  
1, True

	X

Tab [0,True] := 'X';

Pole Tab1 má osm složek (4\*2), každá složka je typu Char.

Jak vyplývá z uvedeného obrázku, dvourozměrné pole je vlastně jednorozměrné pole pro jiné jednorozměrné pole:

Tab1 : array [-2..+1] of array [Boolean] of Char;

Ve vícerozměrném poli lze přistoupit ke koncové složce nebo k dílčímu poli. Např. zápis:

Tab1 [-1,True] označuje koncovou složku,

Tab1 [ 1] označuje dílčí pole.

Naopak zápis:

Tab1 [True] je nesprávný, neboť Tab1 [Boolean] není dílčí pole od Tab1.

Dílčí pole lze používat nejen pro dvourozměrná, ale i pro vícerozměrná pole.

#### Příklad

```
Tab3 : array [Boolean] of array [1..20]
        of array ['A'..'D'] of Real;
```

V tomto příkladu existují dvě dílčí pole, ke kterým lze přistupovat např. takto: Tab3 [True,3] nebo Tab3 [False]. Naopak zápis Tab3 ['B'] je nesprávný.

Poznamenejme, že pořadí zápisu typů indexů v deklaraci vícerozměrných polí určuje možná dílčí pole.

#### Přiřazení

Pro přiřazení hodnoty nějaké složce pole je nutné, buď aby typ hodnoty byl stejného typu jako složka pole, nebo aby typ hodnoty byl kompatibilní s typem složky.

Příklad

```

Program Matice;
Const
  N = 10;
  M = 20;
var
  I : Byte;
  Matice : array [1..N, 1..M] of Real;
Begin
  I := 9;
  Matice [I,12] := Matice [8,10];
  Matice [I,12] := 33 * 7
End.
```

Poznámky:

- jen jedna operace je použitelná při práci s dílčími polí a s polí jako celek, a to přiřazení. Používání dílčích polí umožňuje vyjádřovat algoritmy jiným způsobem, než jak je obvyklé v jiných programovacích jazycích (FORTRAN). Vynulování všech prvků Matice z předchozího příkladu lze zajistit i takto:

```

(* vynulování prvního řádku *)
for I := 1 to M do
  Matice[I,I] := 0;
(* přesun prvního řádku do dalších řádků *)
for I := 2 to N do
  Matice[I] := Matice [1];
```

- pro kontrolu rozsahu přípustných indexů je třeba v Turbo Pascalu zapnout direktivu R, příkazem  
{\$R+} ;
- Turbo Pascal nabízí možnost přiřazení hodnot složkám pole ihned při komplaci, viz oddíl 3.9.

3.4.3 Příklad

Na závěr oddílu o polích je uveden příklad pro součet a rozdíl dvou čtvercových matic. V úseku deklarací konstant jsou zadány rozměry matic. Program je doplněn výsledky.

```
Program VypocetMatic;
{$R+} { Program vypocitava součet a rozdíl dvou matic }
Const
  N = 2;
  M = 2;
type
  tMat = array [1..N, 1..M] of Real;
var
  Mat1, Mat2 : tMat;
  I, J         : Integer;
  Pom          : Real;
Begin
  for I := 1 to N do                      { čtení Mat1 }
    for J := 1 to M do
      begin
        Write ('Zadej prvek Mat1 (',I,',',J,') :');
        Read (Mat1 [I,J]);
        Writeln (Mat1[I,J]:5:1);
      end;
  for I := 1 to N do                      { čtení Mat2 }
    for J := 1 to M do
      begin
        Write ('Zadej prvek Mat2 (',I,',',J,') :');
        Read (Mat2 [I,J]);
        Writeln (Mat2[I,J]:5:1);
      end;
  for I := 1 to N do                      { výpočet }
    for J := 1 to M do
      begin
        Pom := Mat1 [I,J];
        Mat1 [I,J] := Mat1 [I,J] + Mat2 [I,J];
        Mat2 [I,J] := Pom - Mat2 [I,J];
      end;
  for I := 1 to N do                      { výstup výsledku }
    for J := 1 to M do
      begin
        Write (' Výsledek v Mat1 (',I,',',J,') :',
               Mat1 [I,J]:5:1);
        Writeln (' Výsledek v Mat2 (',I,',',J,') :',
               Mat2 [I,J]:5:1);
      end
  End.
```

Tento program přináší tyto výsledky:

```
Zadej prvek Mat1 (1,1) :2 2.0
Zadej prvek Mat1 (1,2) :4 4.0
Zadej prvek Mat1 (2,1) :5 5.0
Zadej prvek Mat1 (2,2) :6 6.0
Zadej prvek Mat2 (1,1) : 0.0
Zadej prvek Mat2 (1,2) :3 3.0
Zadej prvek Mat2 (2,1) :3 3.0
Zadej prvek Mat2 (2,2) :4 4.0
Výsledek v Mat1 (1,1) : 2.0 Výsledek v Mat2 (1,1) : 2.0
Výsledek v Mat1 (1,2) : 7.0 Výsledek v Mat2 (1,2) : 1.0
Výsledek v Mat1 (2,1) : 8.0 Výsledek v Mat2 (2,1) : 2.0
Výsledek v Mat1 (2,2) :10.0 Výsledek v Mat2 (2,2) : 2.0
```

Obr. 3.8 Výstup programu VýpočetMatic

### 3.5 Typ záznam

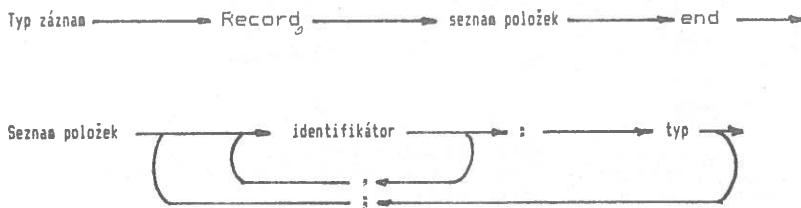
Jestliže je zapotřebí do jednoho celku zařadit prvky rozdílné povahy, jako je tomu např. na osobní kartě, kde jsou uváděny různé údaje (jméno, datum narození, místo bydlistě, ...), potom takovéto struktury v Pascalu zapisujeme pomocí typu záznam (angl. Record).

Záznam je datová struktura, která se skládá z určitého počtu složek libovolného typu. Tyto složky, které se nazývají položky, označujeme jmény – identifikátory.

Lze říci, že záznam je nejobecnější strukturovaný typ, který je v Pascalu zaveden.

#### 3.5.1 Deklarace

Definici typu záznam je možno vyjádřit takto:



Obr. 3.9 Syntaktické diagramy typu záznam a seznamu položek

#### Příklad

```
type
  Adresa = record
    Ulice : array [1..20] of Char;
    Dum   : Integer;
    Mesto : array [1..15] of Char;
  end;
```

V tomto příkladu je definován typ záznam, který je označen jako Adresa. Položkami záznamu Adresa jsou pole Ulice a Mesto, která jsou typu Char, a číslo Dum, které je typu Integer.

#### Příklad

```
Const  
      N = 20;  
type  
    Datum = record  
      Den : 1..31;  
      Mesic : 1..12;  
      Rok : Integer;  
    end;  
    Kniha = record  
      Nazev : array [1..N] of Char;  
      Autor : array [1..N] of Char;  
      CSSR : Boolean;  
      Stran : Integer;  
      DatVydani : Datum;  
    end;
```

V tomto příkladu jsou definovány dva typy záznam, nichž první (Datum) se využívá při definici typu položky DatVydání v druhém záznamu.

Poznámky:

- identifikátor, který se využívá pro označení typu položky musí být předem definován. Tento princip zvyšuje čitelnost programu a zjednodušuje realizaci kompilátoru;
- struktury typu záznam lze hierarchicky, téměř neomezeně, seřazovat. Jediným omezením je kapacita operační paměti.

#### 3.5.2 Používání

Abychom zajistili přístup k nějaké položce typu záznam, je nutno nejdříve označit proměnnou typu záznam a potom zvolit požadovanou položku pomocí vlastního identifikátoru:

Přístup k položce —————→ identifikátor záznamu → → identifikátor položky —————→

Obr. 3.10 Syntaktický diagram pro přístup k položce

Identifikátor položky označuje položku, která může označovat záznam, ve kterém lze zajistit přístup k položce stejným způsobem. Jestliže položka je typu pole, potom se pro přístup k složce pole využívají indexy (viz oddíl 3.4.2).

#### Příklad

```
var
    Skripta : Kniha;      { Kniha byla jiz definovana }

begin
    Skripta.CSSR := True;
    Skripta.Stran := 220;
    Skripta.DatVydani.Mesic := 3;
    Skripta.DatVydani.Rok := 1987;
    Skripta.Nazev [1] := 'X';
end;
```

- zápis "Skripta.CSSR" umožňuje přístup k položce CSSR v záznamu Skripta;
- zápis "Skripta.Stran" umožňuje přístup k položce Stran v záznamu Skripta;
- zápis "Skripta.DatVydani.Mesic" umožňuje přístup k položce Mesic ze záznamu DatVydani, který je vlastně položkou záznamu Skripta;
- zápis "Skripta.Nazev[1]" umožňuje přístup k první složce položky Nazev v záznamu Skripta.

V některých případech, zvláště u složitých struktur, se tento způsob označování položek stává až příliš složitý, a proto pro zjednodušení těchto situací byl v Pascalu navržen příkaz With.

### 3.5.3 Příkaz With

Cílem tohoto příkazu je zjednodušit označování položek proměnných typu záznam. Příkaz With má tuto syntaxi:

Příkaz With —————→ With —————→ proměnná typu záznam —————→ Do —————→ příkaz —————→

Obr. 3.11 Syntaktický diagram příkazu With

V části příkaz tohoto příkazu se pro označování položek uvádějí jen identifikátory položek, identifikátory záznamů se vymezují.

### Příklad

```
var
    Skripta : Kniha;

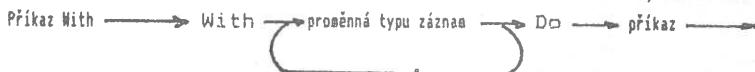
begin
    With Skripta Do
    begin
        CSSR := True;
        Stran := 200
    end
end.
```

Část příkaz v příkazu With může obsahovat také jiné příkazy With.

Příklad

```
var  
    Skripta : Kniha;  
  
Begin  
    With Skripta Do  
        begin  
            CCSR := True;  
            Stran := 230;  
            With DatVydani Do  
                begin  
                    Den := 22;  
                    Mesic := 3;  
                    Rok := 1987  
                end  
            end  
        End.  
    End.  
  
(* konec prikazu With DatVydani *)  
(* konec prikazu With Skripta *)
```

V těchto příkladech není nutné uvádět dva příkazy With. Pro usnadnění zápisu i čtení je možno zapsat jeden příkaz With podle této syntaxe:



Obr. 3.12 Syntaktický diagram pro úplný příkaz With

Příklad

```
(* zjednoduseni predchoziho prikazu *)  
var  
    Skripta : Kniha;  
  
Begin  
    With Skripta,DatVydani Do  
        begin  
            CCSR := True;  
            Stran := 230;  
            Den := 22;  
            Mesic := 3;  
            Rok := 1987  
        end  
    End.  
(* konec prikazu With *)
```

Poznámky: - Pascal umožňuje i tuto formu zápisu příkazu With:

```
With Skripta.DatVydani Do  
potom se tento příkaz použije takto:
```

```
With Skripta.DatVydani Do  
begin  
    Den := 22;  
    Mesic := 3;  
    Rok := 1987  
end  
(* konec prikazu With *);
```

- v Turbo Pascalu pod operačním systémem CP/M je povolen různý stupeň hierarchie příkazů With. Ve verzji 3.0 je implicitně 4. Tuto hodnotu lze však změnit pomocí direktivy W na hodnoty 0 - 9.

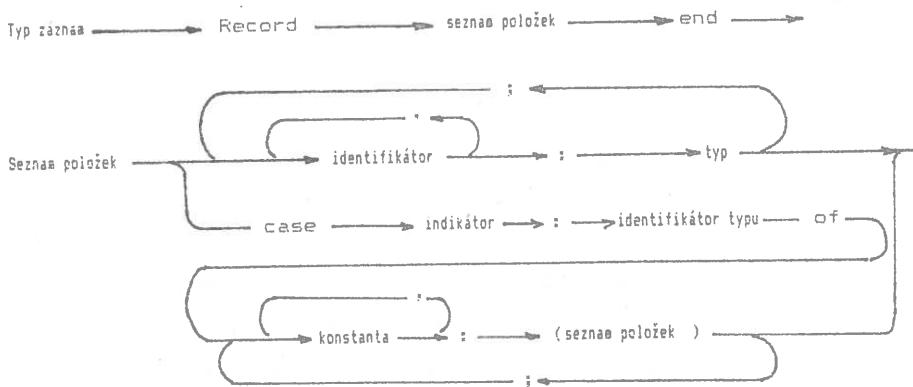
#### 3.5.4 Variantní záznamy

Typ záznam, tak jak byl dosud vysvětlen, umožňuje deklaraci proměnných, které mají stejnou strukturu. V některých případech je třeba, aby proměnné stejné povahy měly rozdílnou datovou strukturu. Např. jestliže chceme zpracovávat osobní údaje, je třeba podle rodinného stavu rozlišit jednotlivé údaje a vytvořit vhodné datové položky, např. pro jméno manžela/ky, pro počet dětí apod.

Tento cíl se zajišťuje pomocí variantních záznamů (respektive záznamů s proměnnou částí).

### DEKLARACE

Úplná definice typu záznam je tato:



Obr. 3.11 Syntaktické diagramy typu záznam  
a úplného seznamu položek

Tento diagram vyžaduje toto vysvětlení:

- pevná část musí být uvedena jako první;
- definice proměnlivé části začíná za rezervovaným slovem  
**Case:**
- jen pro proměnlivou část je nutno dále pojmenovat:
  - pokud se využívá indikátor (rozlišovací položka), tak  
je to položka z pevné části záznamu;
  - každé variantě může být přiřazena jedna nebo několik  
konstant;
  - typ indikátoru musí být ordinálního typu;
  - konstanty musí být stejného typu jako indikátor;
  - všechny možné hodnoty indikátoru se musí objevit jako  
konstanty.

### Příklad

```
type
  DruhStroje = (Traktor, Automobil, PracStroj);
  tStroj = record
    Jmeno : array [1..20] of Char;
    RokVyroby : Integer;
    Cena: Integer;
    Naklady : Real;
    Vprovozu : Boolean;
    Case Druh : DruhStroje of
      Traktor : (PocetHodin : Integer;
                  SpotNafty : Integer);
      Automobil : (PocetKm : Integer;
                    PocetTun : Real;
                    SpotPHM : Integer);
      PracStroj : (PocetDni : Integer);
    end;
```

V tomto příkladu jsou popsány tři druhy strojů (Traktor, Automobil a Pracovní stroj). Protože u každého druhu strojů se evidují různé údaje, byl navržen variantní záznam Stroj.

Poznámky: - indikátor ve variantní části nemusí být předem definován, potom ve vyše uvedeném příkladu by mohl být zápis:

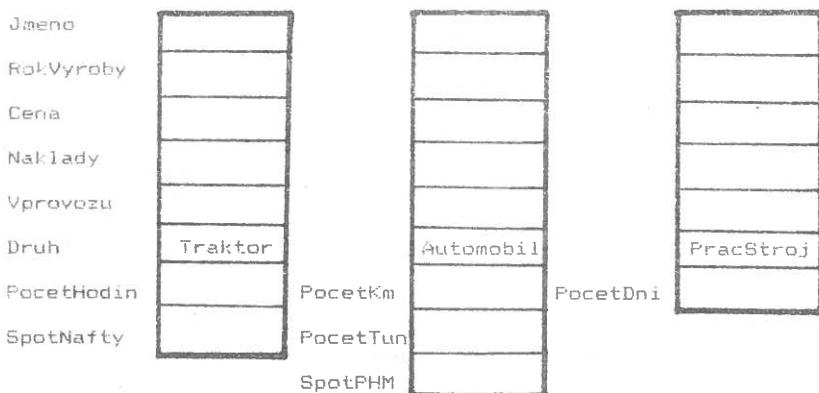
```
Case DruhStroje of
  potom položka Druh nemusí existovat. V tomto případě programátor musí zajistit řádné zpracování všech možností. Nepoužívejme selektor bez jeho předběžné definice, neboť jinak mohou nastat v programu zbytečné chyby.
```

### POUŽÍVÁNÍ

Uvažujme, že platí deklarace typů z minulého příkladu. Potom zápisem v úseku deklarací proměnných

```
var
  Stroj : tStroj;
```

se určuje, že proměnná Stroj bude nabývat jednu ze tří struktur v závislosti na hodnotě položky Druh:



Poznamenejme, že proměnná Stroj má 6 položek v pevné části (Jmeno, RokVyroby, Cena, Naklady, Vprovozu a Druh). Variantní část se skládá ze tří alternativ, jejichž struktura závisí na hodnotě selektoru položky.

Pro přístup k položce z variantní části používáme stejné označení jako pro položky z pevné části.

#### Příklad

```
var
    Stroj1, Stroj2 : tStroj;
begin
    Stroj2.PocetKm := 20;
    Stroj1 := Stroj2;
end;
```

Stejně jako pro proměnné typu pole je možno zajistit globální přiřazení proměnné typu záznam jiné proměnné stejného typu.

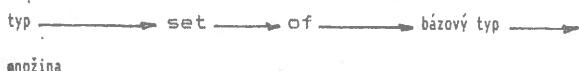
Poznámka - kompilátor pro proměnnou typu záznam s variantní částí přiděluje prostor podle délky největší varianty.

### 3.6 Typ množina

V jazyku Pascal lze pracovat s datovými objekty, jejichž hodnotami jsou množiny. Hodnoty typu množina se vytvářejí podle hodnot tzv. bázového typu. Bázový typ musí být ordinálního typu.

#### 3.6.1 Deklarace

Typ množina se deklaruje takto:



Obr. 3.14 Syntaktický diagram deklarace typu množina

Bázový typ může být kterýkoliv jednoduchý ordinální typ.

#### Příklad

```
type
  Rok = (Jaro, Leto, Podzim, Zima);
  Období = set of Rok;
```

Hodnota typu množina se vytváří výčtem hodnot bázového typu v hranatých závorkách.

#### Příklad

```
[Jaro, Leto]
[Jaro, Leto, Podzim]
[Leto]
```

Jestliže hodnota typu množina obsahuje několik za sebou jdoucích hodnot, lze výčet nahradit zápisem intervalu:

[Leto..Zima] je ekvivalentní [Leto, Podzim, Zima].

Prázdná množina se označuje [].

### 3.6.2 Používání

V Pascalu jsou definovány obvyklé operace z teorie množin.

#### Sjednocení

Sjednocení dvou množin je označováno "+".

#### Příklad

Var

Licha, Suda, Cislice: set of Integer;

Begin

Suda := [0,2,4,6,8];

Licha := [1,3,5,7,9];

Příkaz "Cislice := Suda + licha"

je ekvivalentní

Cislice := [0,1,2,3,4,5,6,7,8,9];

nebo

Cislice := [0..9];

#### Průnik

Průnik dvou množin je označován "\*".

#### Příklad

Program Mnozina;

var

Samohlaska, Slovo, Prunik : set of Char;

Begin

Samohlaska :=

['A','a','E','e','I','i','O','o','U','u','Y','y'];  
Slovo := ['S','u','c','h','d','o','l'];

```
Prunik := Samohlaska * Slovo;
if 'u' in Prunik then
  Writeln ('Spravne');
End.
```

Výsledek je

```
Prunik = ['u', 'o']
```

### Rozdíl

Rozdíl dvou množin je označován "-".

### Příklad

Program Mnozina:

```
var
  Samohlaska, Slovo, Rozdili, Rozdil2 : set of Char;
Begin
  Samohlaska := ['a','e','i','o','u','y'];
  Slovo := ['S','u','c','h','d','o','l'];
  Rozdili := Samohlaska - Slovo;
  Rozdil2 := Slovo - Samohlaska;
End.
```

Výsledek je :

```
Rozdili = ['a','e','i','y']
```

```
Rozdil2 = ['S','c','h','d','l']
```

### Rovnost a nerovnost

Rovnost nebo nerovnost dvou množin může být ověřena pomocí operátorů "=" nebo "<>".

### Příklad

```
['a','e','i'] <> ['a','i','e']
```

dává výsledek False

```
[1,3,7] = [1,7,3]
```

dává výsledek True

### Další operace

Operace "je obsažena v" se zapisuje pomocí "<="

Operace "obsahuje" se zapisuje pomocí ">="

Příklad

[2,3,4] <= [1,2,3,4,5] dává výsledek True

Relace "je prvkem množiny" se zapisuje pomocí "IN".

Operandem může být hodnota a množina. Ověřuje se, zda zadaná hodnota je prvkem množiny. Jestliže hodnota patří do množiny, výsledek je True, jinak je výsledek False.

Příklad

```
Program Mnozina;
var
    Samohlaska : set of Char;
    Zn1, Zn2 : Char;

Begin
    Samohlaska := 
        ['A','a','E','e','I','i','O','o','U','u','Y','y'];
    Zn1 := 'A';
    Zn2 := 'X';

    ...

    lze testovat:
    if Zn1 in Samohlaska then
        { výsledek bude True }
    if Zn2 in Samohlaska then
        { výsledek bude False }
```

Poznámky: - množinové operátory jsou často velmi účinné a zjednodušují složité testy:

Příklad:

```
if (Zn1 = 'V') or (Zn1 = 'S') or (Zn1 = 'Z')
```

může být zapsán zřetelněji:

```
if Zn1 in ['V','S','Z']
```

v Turbo Pascalu maximální počet prvků v jedné množině nesmí být větší než 255. Ordinální čísla bázového typu musí být v intervalu <0,255>.

### 3.7 Typ soubor

Pod pojmem soubor, přesněji snad soubor dat, se běžně rozumí množina dat uložena mimo operační paměť, např. na magnetickém disku.

Soubor může být definovan jako konečná množina číselných nebo nečíselných hodnot stejného typu, které mohou mít složitou strukturu.

Soubor v Pascalu je struktura, která se vytváří množinou složek stejného typu. V určitém okamžiku jen jedna složka je přístupná pro čtení nebo zápis.

Přístup ke složkám souboru může být buď sekvenční, tzn., že přístup k N-té složce vyžaduje, aby se předtím zajistil přístup k N-1 složkám, nebo přímý, tzn., že přístup se zajišťuje přímo podle pořadí složky v souboru.

Vzhledem k tomu, že práce se soubory je v jednotlivých implementacích Pascalu doslova rozdílná, bude dále vysvětlen způsob práce se soubory v Turbo Pascalu. Předem poznamenejme, že práce se soubory v Turbo Pascalu je efektivní a jednodušší než ve standardní definici jazyka.

Turbo Pascal rozlišuje tři druhy souborů:

- typové soubory,
- textové soubory,
- netypové soubory.

Tyto druhy souborů se odlišují typem složek, způsobem uložení na vnějším mediu i přípustnými operacemi.

#### 3.7.1 Typové soubory

Typové soubory jsou určeny pro přímý přístup. Jsou rozšířením proti standardnímu Pascalu.

## DEKLARACE

Definici typového souboru je možno vyjádřit takto:

Typový soubor → File → of → typ →

Obr. 3.15 Syntaktický diagram deklarace typového souboru

typ označuje typ složky. Tento typ může být libovoľný (vyjma typu soubor), ale většinou je to typ záznam nebo typ pole.

### Příklad

#### Type

```
tPlodina = array [1..20] of Real;
tPozemek = record
    Cislo : Integer;
    Nazev : array [1..10] of Char;
    Vymera : Real;
end;
var
    Plodina : File of tPlodina;
    Pozemek : File of tPozemek;
```

Proměnná Plodina označuje soubor, jehož složky jsou typu pole tPlodina. Proměnná Pozemek označuje soubor, jehož složky jsou typu záznam tPozemek. Pozemek a Plodina jsou proměnnými typu soubor. Proměnné typu soubor budeme dále označovat jako PromSou. Tyto proměnné se nemohou používat v přiřazovacích příkazech ani ve výrazech.

## POUŽÍVÁNÍ

Všechny složky typového souboru mají stejnou délku, a proto je možno vypočítat umístění každé složky a nastavit

pozici souboru na požadovanou složku.

Typový soubor je vždy vytvářen jako souvislá posloupnost složek. Turbo Pascal pomocí "ukazovátka" umožňuje vyhledání požadované složky. Pokaždé, když je nějaká složka zapsána nebo přečtena, "ukazovátko" se posune na další složku. Pod pojmem "ukazovátko" se rozumí speciální systémová proměnná Turbo Pascalu, hodnotu této proměnné můžeme v programu určovat.

Při práci se soubory musíme rozlišovat, jestli soubor vytváříme (dosud soubor neexistoval) a nebo zda jej již běžně využíváme, a to jak pro čtení, tak i pro zápis.

Pro práci s typovými soubory využíváme předdefinované podprogramy (procedury a funkce).

#### Předdefinované procedury

##### Assign

Syntaxe: Assign (PromSou,Str);

Zajišťuje přiřazení jména souboru na disku k proměnné PromSou v programu. Str je jméno souboru na disku, které se vytváří podle pravidel operačního systému.

##### Rewrite

Syntaxe: Rewrite (PromSou);

Připravuje nový diskový soubor ke zpracování.

##### Reset

Syntaxe: Reset (PromSou);

Připravuje existující diskový soubor ke zpracování.

### Read

Syntaxe: Read (PromSou,Prom);

Zajišťuje čtení jedné složky Prom do operační paměti.  
Prom označuje složku souboru PromSou.

### Write

Syntaxe: Write (PromSou,Prom);

Zajišťuje zápis jedné složky Prom z operační paměti na disk.  
Prom označuje složku souboru PromSou.

### Seek

Syntaxe: Seek (PromSou,N);

Zajišťuje vyhledání složky v souboru. N je celočíselná proměnná, případně celočíselný výraz. Proměnnou N určujeme hodnotu ukazovátka pro daný soubor. Pro první složku N = 0.

### Flush

Syntaxe: Flush (PromSou);

Přesouvá vyrovnávací paměť souboru PromSou na disk.

### Close

Syntaxe: Close (PromSou);

Uzavírá diskový soubor, který je přiřazen PromSou.

### Předdefinované funkce

#### EOF

Syntaxe: EOF (PromSou);

Vrací hodnotu True, když ukazovátko souboru je na konci diskového souboru. V ostatních případech je hodnota této funkce False.

FilePos

Syntaxe: FilePos (PromSou);

Vrací celočíselnou hodnotu stavu ukazovátka souboru.  
Pro první složku je to 0.

FileSize

Syntaxe: FileSize (PromSou);

Vrací hodnotu, která označuje velikost diskového souboru, tj. počet jeho složek.

Poznámky: - typový soubor musí být nejdříve přiřazen pomocí

Assign a potom musí být otevřen buď pomocí Rewrite (soubor nebyl dosud vytvořen) nebo Reset (soubor je již vytvořen);

- nalezení konce souboru se dá zajistit takto:

Seek (PromSou, FileSize (PromSou));

za konec souboru lze zapisovat další složky;

- každý soubor musí být před ukončením programu uzavřen pomocí procedury Close;

- existují ještě dvě předdefinované procedury, které již nemají tak velký význam:

Erase pro zrušení diskového souboru

a Rename pro přejmenování souboru.

Pro zajištění těchto dvou funkcí se obvykle používají příkazy operačního systému.

Souborný příklad

Dale uvedený celek tří návazných programů vysvětluje možnosti práce s typovým souborem.

Předpokládejme, že určitá STS zajišťuje sklizňové práce pro zemědělské podniky. Dosavadní ruční evidence pomocí karet je převáděna na mikropočítač.

Nejdříve je třeba připravit prostor pro prázdné záznamy (karty) na disku. Pro tento účel byl vytvořen program TvorbaVSTS.

Pro běžné zaznamenávání údajů o průběhu sklizňových prací byl vytvořen program AktualizaceVSTS.

Možnosti využívání takto aktualizovaného typového souboru jsou nastíněny v programu PrumervSTS.

Program TvorbaVSTS, jakož i oba dva návazné programy, jsou rozděleny záměrně do podprogramů, aby se znázornila technika strukturovaného programování.

Uváděný celek programů je dosti náročný a bude jistě srozumitejnější až po prostudování čtvrté kapitoly a oddílu 3.8. (V deklaraci typu tKarta se používá typ řetězec, který je vysvětlen až v oddíle 3.8 ).

Program TvorbaVSTS umožnuje vytvoření prázdného souboru na disku pod označením Smlouva.DTA. Tento soubor bude obsahovat 100 prázdných záznamů.

Poznamenejme, že soubor Smlouva.DTA lze v případě potřeby rozšířit o další záznamy. Do typového souboru lze přidávat další záznamy za dosud existující horní mez, tzn., že jiným programem je možno vytvářet záznamy č. 101, 102 atd.

```
Program TvorbaVSTS;
const
  MaxPocetK = 100;
type
  tKarta = record
    Podnik : String[20];
    Plodina : String[10];
    PlanHA : Integer;
    SkuVykhA : Integer;
    Tuny : Real;
  end;

var
  SouborHS : File of tKarta;
  KartaHS : tKarta;
  I : Integer;

Procedure Otevirani;
Begin
  Assign (SouborHS, 'Smilouva.DTA');
  Rewrite (SouborHS);
end;

Procedure Tvorba;
begin
  with KartaHS do
  begin
    Podnik := 'Pocaply'; Plodina := 'Mata';
    PlanHA := 0; SkuVykhA := 0; Tuny := 0;
    For I := 1 to MaxPocetK do
      Write (SouborHS,KartaHS)
    end
  end;
end;

Procedure Zavirani;
Begin
  Close ( SouborHS );
end;

Begin
  Otevirani; { zde zacina program }
  Tvorba;
  Zavirani;
End.
```

Program AktualizaceVSTS umožňuje průběžné ukládání dat do vytvořeného diskového souboru. Tento program lze spustit dle skutečné potřeby.

```
Program AktualizacevSTS;
const
  MaxPocetKaret = 100;
type
  tKarta = record
    Podnik : String[20];
    Plodina : String[10];
    PlnyHA : Integer;
    SkuVykHA : Integer;
    Tuny      : Real;
  end;

var
  SouborHS : File of tKarta;
  KartaHS : tKarta;

Procedure Otevirani;
begin
  Assign (SouborHS,'Smlouva.DTA');
  Reset (SouborHS);
end;

Procedure Zpracovani;
var
  Ukz, Ha : Integer;
  T : Real;
begin
  ClrScr;
  Write ('Zadej cislo podniku (0 - 100), 0 = Konec ');
  Readln (Ukz);
  While Ukz in [1..MaxPocetKaret] do
  begin
    begin
      Seek (SouborHS,Ukz-1);
      Read (SouborHS,KartaHS);
      with KartaHS do
        begin
          Write('Zadej nazev podniku (',
                Podnik:20,') : ');
          Readln (Podnik);
          Write('Zadej plodinu (,Plodina:10,) : ');
          Readln (Plodina);
          Write('Dalsi sklizeny ha (Dosud',
                SkuVykHa,') : ');
          Readln (Ha);
          SkuVykHa := SkuVykHa + Ha;
          Write('Sklizeny tuny (Dosud',
                Tuny:6:1,') : ');
          Readln(T);
          Tuny := Tuny + T;
        end;
      Seek (SouborHS,Ukz-1);
      Write(SouborHS,KartaHS);
      ClrScr;
      Write
        ('Zadej cislo podniku (0 - 100), 0 = Konec ');
      Readln (Ukz);
    end;
  end;
end;
```

```
Procedure Zavirani;
begin
  Close ( SouborHS )
end;

Begin
  Otevirani;
  Zpracovani;
  Zavirani;
End.
```

Činnost tohoto programu lze dokumentovat takto:

```
Zadej cislo podniku (0 - 100), 0 = Konc i
Zadej nazev podniku (           Slusovice) : Slusovice
Zadej plodinu (Psenice oz)   : Psenice osima
Dalsi sklizene ha (Dosud150 ) : 100
Sklizene tuny (Dosud1235.0 ) : 750
```

#### Ubr. 3.16 Výstup programu AktualizacevSTS

Program PrumervSTS zjišťuje průměrný výnos zadané plodiny v zadáném podniku. V tomto programu se typový soubor KartaHS prohledává sekvenčně.

```
Program PrumervSTS;
const
  MaxPocetK = 100;
type
  tKarta = record
    Podnik : String[20];
    Plodina : String[10];
    PlanHA : Integer;
    SkuVykHA : Integer;
    Tuny     : Real;
  end;

var
  SouborHS : File of tKarta;
  KartaHS : tKarta;
  Podnik : String[20];
  Plodina : String [10];
  Prumer : Real;

Procedure UvodCteniZadani;
begin
  ClrScr;
  Writeln('Program PRUMER zjistuje prumerny vynos');
  Write ('Zadej nazev podniku: ');
  Readln( Podnik );
  Write ('Zadej nazev plodiny: ');
  Readln (Plodina)
```

```
end;

Procedure Otevirani;
begin
  Assign (SouborHS,'Smlouva.DTA');
  Reset (SouborHS);
end;

Procedure HledaniVypocet;
var
  I : Integer;
begin
  for i := 1 to MaxPocetK do
    begin
      Read (SouborHS,KartaHS);
      if ( KartaHS.Podnik = Podnik ) and
          ( KartaHS.Plodina = Plodina ) then
        begin
          Prumer := KartaHS.Tuny / KartaHS.SkuVykhA;
          Write ('Prumerny vynos je : ',Prumer:5:2,
                 ' tun');
          Exit;
        end;
      end;
      Write('Hledane zadani neexistuje');
    end;

Procedure Zavirani;
begin
  Close ( SouborHS )
end;

Begin
  Otevirani;                                { zde zacina program }
  UvodCteniZadani;
  HledaniVypocet;
  Zavirani;
End.
```

Činnost programu PrumervSTS je možno dokumentovat takto:

Program PRUMER zjistuje prumerny vynos  
Zadej nazev podniku: Blusovice  
Zadej nazev plodiny: Psenice ozima  
Prumerny vynos je : 7.94 tun

Obr. 3.17 Výstup programu PrumervSTS

### 3.7.2 Textové soubory

Textové soubory jsou určeny pro práci s textem. Textové soubory lze zpracovávat jen sekvenčně.

#### DEKLARACE

Definici textového souboru je možno vyjádřit takto:

Textový soubor → Text →

Obr. 3.18 Syntaktický diagram deklarace textového souboru

Slovo Text je identifikátor standardního typu (viz oddíl 2.1.4).

#### Příklad

```
var
    Skripta : Text;
    Pokyny : Text;
```

#### POUŽÍVÁNÍ

Textové soubory jsou vytvářeny posloupností hodnot stejného typu. Základními složkami textového souboru jsou znaky, které jsou strukturovány do řádků, přičemž každý řádek je ukončen speciální značkou konce řádku (EOL marker = posloupnost znaků CR/LF). Soubor je ukončen značkou pro konec souboru (EOF marker = znak Ctrl-Z). Vzhledem k tomu, že délka řádku může být různá, pozice řádku nemůže být

vypočtena.

Pro práci s typovými soubory využíváme předdefinované podprogramy (procedury a funkce).

### Předdefinované procedury

#### Assign

Syntaxe: Assign (PromSou,Str);

Zajišťuje přiřazení jména souboru na disku k proměnné PromSou v programu. Str je jméno souboru na disku, které se vytváří podle pravidel operačního systému.

#### Rewrite

Syntaxe: Rewrite (PromSou);

Připravuje nový diskový soubor ke zpracování.

#### Reset

Syntaxe: Reset (PromSou);

Připravuje existující diskový soubor ke zpracování.

#### Read

Syntaxe: Read (PromSou,Prom);

Zajišťuje čtení jedné složky Prom do operační paměti. Prom označuje složku souboru PromSou = řádek.

#### Readln

Syntaxe: Readln (PromSou,Prom);

Zajišťuje čtení jedné složky Prom (řádku) do operační paměti, včetně značky pro konec řádku.

### Write

Syntaxe: Write (PromSou,Prom);

Zajišťuje zápis jedné složky Prom z operační paměti na disk. Prom označuje složku souboru PromSou = řádek.

### Writeln

Syntaxe: Writeln (PromSou,Prom)

Zajišťuje zápis jedné složky Prom (řádku) z operační paměti na disk, včetně značky pro konec řádku.

### Close

Syntaxe: Close (PromSou);

Uzavírá diskový soubor, který je přiřazen PromSou.

## Předdefinované funkce

### EOF

Syntaxe: EOF (PromSou);

Vrací hodnotu True, když byl nalezen konec diskového souboru. V ostatních případech je hodnota této funkce False.

### Eoln

Syntaxe: Eoln (PromSou);

Vrací hodnotu True, když byl nalezen konec běžného řádku (tj. znak CR). Je-li EOF (PromSou) True, potom Eoln (PromSou) je také True.

### SeekEoln

Syntaxe: SeekEoln (PromSOU);

Vrací hodnotu True, když byl nalezen konec běžného řádku. Obdobná funkce jako Eoln, výjimka spočívá v tom, že před provedením testu jsou vynechány všechny mezery a znaky TAB.

### SeekEof

Syntaxe: SeekEof (PromSOU);

Vrací hodnotu True, když byl nalezen konec diskového souboru. Obdobná funkce jako EOF, výjimka spočívá v tom, že před provedením testu jsou vynechány všechny mezery a znaky TAB.

Poznámky:

- textový soubor musí být nejdříve přiřazen pomocí Assign a potom musí být otevřen buď pomocí Rewrite (soubor nebyl dosud vytvořen) nebo Reset (soubor je již vytvořen);
- každý soubor musí být před ukončením programu uzavřen pomocí procedury Close;
- lze také použít předdefinované procedury Erase pro zrušení diskového souboru a Rename pro přejmenování souboru.

### Příklad

Dále uvedený příklad vypisuje jakýkoliv soubor z disku (např. program v Pascalu) na tiskárnu. Rozšíření příkazu Writeln je uvedeno v následujícím odstavci Logická zařízení. V programu se využívá direktiva kompilátoru I- pro testování přítomnosti souboru na disku (viz oddíl 7.4.2).

```
Program Textovysoubor;
var
  TextSoubor : Text;
  JmenoSou : String [12];
  Radek : String [80];

Begin
  ClrScr;
  Write ('Zadej jmeno souboru: ');
  Readln (JmenoSou);
  Assign (TextSoubor, JmenoSou);
  {$I-}
  Reset (TextSoubor);
  {$I+}
  if IOResult <> 0 then
    Writeln ('Soubor ', JmenoSou, ' nebyl nalezen')
  else
    begin
      while not Eof (TextSoubor) do
        begin
          Readln (TextSoubor, Radek);
          Writeln (Lst, Radek)
        end;
      { koniec While }
      Writeln (Lst)
    end
  end.
end.
```

### Logická zařízení

V Turbo Pascalu se na různá fyzická zařízení počítače, jako jsou terminály a tiskárny, díváme jako na tzv. logická zařízení. S logickými zařízeními pracujeme jako s textovými soubory.

K disposici jsou tato logická zařízení:

CON: Zařízení vstupu/výstupu (klávesnice/obrazovka), které využívá vyrovnávací paměť.

TRM: Zařízení vstupu/výstupu (klávesnice/obrazovka), které využívá vyrovnávací paměť.

KBD: Zařízení vstupu (klávesnice). Vstupující znaky nejsou zobrazovány na obrazovce.

LST: Zařízení výstupu (tiskárna).

AUX: Pomocné zařízení (např. snímač/děrovač děrné

pásky).

USR: Uživatelská zařízení, která je možno definovat v řidícím programu.

Logická zařízení se mohou využívat jako tzv. standardní soubory nebo je lze přiřadit proměnným typu soubor.

#### Standardní soubory

Programátor může využívat řadu předdeklarovaných souborů, které byly předem přiřazeny logickým zařízením a tak připraveny pro zpracování. Programátor tyto soubory nemusí přiřazovat, otevírat ani zavírat. Použití procedur Assign, Reset, Rewrite a Close je zbytečné a nepřípustné.

K dispozici jsou tyto standardní soubory:

#### Input

Vstupní soubor, který je přiřazen zařízení CON: nebo TRM:

#### Output

Výstupní soubor, který je přiřazen zařízení CON: nebo TRM:

Con soubor přiřazený zařízení CON:

Trm soubor přiřazený zařízení TRM:

Kbd soubor přiřazený zařízení KBD:

Lst soubor přiřazený zařízení LST:

Aux soubor přiřazený zařízení AUX:

User soubor přiřazený zařízení USR:

Když procedura Read je použita bez označení jména souboru, potom pracujeme se standardním souborem Input. Tzn., že

zápis Read (vi, ... , vn)

je identický

Read (Input, vi, ... , vn).

Proto u všech předdeklarovaných procedur vstupu/výstupu, které byly uvedeny v oddíle 2.5.3, je možno jako první parametr uvádět jmeno standardního souboru. Tzn., že u procedur Read a Readln je to standardní soubor Input, u procedur Write a Writeln je to soubor Output.

U vstupní procedury Read je někdy výhodné použít standardní soubor Kbd. Zapisované znaky nejsou viditelné a tak můžeme ošetřovat vstup speciálního hesla pro určitého uživatele.

U výstupní procedury Write používáme standardní soubor Lst při výstupu na tiskárnu (viz minulý příklad v tomto oddílu).

### 3.7.3 Netypové soubory

Netypové soubory jsou určeny pro přímý přístup k diskovým souborům, přičemž délka záznamů je vždy 128 B.

### DEKLARACE

Definici netypového souboru je možno vyjádřit takto:

Netypový soubor → File →

Obr. 3.19 Syntaktický diagram deklarace netypového  
souboru

Slovo File je rezervované slovo jazyka Pascal.

### Příklad

```
var  
    DSoubor : File;  
    Zdroj, Cil : File;
```

### POUŽÍVÁNÍ

Při vstupních a výstupních operacích pro netypové soubory jsou data přenášena přímo mezi diskem a proměnnou a tak se šetří prostor pro vyrovnávací paměť, který je vyžadován pro typové soubory. Protože netypové soubory jsou kompatibilní s jinými typy souborů, je výhodné používat netypové soubory tehdy, když není požadována operace vstupu/výstupu (náprkř. pro přejmenování souboru).

Pro netypové soubory lze používat všechny procedury a funkce pro typové soubory vyjma Read, Write a Flush. Procedury Read a Write jsou nahrazeny vysoko účinnými přenosovými procedurami: BlockRead a BlockWrite. Syntaxe pro vyvolávání těchto procedur je tato:

```
BlockRead (PromSou, Prom, NZaz)
BlockWrite (PromSou, Prom, NZaz)

nebo

BlockRead (PromSou, Prom, NZaz) Vysledek)
BlockWrite (PromSou, Prom, NZaz, Vysledek)
```

kde PromSou je identifikátor proměnné pro netypový soubor, Prom je libovolná proměnná a NZaz je celočíselný výraz, který určuje počet záznamů přenášených mezi diskovým souborem a proměnnou. Volitelný parametr Výsledek vraci počet právě přenesených záznamů.

#### Příklad

Dále uvedený program umožňuje zkopirovat libovolný soubor na disku.

```
Program KopirovaniSouboru;
Const
  DelkaZaz = 128;
  DelkaBuf = 200;
var
  Zdroj, cil : File;
  ZdrojJmeno, CilJmeno : String [12];
  Buffer : array[1..DelkaZaz,1..DelkaBuf] of Byte;
  NZaz : Integer;

Begin
  ClrScr;
  Write('Zadej jmeno zdrojoveho souboru: ');
  Readln (ZdrojJmeno);
  Assign (Zdroj, ZdrojJmeno);
  Reset (Zdroj);
  Write('Zadej jmeno ciloveho souboru: ');
  Readln (CilJmeno);
  Assign (Cil, CilJmeno);
  Rewrite (Cil);
  repeat
    BlockRead(Zdroj, Buffer, DelkaBuf, NZaz);
    BlockWrite(Cil, Buffer, NZaz);
  until NZaz = 0;
  Close (Zdroj);
  Close (Cil);
End.
```

### Kontrola vstupu/výstupu

Direktiva kompilátoru I umožňuje ošetření chyb vstupu/výstupu. Když je tato direktiva pasivní, {\$I-}, tak programátor může stav po vykonání vstupní/výstupní operace testovat pomocí standardní funkce IOresult, která je typu Boolean.

### Příklad

```
{ procedura Exist testuje prítomnost souboru na disku. }
function Exist(FileN:AnyStr):boolean;
var F:File;
begin
  {$I-}
  assign(F,FileN);
  reset(F);
  {$I+}
  if IOResult <> 0 then
    Exist:=false
  else
    Exist:=true;
  end;
```

Funkce Exist dává výsledek True, když soubor zadaného jména se nachází na disku. Pokud soubor zadaného jména neexistuje, tak se vrací hodnota False. Nutným předpokladem pro použití této funkce je, aby v globální deklaraci typů byl definován typ AnyStr, např. takto:

Type

```
AnyStr : String [80];
```

Funkci	IOresult	Ize	použít	pro	tyto	standardní
procedury:						
Assign	BlockRead	BlockWrite	Chain	Close		
Erase	Execute	Flush	Read	Readln		
Rename	Reset	Rewrite	Seek	Write		
	WriteIn					

### 3.8 Typ řetězec

Pro usnadnění zpracování úloh, ve kterých se pracuje s řetězci znaků, je v Turbo Pascalu strukturovaný typ řetězec (angl. String). Tento typ je dosti podobný typu pole, ve kterém složky pole jsou znaky (array of Char). Hlavní rozdíl spočívá v tom, že počet znaků v řetězci (tzv. délka řetězce) je proměnlivý, kdežto počet prvků v poli je stálý.

Typ řetězec není standardní, ale vyskytuje se téměř ve všech verzích Pascalu. V Turbo Pascalu je široce rozpracován a pro programátora je k disposici řada učinných podprogramů.

#### 3.8.1 Deklarace

Definici typu řetězce je možno zapsat takto:

Typ řetězec → String → [ → počet znaků → ] →

Obr. 3.20 Syntaktický diagram deklarace typu řetězec

počet znaků označuje maximální délku pro vytvářený řetězec. Délka řetězce se pohybuje v intervalu 1 až 255.

#### Příklad

##### Type

```
tStr3 : String [3];  
tJmeno : String [10];  
tRadek : String [80];  
AnyStr : String [255];
```

- Poznámky:
- typ řetězec nemá implicitní délku, délka musí být vždy uvedena;
  - proměnná typu řetězec zabírá definovanou délku plus jeden byte, ve kterém je údaj o běžné délce. Délka řetězce je umístěna v nultém byte řetězce. Jednotlivé znaky jsou jsou indexovány od jedné do zadáné délky.

### 3.8.2 Používání

Přistup k proměnným typům řetězec je stejný jako u jednorozměrných polí. S jednotlivými složkami řetězce pracujeme stejně jako s indexovanými proměnnými.

Při práci s řetězci se používají operace přiřazení, sčítání, porovnávání a některé standardní funkce a procedury.

#### Přiřazení

Pomocí přiřazovacího příkazu se přiřazuje hodnota výrazu proměnného typu řetězec.

#### Příklad

```
var  
    Jmeno      : tJmeno;  
    Radek     : tRadek;  
    Zkratka1, Zkratka2 : tStr3;  
  
begin  
    Jmeno    := 'Jana';  
    Zkratka1 := 'VSZ';  
    Zkratka2 := 'KISVP';
```

Jestliže maximální délka řetězce je překročena, potom přebytečné znaky jsou odříznuty. Tzn., že proměnná Zkratka2 obsahuje jen znak 'KIS'.

### Sčítání

Při práci s řetězci je možno používat operátor sčítání

+

### Příklad

Uvažujme, že platí predchozí deklarace, potom lze zapsat:

```
Radek := 'Vse nejlepsi ' + Jmeno;
```

Operace sčítání může být nahrazena funkcí Concat.

### Porovnávání

Při manipulaci s řetězci se používají již definované relační operátory: =, <>, <, >, <= a >=. Při porovnávání dvou řetězců se porovnávají jednotlivé znaky v obou řetězcích .leva doprava. Výsledkem porovnávání je hodnota Boolean.

### Příklad

porovnávání	výsledek
Jmeno = 'Jana'	True
Jmeno = 'JANA'	False
'A' < 'B'	True
'A' < 'a'	True

Řetězce se porovnávají podle ordinálních čísel jednotlivých znaků.

### Přededefinované procedury

#### Delete

Syntaxe: Delete (St, Poz, N);

Ruší část řetězce o N znacích z proměnné St od pozice Poz. Proměnná St je typu řetězec, Poz a N jsou celočíselné výrazy.

#### Příklad

Jestliže St obsahuje hodnotu 'ABCDEFG', potom:

Delete (St,2,4) způsobí,

že v St je hodnota 'AFG';

Delete (St,3,2) způsobí,

že v St je hodnota 'ABCfg'.

#### Insert

Syntaxe: Insert (Obj, Cil, Poz);

Vkládá řetězec Obj do řetězce Cil od pozice Poz. Proměnné Obj a Cil jsou typu řetězec, Poz je celočíselný výraz.

#### Příklad

Jestliže St obsahuje hodnotu 'ABCDE', potom:

Insert ('XX',St,2) způsobí,

že v St je hodnota 'AXXBCDE'.

#### Str

Syntaxe: Str (Hodnota,St);

Převádí číselnou hodnotu Hodnota do řetězce St. Hodnota se zapisuje jako parametr výstupu pro typ Integer nebo typ Real v proceduře Write (viz oddíl 2.5.3). St je proměnná typu řetězec.

Příklad

Jestliže I obsahuje hodnotu 1234, potom:

Str (I:5,St) způsobí,

že v St je hodnota ' 1234 '.

Jestliže X obsahuje hodnotu 3.14, potom:

Str (X:5:2,St) způsobí,

že v St je hodnota ' 3.14 '.

Val

Syntaxe: Val (St, Prom, Kod);

Převádí řetězec St na hodnotu typu Real nebo Integer (v závislosti na typu proměnné Prom) a tuto hodnotu ukládá do Prom. Prom musí být proměnná typu Integer nebo Real, Kod je proměnná typu Integer. Jestliže se převod uskuteční bezchybně, potom v proměnné Kod je hodnota 0. V případě chybného převodu proměnná Kod obsahuje první pozici, kde byla zjištěna chyba; v takovémto případě hodnota Prom je nedefinována.

Příklad

Jestliže St obsahuje hodnotu '234', potom:

Val (St,I,Kod) způsobí,

že v I je hodnota 234 a Kod = 0.

Jestliže St obsahuje hodnotu '12a6', potom:

Val (St,I,Kod) způsobí,

že hodnota proměnné I není

definována a Kod = 3.

Jestliže St obsahuje hodnotu '3.1', potom:

Val (St,X,Kod) způsobí,

že v X je hodnota 3.1 a Kod = 0.

### Předdefinované funkce

#### Copy

Syntaxe: Copy (St, Poz, N);

Vrací část řetězce o N znacích, které jsou získány z řetězce St od pozice Poz. Proměnné Poz a N jsou celočíselné výrazy.

#### Příklad

Jestliže St obsahuje hodnotu 'ABCDEFG', potom:

Copy (St,2,4) vrací hodnotu 'BCDE';

Copy (St,3,2) vrací hodnotu 'CD'.

#### Concat

Syntaxe: Concat (St1, St2, { ,StN } );

Vrací řetězec, který vzniká sloučením několika jiných řetězců. Jednotlivé řetězce se v zápisu oddělují čárkami. Funkce Concat má stejný význam jako operátor +.

#### Příklad

Jestliže St1 obsahuje hodnotu 'System'

a St2 hodnotu 'repky', potom:

Concat (St1, ' výroby ', St2) vrací hodnotu

'System výroby repky'.

#### Length

Syntaxe: Length (St);

Vrací délku řetězce St, tj. počet znaků umístěných v řetězci. Výsledek je typu Integer.

#### Příklad

Jestliže St obsahuje hodnotu 'ABCDEFG', potom:

Length (St) vrací hodnotu 7.

### Pos

Syntaxe: Pos (Obj, Cil);

Vrací hodnotu prvního výskytu řetězce Obj v řetězci Cil. Proměnné Obj a Cil jsou typu řetězec. Výsledek je typu Integer. Pokud se řetězec Obj v řetězci Cil nevyskytuje, potom funkce Pos vrací hodnotu 0.

### Příklad

Jestliže St obsahuje hodnotu 'ABCDEF6', potom:

Pos ('DE',St) vrací hodnotu 4;

Pos ('X', St) vrací hodnotu 0.

Poznámky: - typy řetězec a Char jsou kompatibilní;

- první znak řetězce obsahuje jeho délku. Zapis Length (St) je ekvivalentní jak Ord (St[0]);
- překročení přípustné délky řetězce lze testovat pomocí direktivy R+ (viz oddíl 7.4.2).

### 3.8.3 Souhrnný příklad

Dále uvedený program využívá několik podprogramů, ve kterých se pracuje s řetězci znaků.

Použití funkce InputReal umožňuje čtení čísel typu Real. Pro vyvolání této funkce je třeba zadat počet pozic čteného čísla, souřadnice obrazovky X,Y (tj. určit pozici, odkud číslo bude čteno) a přípustný rozsah čteného čísla. Vzhledem k tomu, že se jedná asi o nejnáročnější příklad v tomto učebním textu, doporučujeme jednotlivé podprogramy postupně vyzkoušet.

```
Program TestCteni;
type
  Anystr = String [20];
var
```

```
Cislo : Real;
L,X,Y : Integer;
Dolni, Horni : Real;

(* ConstStr vraci retezec o N znacich,
ktere maji hodnotu C *)
function ConstStr(C : Char; N : Integer) : AnyStr;
var
  S : AnyStr;
begin
  if N < 0 then
    N := 0;
  S[0] := Chr(N);
  FillChar(S[1],N,C);
  ConstStr := S;
end;

(* Beep spusti zvukovy signal *)
procedure Beep;
begin
  Write(^G);
end;

(* InputStr cte libovolny retezec znaku *)
procedure InputStr(var S : AnyStr;
                    L,X,Y : Integer );
const
  Podtrzeni = '_';
var
  P : Integer;
  Ch : Char;
begin
  GotoXY(X + 1,Y + 1);
  Write(S,ConstStr(Podtrzeni,L - Length(S)));
  P := 0;
  repeat
    GotoXY(X + P + 1,Y + 1);
    Read(Kbd,Ch);
    case Ch of
      #32..#126 : if P < L then
        begin
          if Length(S) = L then
            Delete(S,L,1);
          P := P + 1;
          Insert(Ch,S,P);
          Write(Copy(S,P,L));
        end
        else Beep;
      ^H,#127 : if P > 0 then { test na opravu }
        begin
          Delete(S,P,1);
          Write(^H,Copy(S,P,L),Podtrzeni);
          P := P - 1;
        end
        else Beep;
    else
      if Ch <> ^M then Beep;
    end
  until Ch = Podtrzeni;
end;
```

```
        end; { konec case }
      until Ch = ^M;
      P := Length(S);
      GotoXY(X + P + 1, Y + 1);
      Write(' ' :L - P);
    end;

(* InputReal umožnuje čtení čísla typu Real *)
Function InputReal ( L,X,Y : Integer;
                      Dolni, Horni : Real ) : Real;
var
  Pom   :AnyStr;
  OK    :Boolean;
  Kod   :Integer;
  Cc    :Real;
begin
  OK := false;
  repeat
    FillChar(Pom,SizeOf(pom),0);
    InputStr(Pom,L,X,Y);
    Val(Pom,Cc,Kod);
    if kod <> 0 then
      begin
        GotoXY(X+1,Y+2);
        Write ('neni zadano cislo');
      end
    else
      begin
        if ( Cc < Dolni ) or (Cc > Horni ) then
          begin
            GotoXY(X+1,Y+2);
            Write('Nesprávny format');
          end
        else
          begin
            InputReal := Cc;
            GotoXY(X+1,Y+2); ClrEol;
            OK := True;
          end;
      end;
    until OK ;
  end;

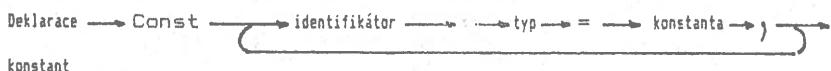
Begin      { Začátek hlavního programu }
  ClrScr;
  X := 10;
  Y := 5;
  Dolni := 10;
  Horni := 1000;
  L := 5;
  repeat
    Cislo := InputReal ( L,X,Y, Dolni, Horni );
    Y := Y + 1;
  until Cislo = 77.7;
End.
```

### 3.9 Typové konstanty

Typové konstanty jsou specifikou Turbo Pascalu. Typová konstanta se používá obdobně jako proměnná stejného typu. Typové konstanty se používají jako "inicializované proměnné", neboť hodnoty typových konstant jsou definovány. Poznamenejme, že počáteční hodnoty proměnných před prvním přiřazením nejsou definovány.

#### 3.9.1 Deklarace

Deklarace typových konstant se uskutečňuje také v úseku deklarace konstant, který začíná rezervovaným slovem "Const" (viz oddíl 2.3.3). Syntaxe tohoto úseku pro typové konstanty je tato:



Obr. 3.18 Syntaktický diagram úseku deklarace konstant pro typové konstanty

V tomto případě typ musí být znám, a proto úsek deklarace konstant musí předcházet úsek deklarace typů.

Zde se setkáváme s výjimkou vzhledem k definici deklarační části (viz obr. 3.1). V Turbo Pascalu pořadí jednotlivých deklaračních úseků je libovolné.

#### Nestrukturované typové konstanty

Nestrukturovaná typová konstanta je konstanta

definovaná pro jednoduchý typ, případně typ řetězec.

Příklad

Const

PocetSt : Integer = 218;

PrumTep : Real = 9.5;

Jmeno : String[4] = 'Jana';

KonZnak : Char = ^M;

S typovými konstantami pracujeme stejně jako s proměnnými, můžeme je využívat při volání podprogramů jako skutečné parametry pro přenos odkazem. Připomeňme, že pro tento účel nelze použít netypové (normální) konstanty.

Vzhledem k tomu, že typová konstanta je proměnná s konstantní hodnotou, nelze ji použít pro definici jiných konstant či typů. Dále uvedená definice typu tVektor je nesprávná:

Příklad

Const

Min : Integer = 1;

Max : Integer = 80;

Type

tVektor : array[Min..Max] of Integer;

Strukturované typové konstanty

Strukturované typové konstanty umožňují přiřadit počáteční hodnoty strukturovaným typům jako je pole, záznam a množina. Vzhledem k tomu, že typové konstanty dosti modifikují předchozí výklad, budou uvedeny jen dva příklady, a sice pro jednorozměrné pole a pro množinu.

Příklad

```
Const  
    Cislice : array[0..9] of Char =  
        ('0','1','2','3','4','5','6','7','8','9');
```

lze zapsat jednodušeji takto:

```
Const  
    Cislice : array[0..9] of Char = '0123456789';
```

Příklad

```
type  
    Velka = set of 'A'..'Z';  
    Mala = set of 'a'..'z';  
  
Const  
    Samohlasky : Mala = ['a','e','i','o','u','y'];  
    Oddelovace : set of Char = [' ', '.', ',', '?', '!']
```

3.10 Identita a kompatibilita typů

Na závěr kapitoly o typech bude upřesněn především pojem kompatibilita.

Výklad v tomto oddílu vychází z lit. [7] a platí pro Pascal obecně.

Identita

Dva objekty A a B jsou stejného typu, když se deklaruji jedním z těchto tří způsobů:

- a) var A,B : T;
- b) var A : T;  
 B : T;
- c) type Ti = ...;

```
T2 = T1;  
var  
  A : T1;  
  B : T2;
```

### Kompatibilita

Dva objekty A a B jsou považovány za kompatibilní typy, když jejich deklarace odpovídají jedné z těchto situací:

a) Typ objektu A je identický s typem objektu B

b) Typ objektu A je interval typu objektu B

nebo

typ objektu B je interval typu objektu A

nebo

typ objektu A a typ objektu B jsou intervaly stejného typu.

### Kompatibilita pro přiřazení

Hodnota B je kompatibilní pro přiřazení proměnné A v těchto případech:

- A a B jsou stejného typu, ale ne typu soubor nebo strukturovaného typu, který obsahuje složku typu soubor;
- A je typu Real a B je typu Integer;
- A a B jsou kompatibilní typy a hodnota B je z intervalu, který je popsán typem A;
- A a B jsou typu množina a všechny prvky B jsou z intervalu, který je popsán bázovým typem množiny A;
- A a B jsou dva kompatibilní typy řetězců.

## 4. Podprogramy

### 4.1 Pojem podprogram

Jeden z největších problémů při vytváření programů spočívá ve velkém množství detailů, které je třeba neustále zvažovat; čím více se toto množství zvyšuje, tím se zvyšuje i riziko chyb a omylů. Jedna z cest, jak zlepšit a usnadnit koncepci programů a současně přispět ke zvýšení spolehlivosti vytvářeného programu, spočívá v redukci tohoto velkého množství detailů pomocí dekompozice.

Tento přístup umožnuje dekomponovat úlohu, která se má programovat, na dílčí úlohy, přičemž se specifikuje jejich poradí při vykonávání a podmínky pro jejich využívání. Následující etapy spočívají v postupné dekompozici těchto dílčích úloh na jiné dílčí úlohy až do dosažení žádané úrovně detailů, které je možno bez problému zvažovat. Tento postup se označuje jako metoda shora-dolů (angl. Top-down design).

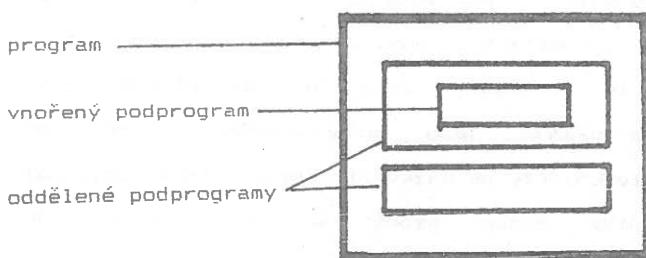
Metoda shora-dolů umožňuje oddělit realizaci každé dílčí úlohy od jiných dílčích úloh. Tato metoda, mimo výhody v modulární koncepci, přináší vyšší kvalitu do programování (v daném okamžiku je řešena jen jedna dílčí úloha) a usnadňuje údržbu hotových programů, neboť dílčí úloha je snadněji modifikovatelná.

V Pascalu se pro usnadnění konstrukce programů využívají podprogramy. Vypracování programu pro určitou úlohu spočívá v tvorbě tolika podprogramů, kolik je rozdílných dílčích úloh.

Je pravda, že pojem podprogram je starší než myšlenky

uvedené v záhlaví oddílu. Např. když určitá sekvence příkazů, která popisuje složitou operaci, se opakuje v programu několikrát, potom tato sekvence se zapisuje jako podprogram.

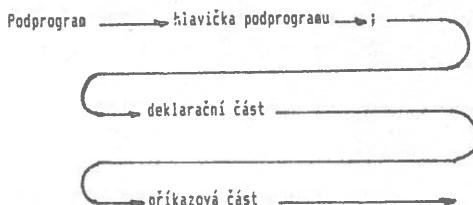
Metoda shora-dolů se využívá při dekompozici složitého problému na dílčí celky, které jsou jednodušší a jednodušší.



Obr. 4.1 Struktura programu a podprogramů

#### 4.2 Deklarace a vyvolávání podprogramů

Obecná deklarace podprogramů se dá vyjádřit takto:



Obr. 4.2 Syntaktický diagram deklarace podprogramu

Deklarace podprogramů se zajišťují v úseku deklarace podprogramů, který se uvádí jako poslední v deklarační části programu nebo podprogramu.

Podprogram (program), který obsahuje v příkazové části bod k vyvolávání podprogramu, se nazývá "volající", a podprogram, který se bude po vyvolávání zpracovávat, se nazývá "volaný".

V podprogramu, v jeho definiční části, lze deklarovat objekty (návěští, konstanty, typy, proměnné a podprogramy), které jsou mu vlastní, tzn., že tyto objekty jsou nepřístupné programům nebo podprogramům, které ho vyvolávají. Tyto objekty se nazývají lokální. Tento pojem má v Pascalu velký význam, neboť umožnuje v podprogramu izolovat jak příkazy určené pro zajištění požadované akce, tak také objekty, které jsou specifické a jejichž deklarace v programu (nebo volajícím podprogramu) by "zatěžovala" popis objektů programu. Lokální deklarace přispívají také k zlepšení čitelnosti podprogramu, tak i k ochraně proti nežádoucímu používání.

Vyvolání podprogramu zahajuje vykonávání požadovaného podprogramu a komunikaci mezi podprogramy "volající" a "volaný", pomocí níž jsou informace vyměňovány. Komunikace je platná po celou dobu vykonávání.

Podle formy této komunikace se rozlišují dva typy podprogramů, tj. procedura a funkce:

- procedura vykonává určitou akci, vykonávání této akce je spouštěno pomocí příkazu procedury;
- funkce přináší jeden výsledek a bod jejího vyvolávání se vždy objevuje ve výrazu.

#### 4.2.1 Procedura

Hlavičku deklarace procedury lze vyjádřit takto:

Hlavička → Procedure → identifikátor → seznam parametru →  
procedury

Obr. 4.3 Syntaktický diagram pro hlavičku procedury

Deklařační i příkazová část procedury mají stejnou strukturu jako v programu.

#### Příklad

```
Program Tisk;
var
    I : Integer;

Procedure Tiskni;
var
    J : Integer;
begin
    J := 100;
    writeln ('Zde je podprogram Tiskni, J= ', J);
end;

Begin          (* zacatek hlavního programu *)
    I := 10;
    Writeln ('Zde je hlavní program, I= ',I);
    Tiskni;
    I := 0;
    Writeln ('Zde je opet hl. program, I= ',I)
End.
```

V tomto příkladu program Tisk obsahuje v deklarační části deklaraci proměnné I a procedury Tiskni. Procedura Tiskni obsahuje deklaraci proměnné J. Vyvolání procedury Tiskni se zajišťuje v programu příkazem Tiskni. Program Tisk dává tyto výsledky:

Zde je hlavní program, I= 10
Zde je podprogram Tiskni, J= 100
Zde je opet hl. program, I= 0

Obr. 4.4 Výstup programu Tisk

#### 4.2.2 Funkce

Hlavičku deklarace funkce lze vyjádřit takto:



Obr. 4.5 Syntaktický diagram pro hlavičku funkce

Typ výsledku označuje typ výsledku funkce, může to být jednoduchá proměnná nebo proměnná typu ukazatel.

Jak vyplývá z následujícího příkladu, v příkazové části deklarace funkce se musí alespoň jednou vyskytnout jméno funkce vlevo v přiřazovacím příkazu, aby mohl být předán výsledek do "volajícího" podprogramu.

#### Příklad

```
Program TiskF;
  var
    I : Integer;

  Function Konstanta : Integer;      { hlavicka funkce }
    begin
      konstanta := 5;
    end;

  Procedure Tiskni;                  { hlavicka procedury }
    var
      J : Integer;
    begin
      J := 100;
      Writeln ('konstanta * 2 = ', Konstanta*2)
    end;

Begin                                    { zacatek hl. progr }
  I := Konstanta + 2;
  Writeln (' I = ', I);
  Tiskni;
  if konstanta * 3 < 25 then
    Writeln ('Konstanta * 3 = ', Konstanta*3)
End.
```

Funkce konstanta předává výsledek typu Integer. Příkazová část programu obsahuje tři odkazy na funkci Konstanta, tzn., že funkce Konstanta je 3x vyvolávána z programu. Jednou je tato funkce vyvolávána z procedury Tiskni. V tomto programu se funkce Konstanta vykonává celkem 4x. Program přináší tyto výsledky:

```
I = 7  
konstanta + 2 = 10  
Konstanta * 3 = 15
```

Obr. 4.6 Výstup programu TiskF

### 4.3 Parametry

#### 4.3.1 Úvod

Způsob komunikace mezi "volajícím" a "volaným" podprogramem se zajišťuje pomocí parametrů. Rozlišujeme dva typy parametrů:

- formální parametry: jsou to identifikátory, které se objevují v seznamu parametrů v hlavičce procedury nebo funkce. Tyto parametry jsou používány v těle procedury nebo funkce a určují typ a způsob přenosu přenášených objektů;
- skutečné parametry: jsou to objekty, které jsou přenášeny během volání od "volajícího" podprogramu k "volanému" podprogramu. Tyto parametry jsou během volání nahrazeny formálními parametry.

Formální parametry jsou lokálními objekty podprogramu, tzn., že nejsou přístupné z volávajícího programu nebo podprogramu.

Příklad

```
Program Konverze;
var
    Cas : Integer;

Procedure Konvertuj (Trvani : Integer);
var
    Hod, Min, Sec : Integer;
begin
    Hod := Trvani Div 3600;
    Min := (Trvani - Hod * 3600) Div 60;
    Sec := Trvani - (Hod * 3600 + Min * 60);
    Writeln (' Trvani je: ',Hod, ' hodin ',Min,
             ' minut ', Sec,' sekund')
end;

Begin
    ClrScr;
    Writeln ('Program na konverzi casu');
    Writeln ('Zadej cas v sekundach,
              0 znamena konec programu');
    Write ('Cas: ');
    Read (Cas);
    While Cas <> 0 do
    begin
        Konvertuj (Cas);
        Write ('Cas: ');
        Readln (Cas)
    end;
    Write (' Konec programu')
End.
```

Pomoci programu Konverze lze získat tyto výsledky:

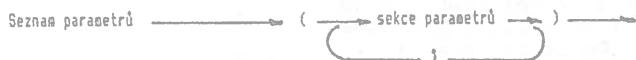
```
Program na konverzi casu
Zadej cas v sek., 0 znamena konec programu
Cas: 333 Trvani je: 0 hodin 5 minut 33 sekund
Cas: 1000
Trvani je: 0 hodin 16 minut 40 sekund
Cas: 0
Konec programu
```

Obr. 4.7 Výstup programu Konverze

Mezi podprogramem "volajícím" a podprogramem "volaným" lze přenášet různý počet informací. Existují čtyři typy informací, které lze přenášet:

- hodnoty: potom se hovoří o přenosu hodnotou;
- odkazy na proměnné: potom se hovoří o přenosu odkazem;
- podprogramy: potom se hovoří o přenosu procedury nebo o přenosu funkce;
- proměnné typu pole, jehož rozměry nejsou určeny při zápisu podprogramu.

Obecná forma seznamu parametrů je tato:



Obr. 4.8 Syntaktický diagram pro seznam parametrů

Sekce parametrů má speciální syntaxi v závislosti na povaze parametrů a na způsobu přenosu.

#### 4.3.2 Přenos hodnotou

Jedná se o nejjednodušší způsob přenosu. Hodnoty mohou být jak jednoduchého, tak strukturovaného typu. V okamžiku volání podprogramu formálním parametrům jsou přiřazeny hodnoty odpovídajících skutečných parametrů.

Syntaxe sekce parametrů je tato:



Obr. 4.9 Syntaktický diagram pro sekci parametrů

při přenosu hodnotou

Tento způsob přenosu byl použit v programu Konverze.

Příklad

```
Program PocetZn;
Var
  Veta      : String [80];
  DelkaVety : Integer;
  HlZnak    : Char;

Procedure Hledej (Del :Integer; Znak : Char);
  var
    Pocet   : Integer;
    I       : Integer;
  begin
    Pocet := 0;
    For I := Del downto 1 do
      if Veta [I] = Znak then
        Pocet := Pocet + 1;
    Writeln ('Znak ', Znak, ' se vyskytuje ',
              Pocet, ' krát ve vete')
  end;

Begin
  Write ('Napis vetu:');
  Readln (Veta);
  Writeln;
  Write ('Vyskyt ktereho znaku je treba zjistit:');
  Readln (HlZnak);
  DelkaVety := Length (Veta);
  Hledej ( DelkaVety, HlZnak)
End.
```

Pomoci programu PocetZn lze získat např. tyto výsledky:

Napis vetu: Ema ma milou mamu

Vyskyt ktereho znaku je treba zjistit:m  
Znak m se vyskytuje 5 krát ve vete

Obr. 4.10 Výstup programu PocetZn

Procedura Hledej používá dva parametry, Del typu Integer a Znak Typu Char a dvě lokální proměnné Pocet a I.

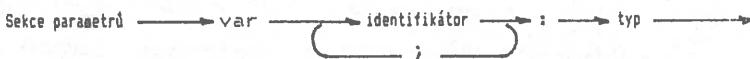
Del a Znak jsou formální parametry procedury Hledej, zatímco proměnné DelkaVety a HlZnak jsou skutečné parametry pro vykonání procedury Hledej.

- Poznámky:
- při komplikaci je ověřováno, zda počet parametrů je stejný a zda parametry jsou kompatibilní v typu;
  - parametry přenášené hodnotou umožňují přenos jen v jednom směru, z "volajícího" podprogramu do podprogramu "volaného".

#### 4.3.3 Přenos odkazem

V tomto případě se při volání jako skutečný parametr musí použít identifikátor proměnné "volajícího" podprogramu. Říkáme, že skutečné a formální parametry jsou synonyma. Identifikátory proměnných použité jako skutečné parametry jsou během volání podprogramu nahrazovány identifikátory proměnných, které jsou použity pro odpovídající formální parametry.

Sekce parametru má tuto syntaxi:



Obr. 4.11 Syntaktický diagram pro sekci parametrů  
při přenosu odkazem

Příklad

```
Program Prenos;
var
    Aktualni : Integer;

Procedure Hodnotou (Formalni : Integer);
begin
    Write ('Prenos hodnotou: ');
    Formalni := Formalni + 3;
    Write (' Aktualni= ', Aktualni);
    Writeln (' Formalni= ', Formalni);
end;

Procedure Odkazem ( var Formalni : Integer);
begin
    Write ('Prenos odkazem: ');
    Formalni := Formalni + 3;
    Write (' Aktualni= ', Aktualni);
    Writeln (' Formalni= ', Formalni);
end;

Begin
    Aktualni := 0;
    Hodnotou (Aktualni);
    Writeln ('Program : Aktualni= ', Aktualni);
    Odkazem (Aktualni);
    Writeln ('Program : Aktualni= ', Aktualni);
End.
```

Tento program dává tyto výsledky:

```
Prenos hodnotou: Aktualni= 0 Formalni= 3
Program : Aktualni= 0
Prenos odkazem : Aktualni= 3 Formalni= 3
Program : Aktualni= 3
```

Obr. 4.12 Výstup programu Prenos

Tento příklad ukazuje na rozdíl při přenosu parametrů odkazem a hodnotou. V tomto příkladu procedura Hodnotou nemodifikuje hodnotu skutečného parametru, který má stále hodnotu 0. Naopak při přenosu odkazem každá změna formálního parametru způsobuje stejnou změnu odpovídajícího skutečného parametru.

Přenos parametrů odkazem umožňuje komunikaci mezi podprogramy v obou směrech.

- Poznámky: - formální parametry procedur Hodnotou a Odkazem označují rozdílné proměnné, neboť jsou deklarovány v odlišných podprogramech. Tyto proměnné nejsou přístupné z programu;
- jestliže skutečný parametr je proměnná, jejíž index je výraz, potom tento výraz je v okamžiku vyvolávání vyhodnocen;
  - výraz může být přenášen jen hodnotou.

#### 4.3.4 Další možnosti přenosu

V Pascalu jsou definovány ještě další dvě možnosti přenosu, a to podprogramem a pomocí tzv. konformního pole. Obě možnosti nejsou v Turbo Pascalu realizovány.

- Poznámky: - přenos podprogramem umožňuje zapsat skutečné parametry jako odkazy na jiné podprogramy;
- přenos pomocí tzv. konformního pole umožnuje zápis podprogramů, které budou mít při vykonávání rozdílné hranice rozměrů polí. Konformní pole je definováno pouze v norme Pascalu úrovně 1, viz lit. [6];
  - Turbo Pascal nabízí dále přenos pomocí netypových parametrů. Při zápisu formálního parametru se neuvádí typ, tzn., že skutečný parametr může být libovolného typu. Netypový parametr je nekompatibilní k jiným typům a lze jej využívat jen tam, kde typ dat nemá význam, např. při volání procedur Move, FillChar, BlockRead/Write nebo jako adresu pro tzv. absolutní proměnnou.

Příklad

```
Procedure Vymena ( Var A1p, A2p;
                   Vel : Integer );
begin
  type
    A = array [1..MaxInt] of Byte;
  var
    A1 : A Absolute A1p;
    A2 : A Absolute A2p;
    Pom : Byte;
    I : Integer;
  begin
    for I := 1 to Vel do
      begin
        Pom := A1 [I];
        A1 [I] := A2 [I];
        A2 [I] := Pom
      end
    end;
end;
```

Předpokládejme, že existuje ve "volajícím"  
podprogramu tato deklarace:

```
type
  tMat := array [1..50,1..25] of Real;
var
  AMat, BMat : tMat;
potom proceduru Vymena lze použít
pro zajištění výměny hodnot obou polí:
Vymena (AMat, BMat, SizeOf (tMat)).
```

#### 4.3.5 Obecná pravidla pro přenos parametrů

V Pascalu se při přenosu parametrů využívají tato  
pravidla:

- procedura nebo funkce nemusí mít parametry;
- typ formálních parametrů se specifikuje v hlavičce  
procedury nebo funkce;
- jestliže před formálním parametrem je uvedeno rezervované  
slovo Var, potom se jedná o způsob přenosu odkazem. Jestliže  
se slovo Var neuvede, jedná se o přenos hodnotou;
- počet skutečných parametrů při volání procedury nebo

funkce musí být stejný jako počet formálních parametrů příslušné funkce nebo procedury;

- typ skutečného parametru musí být kompatibilní s typem formálního parametru;

- skutečné a formální parametry se musí shodovat v pořadí:

příklad

```
Procedure Parametry (X, Y, : Integer, Z : Real);
begin
end;
```

Tato procedura má tři formální parametry: 1. a 2. parametr jsou typu Integer, 3. parametr je typu Real.

Zápis: Parametry ( 3, 4, 3.14 ) je správný,

zápis: Parametry (3.14, 4, 5.2) je nesprávný, neboť první hodnota musí být typu Integer.

4.4 Vztahy mezi podprogramy

Jak již bylo ukázáno, existuje v Pascalu několik možností pro výměnu informací mezi programem a podprogramem, nebo mezi podprogramy vzájemně.

V tomto oddíle bude používán místo terminu program termín podprogram, neboť lze uvažovat, že program je zvláštní případ podprogramu.

4.4.1 Rozsah a viditelnost deklarací

Deklarace umožňuje přiřadit identifikátor nejakemu objektu.

V podprogramu lze využívat objekty, které byly deklarovány lokálně, tzn. objekty, které se objevily buď v deklarační části podprogramu nebo jalo identifikátory

formálních parametrů.

Naopak je třeba zajistit, aby objekty deklarované v podprogramu nebyly přístupné z vnitřního podprogramu. Je třeba izolovat příkazovou část podprogramu a v ní zakryt všechny detaily. Jen tak je možno zjednodušit strukturu řešené úlohy a zlepšit čitelnost programu.

Jestliže se stejný identifikátor používá v několika deklaracích v různých podprogramech, je nutné vždy přesně vědět, ke kterému objektu se v daném okamžiku identifikátor vztahuje.

Je třeba, aby existovala přesná pravidla, která určují použitečné identifikátory v každém bodu programu a tím i příslušné objekty.

#### Příklad

```
Program Rozsah;
(* Tento program nic užitečného nepočítá, jenom
 upozorňuje na problémy v rozsahu a viditelnosti
 deklaraci *)
type
  Barva = (Modra, Zelena, Zluta, Cervena);
var
  Pa, Pb : Integer;
  X, Y : Barva;

Procedure Q (Z,X : Integer);
var
  Barva : Integer;
Procedure S (Pa : Real);
begin
  Pa := 3.2;
  Pb := 10
end;
begin
  Barva := X * Z;
  Y := Zelena;
  Pa := Barva;
  S (Pa)
end;

Procedure R (Z : Integer);
begin
  X := Zluta
end;
```

```
Begin
  Pa := 0;
  Q ( Pa, 23 );
  R ( 0 );
  X := Succ ( Y )
End.
```

Rozsahem deklarace se rozumí část textu podprogramu od této deklarace až do konce tohoto podprogramu.

V předchozím příkladu rozsah deklarace typu Barva je od této deklarace až do konce programu. Stejný rozsah platí pro proměnné Pa, Pb, které jsou typu Integer a pro proměnné X, Y, které jsou typu Barva. Stejný rozsah platí také pro identifikátory procedur Q a R.

Procedura Q: rozsah formálních parametrů Z a X a proměnné Barva je platný pro celou proceduru Q.

Procedura R: rozsah formálního parametru Z je platný pro celou proceduru R.

Je třeba poznamenat, že rozsah identifikátoru nějakého podprogramu je větší než rozsah jeho lokálních objektů. Podprogram je deklarován v podprogramu, který ho obklopuje a rozsah jeho identifikátoru je platný až do konce obklopujícího podprogramu.

Jestliže dva rozsahy stejného identifikátoru mají neprázdný průnik, potom jsou buď ekvivalentní a odpovídají jedné deklaraci, nebo jeden je vnořen do druhého. Tzn., že rozsah parametru X procedury Q je vložen do rozsahu proměnné X z programu.

Jestliže dva rozsahy stejného identifikátoru mají prázdný průnik, potom se jedná o objekty bez nejasnosti.

Nejasnost při použití identifikátorů může nastat především v deklaracích, jejichž rozsah je malý. Např. v

těle procedury Q, X označuje formální parametr procedury Q a ne proměnnou X z programu. Naopak v proceduře R, X označuje proměnnou X z programu.

Identifikátor, který má větší rozsah, se po dobu překrytí nižším rozsahem stává "neviditelný", ale po ukončení platnosti nižšího rozsahu je opět použitelný v původním významu. Např. identifikátor Barva z programu se stává neviditelný v proceduře Q (je použit pro označení proměnné typu Integer). Nelze zapsat:

```
Procedure Q ...
var
    W : Barva;
    Barva : Integer;
    Y : Barva;
```

Deklarace proměnné W je možná, ale deklarace Y : Barva je nemožná, neboť předchozí deklarace učinila typ Barva neviditelným.

Na závěr je třeba shrnout, že v podprogramu je možno používat dvě skupiny objektů:

- lokální objekty, tj. objekty, které jsou deklarovány v podprogramu;
- globální objekty, tj. objekty, které jsou deklarovány v obklopujících podprogramech.

#### 4.4.2 Komunikace mezi podprogramy

Jak bylo uvedeno existují čtyři možnosti přenosu mezi podprogramem a "zbytkem programu":

- přenos hodnotou;
- přenos odkazem;
- přenos výsledku funkce do "volajícího" podprogramu;
- používání globálních proměnných.

Způsob komunikace mezi podprogramy lze studovat z několika hledisek, např.:

- zda jde o explicitní nebo implicitní charakter komunikace;
- jaký je směr při předávání informací;
- ve kterém okamžiku se uskutečňuje komunikace a
- zda jde o statický nebo dynamický charakter komunikace.

Poznání způsobu komunikace mezi podprogramy je nezbytné pro správné využívání logiky podprogramů a pro zpřesnění programátorské práce.

#### Implicitní a explicitní komunikace

Přenos hodnotou či odkazem a přenos výsledku funkce je v textu programu popsán při každém volání podprogramu. Jedná se tedy o explicitní komunikaci. Např.

Nasobeni (A, 3, B);

Poznamenejme, že i tento způsob přenosu není zcela explicitní. Vždyť např. není známo, zda parametry A,C představují přenos hodnotou či odkazem. Toto lze zjistit jedině kontrolou formálních parametrů v hlavičce "volaného" podprogramu Nasobeni.

Jestliže komunikace se uskutečňuje pomocí globálních proměnných, potom se jedná o implicitní komunikaci. Např. výsledek volání podprogramu Nasobeni mohl být zajištěn i tímto způsobem. Implicitní komunikace je, však méně kontrolovatelná a méně pružná, neboť se při každém vykonávání podprogramu používá stále stejná proměnná. Je třeba, aby program znal všechny detaily "volaného" podprogramu, jinak se mohou objevit nepředvídané chyby. Z

toto vyplývá, že implicitní komunikace musí být více kontrolovaná a podrobněji dokumentována v deklaracích podprogramů.

#### Směr komunikace

Informace mezi podprogramy mohou být přenášeny:

- jen dovnitř podprogramu, parametry předávané hodnotou;
- jen vně podprogramu, výsledek funkce;
- oběma směry, parametry předávané odkazem a globální proměnné.

#### Okamžik, kdy se uskutečňuje komunikace

Okamžik, kdy se uskutečňuje výměna informací mezi podprogramem a "zbytkem programu" závisí na způsobu přenosu.

Parametry předávané odkazem nebo globální proměnné umožňují komunikaci podprogramu s okolím po celou dobu vykonávání podprogramu. Neboť podprogram může vyvolávat jiný podprogram, který je deklarován vně podprogramu. Takto mohou vznikat složité situace, které jsou nesnadno kontrolovatelné.

#### Příklad

```
Program Komunikace;
  var
    U : Integer;

  Procedure B;
  begin
    Writeln ('B1: ',U);
    U := 3;
    Writeln ('B2: ',U)
  end;
```

```
Procedure A (var X : Integer);
begin
    Writeln ('A1: ',X);
    B;
    Writeln ('A2: ',X);
    X := 5;
    Writeln ('A3: ',X);
    B;
    Writeln ('A4: ',X);
end;

Begin
    U := 0;
    Writeln ('U1: ',U);
    A (U);
    Writeln ('U2: ',U)
End.
```

```
U1: 0
A1: 0
B1: 0
B2: 3
A2: 3
A3: 5
B1: 5
B2: 3
A4: 3
U2: 3
```

Obr. 4.13 Výsledky programu Komunikace

Při jediném volání procedur A, parametr X předávaný odkazem je synonymem globální proměnné U. Během vykonávání procedury A, přiřazení A := 5 modifikuje chování podprogramu B, který je deklarován vně podprogramu.

Je zřejmé, že komunikace, které způsobují během vykonávání podprogramu složité účinky, je třeba omezovat.

Naopak, při přenosu hodnotou a při přenosu výsledku funkce se jedná o komunikace limitované na okamžik volání a na okamžik návratu z podprogramu. Z toho vyplývá, že účinky při takovéto komunikaci jsou snadněji kontrolovatelné.

#### Komunikace statická a dynamická

Toto se týká jen bodů programu, s nimiž podprogram

komunikuje. Podprogram má vztah k témtu částem programu:

- k podprogramu nebo programu, ve kterém je deklarován;
- k podprogramům, ve kterých je vyvoláván.

Jestliže komunikace je explicitní, ať pomocí parametrů (hodnotou či odkazem) nebo výsledkem funkce, potom tato komunikace je určována "volajícím" podprogramem. Protože tento podprogram může být změněn při každém volání, hovoří se o dynamické komunikaci.

Naopak když "volaný" podprogram používá globální proměnnou, tak tato musí být vyhledána v části programu nebo podprogramu, kde je deklarována, a ne v části, kde podprogram je volán. Protože tato komunikace je stejná při každém zpracování podprogramu, hovoří se o statické komunikaci.

#### Příklad

```
Program StatDynKomunikace;
var
    Globalni : Integer;

Procedure A ( var Odkaz : Integer );
begin
    Writeln ('A - globalni = ', Globalni);
    Writeln ('A - odkazem = ', Odkaz)
end;

Procedure B;
var
    Globalni : Integer;
begin
    Globalni := 5;
    Writeln ('          Druhe volani A');
    A (Globalni);
end;

Begin
    Globalni := 0;
    Writeln ('          Prvni volani A');
    A (Globalni);
    B;
    Write ('Zaverecna hodnota      ');
    Writeln ('Globalni = ', Globalni)
End.
```

```
Prvni volani A
A - globalni = 0
A - odkazem = 0
Druhe volani A
A - globalni = 0
A - odkazem = 5
Zaverecna hodnota Globalni = 0
```

Obr. 4.14 Výsledky programu StatDynKomunikace

Proměnná Globalni z programu se stává neviditelnou v proceduře B díky nové deklaraci: Během volání procedury A v těle procedury B (tj. druhé volání A v programu) se jedná o dynamickou komunikaci, přičemž lze komunikovat jen s parametrem viditelným v proceduře B, který se stává synonymem formálního parametru Odkaz, který je předáván odkazem. Naopak proměnná Globalni, která je používána v proceduře A, je stále ta, která byla viditelná v okamžiku deklarace procedury A.

To vysvětluje, že při druhém zpracování A se vytisknou dvě proměnné Globalni (jedná se o homonyma Globalni z programu a Globalni z procedury B).

#### Pojem vedlejší efekt

Jako vedlejší efekt (angl. side-effect) se označují modifikace okolí programu, které vznikají vyhodnocením výrazu nebo vykonáním příkazu. Tyto modifikace obvykle mění hodnotu proměnných.

Pojem vedlejší efekt je významný především pro podprogramy, při jejichž volání se zajišťuje složitá operace, která může způsobit mnohé modifikace globálních proměnných nebo parametrů předávaných odkazem.

Příklad

```
Program VedlejsiEfekt;
var
    Vysledek, Hodnota : Integer;

Function Rozdil (X : Integer) : Integer;
begin
    Hodnota := Hodnota - X;
    Rozdil := 2 * X;
end;

Begin
    Hodnota := 2;
    Vysledek := Rozdil (Hodnota);
    Writeln (Vysledek, Hodnota);
    Hodnota := 2;
    Vysledek := Rozdil (2) * Rozdil (Hodnota);
    Writeln (Vysledek, Hodnota);
    Hodnota := 2;
    Vysledek := Rozdil (Hodnota) * Rozdil (2);
    Writeln (Vysledek, Hodnota)
End.
```

Výsledky jsou tyto:	Vysledek	Hodnota
4	0	
0	0	
16	-2	

Při běžném ověření přesnost výsledků není zřejmá.

Je doporučováno omezovat vedlejší efekty, které jsou způsobovány podprogramy.

Na závěr tohoto oddílu je uveden příklad s vedlejším efektem. Výsledky programu závisejí na technickém i programovém vybavení určitého počítače:

Příklad

```
Program Loterie;
Var
    Citac : Integer;

Procedure Osud (X,Y : Integer);
begin
    if X < Y then
        Writeln (' Budete mit stastne manzelstvi')
    else
        Writeln (' Budete mit nestastne manzelstvi')
end;

Function Des : Integer;
begin
    Citac := Citac + 1;
    Des   := Citac
end;

Begin
    Citac := 0;
    Osud (Des, Des)
End.
```

Shrnutí

Způsob komunikace	implicitní nebo explicitní	okamžik komunikace	sáér komunikace	statická nebo dynamická	mohný vedlejší efekt
hodnotou	explicitní	zač. zprac.	vstupní	dynamická	ne
odkazem	explicitní	během zprac.	vst/výst	dynamická	ano
výsl. funkce	explicitní	konec zprac.	výstupní	dynamická	ne
gl. proměnná	implicitní	během zprac.	vst/výst	statická	ano

Poznámka: - Zdá se, že v uvedené tabulce se jedná o stejné způsoby komunikace, které jsou označeny jako implicitní a statický. To platí pro jazyk Pascal, ale ne pro jiné jazyky.

## 4.5 Rekurzivní volání

### 4.5.1 Jednoduchá rekurze

Pokud je podprogram viditelný, neexistuje žádné omezení pro jeho volání. Tzn., že podprogram může být vyvoláván i z vlastní příkazové části.

Na této situaci není nic výjimečného. Běžně se s ní setkáváme v matematice při řešení funkcí. Např. funkci pro výpočet faktoriálu lze zapsat takto:

Jestliže  $N = 0$  tedy  $N! = 1$

jinak  $N! = N * (N - 1)!$

Takováto definice se nazývá rekurzivní (rekurentní) a přepisuje se do Pascalu relativně snadno pomocí deklarace rekurzivní funkce:

```
Function Faktorial (N : Integer) : Integer;
begin
  if N = 0 then
    faktorial := 1
  else
    faktorial := N * Faktorial (N - 1)
end;
```

Je třeba si všimnout, že identifikátor Faktorial se v druhém přiřazení používá dvěma rozdílnými způsoby. Vlevo se jedná o jméno funkce, ve které je příkaz zapsán. Toto jméno se bude využívat pro předání vypočteného výsledku. Vpravo se jedná o volání funkce, která je stejná jako ta, ve které je příkaz zapsán.

Poznámky: - každé rekurzivní vyvolávání zahajuje nové zpracování funkce Faktorial, zatímco předchozí zpracování nebylo dokončeno. Jestliže skutečný parametr při prvním zpracování je  $N$ , bude se

- zajišťovat současně N-1 zpracování funkce Faktorial;
- rekurzivní vyvolávání podprogramů vyžaduje větší prostor paměti, který je potřebný pro přiřazení hodnot lokálním proměnným při každém volání;
  - v Turbo Pascalu pod operačním systémem CP/M je třeba, aby při rekurzivním vyvolávání byla vypnuta direktiva A. Je nutno použít {\$ A-}.

#### 4.5.2 Vzájemná rekurze - direktiva Forward

Jestliže podprogram P vyvolává jiný podprogram Q, který sám vyvolává podprogram P, potom se hovoří o vzájemné rekurzi mezi dvěma podprogramy.

Vzhledem k tomu, co bylo uvedeno v oddíle 4.4, není možno zapsat dvě deklarace vzájemně viditelné. Abyste podprogram P mohl volat podprogram Q je třeba, aby podprogram Q byl deklarován později než podprogram P. Tento požadavek lze formulovat i opačně.

Directiva Forward má umožnit, aby kompilátor vyřešil tento rozpor. Její používání je možno vyjádřit takto:

Directiva Forward → hlavička podprogramu → Forward → ; →

Obr. 4.15 Syntaktický diagram direktivy Forward

Příklad

```
Program TrébaForward;
  Function F (X : Integer) : Integer; Forward;

  Function G (X : Integer) : Integer;
  begin
    if X = 0 then
      G := 1
    else
      G := G (X - 1) + F (X - 1)
  end;

  Function F;
  begin
    if X = 0 then
      F := 1
    else
      F := F (X - 1) + G (X - 1)
  end;

Begin
  Writeln (F(5))
End.
```

Výsledek je 32.

Funkce F, která je používána ve funkci G, je předem deklarována pomocí direktivy Forward, neboť sama vyvolává proceduru G.

4.6 Předdeklarované podprogramy

V každé konkrétní implementaci Pascalu je předdeklarována řada procedur a funkcí, které výrazně usnadňují programátorskou práci.

V Turbo Pascalu jsou to především podprogramy:

- pro práci s řetězci (viz oddíl 3.8.2);
- pro práci se soubory (viz oddíl 3.7).

V tomto oddíle budou uvedeny některé další předdeklarované podprogramy, které mají význam pro běžné programování.

#### 4.6.1 Předdeklarované procedury

##### ClrEol

Syntaxe: ClrEol;

Zajišťuje zrušení všech znaků od pozice kurzoru do konce řádky.

##### ClrScr

Syntaxe: ClrScr;

Vymaže obrazovku a kurzor umístí do levého horního rohu.

##### Delay

Syntaxe: Delay (Cas);

Zastaví zpracování výpočtu na tolik milisekund, kolik je uvedeno v argumentu.

##### GotoXY

Syntaxe: GotoXY (X,Y);

Nastaví kurzor do pozice X a Y, kde X určuje vodorovnou hodnotu a Y svislou hodnotu.

##### Exit

Syntaxe: Exit;

Umožňuje výstup z bloku. Když je zpracováván nějaký podprogram, je možno vyvoláním procedury Exit ukončit zpracování tohoto podprogramu. Vyvolání procedury Exit lze přirovnat příkazu GoTo, který má odkaz na návěští umístěné před posledním příkazem End v podprogramu.

Halt

Syntaxe: Halt;

Ukončuje zpracování programu a předává řízení operačnímu systému.

Randomize

Syntaxe: Randomize;

Inicializuje generátor náhodných čísel.

Move

Syntaxe: Move (Prom1, Prom2, Počet);

Přesunuje úsek paměti. Prom1 Prom2 jsou proměnné libovolného typu, Počet je výraz typu Integer. Procedura zkopíruje Počet byte od prvního byte Prom1 do Prom2. Umístění v Prom2 je také od prvního byte.

FillChar

Syntaxe: FillChar (Prom, Počet, Hodnota);

Vyplňuje část paměti zadánou Hodnotou. Prom je proměnná libovolného typu. Počet je výraz typu Integer a hodnota je výraz typu Byte nebo Char. Procedura vyplní zadáný Počet byte od prvního byte proměnné Prom.

4.6.2 Předdeklarované funkce

V Turbo Pascalu jsou k dispozici:

- aritmetické funkce: Abs, Arctan, Cos, Exp, Ln, Sin, Sqr a Sqrt;
- funkce pro jednoduché typy dat: Pred, Succ a Odd;
- funkce pro přenos: Chr, Ord, Round a Trunc.

Uvedený výčet funkcí je třeba doplnit:

Frac

Syntaxe: Frac (Číslo);

Vrací desetinnou část čísla. Výsledek je reálný.

Int

Syntaxe: Int (Číslo);

Vrací celočíselnou část čísla. Výsledek je reálný.

Keypressed

Syntaxe: Keypressed;

Vrací hodnotu typu Boolean True, když je stisknuto jakékoliv tlačítko na klávesnici.

Random

Syntaxe: Random;

Vrací náhodné číslo z intervalu <0,1>. Typ výsledku je reálný.

Random

Syntaxe: Random (Číslo);

Vrací náhodné číslo z intervalu <0,Číslo>. Číslo i výsledek funkce jsou typu Integer.

SizeOf

Syntaxe: SizeOf (Jméno);

Vrací počet byte, které v paměti zabírá proměnná nebo typ Jméno. Výsledek je typu Integer.

UpCase

Syntaxe: UpCase (Znak);

Vrací velké písmeno, které je ekvivalentní argumentu Znak. Znak musí být typu Char.

## 5. Dynamické proměnné a typ ukazatel

### 5.1 Statické a dynamické proměnné

Všechny proměnné, které jsme dosud poznali, musí být před použitím v programu předem deklarovány. Tzn., že jejich životnost je stejná jako životnost programu nebo podprogramu, ve kterém jsou deklarovány. Protože jejich životnost je určena při zápisu programu, nazývají se jako statické proměnné a jsou v programu dosažitelné pomocí svých identifikátorů.

V některých případech je třeba, aby během zpracování programu mohly být vytvářeny proměnné, jejichž počet při zápisu programu není znám nebo se tento počet během zpracování programu mění.

#### Příklad

Na vstupu je posloupnost čísel, rozdělená na jednotlivé úseky číslem 0. Na výstupu je požadována tato posloupnost, ale s opačným pořadím čísel v jednotlivých úsecích. Tedy:

vstup: 9 5 4 10 0 2 7 0 13 5 0 2 ...

výstup: 10 4 5 9 0 7 2 0 5 13 0 6 ...

Délka jednotlivých úseků v posloupnosti není předem známa. Proto nelze použít statických proměnných; proměnné určené pro seřazování čísel v úseku musí být vytvářeny dynamicky až při zpracování programu. To je hlavní cíl typu ukazatel, který umožňuje vytvářet takovéto dynamické proměnné.

Princip je tento: Proměnná typu ukazatel (jak vyplývá z jejího označení) umožňuje ukázat na dynamickou proměnnou

určitého typu. V programu se deklaruje jen proměnná typu ukazatel, zatímco dynamická proměnná je vytvářena v příkazové části podprogramu nebo programu pomocí předdeklarované procedury New. Tato procedura vyhradí v paměti (přesněji na hromadě - heap) místo odpovídající typu dynamické proměnné a adresu počátku tohoto paměťového místa uloží do proměnné typu ukazatel. Při vytváření dynamických proměnných jejichž odkazovaný typ obsahuje variantní záznam je zapotřebí místo procedury New použít proceduru GetMem. Blíže viz lit. [8].

Dynamické proměnné nelze při zápisu programu deklarovat a jejich organizaci si program obhospodařuje sám. Tyto proměnné nejsou přístupné pomocí svých identifikátorů (nemají je), ale přistupovat k nim lze pomocí proměnných typu ukazatel.

### 5.2 Deklarace

Deklarace typu ukazatel je definována takto:

Deklarace typu → identifikátor → = → ^identifikátor →  
ukazatel → typu

Obr. 5.1 Syntaktický diagram deklarace typu ukazatel

Identifikátor označuje jméno vytvářeného typu ukazatel a identifikátor typu označuje typ odkazované dynamické proměnné (někdy nazývaný jako doménový typ).

#### Příklad

```
type
  ukInt = ^Integer;
  ukBool = ^Boolean;
```

ukInt a ukBool označují dva typy ukazatelů, typy

dynamických proměnných jsou Integer a Boolean.

Příklad

```
type
    JmenoStr = String [20];
    ukClovek = ^Clovek;
    Clovek = record
        Jmeno : JmenoStr;
        Vyska : Integer;
        DalsiClovek:ukClovek;
    end; var
    PrvniClovek : ukClovek;
    JedenClovek : Clovek;
```

ukClovek je identifikátor typu ukazatel, který ukazuje na dynamické proměnné typu Clovek. Žaznam Clovek obsahuje položky Jmeno, Vyska, DalsiClovek. Položka DalsiClovek je typu ukClovek, tedy je typu ukazatel. PrvniClovek je proměnná typu ukazatel.

Je třeba upozornit, že struktura deklarace proměnné ukazatel přináší dva druhy informací:

- jméno proměnné typu ukazatel;
- jméno odkazovaného typu pro tuto proměnnou.

Při vytváření definic typů dat nemůžeme používat identifikátory, které nebyly ještě definovány. Výjimka je pouze u typu ukazatel. Např. v deklaraci typu ukazatel ukClovek se používá ještě nedefinovaný identifikátor typu Clovek. Tato výjimka umožňuje vytváření složitějších datových struktur.

5.3 Přístup k dynamickým proměnným

Pro zajištění přístupu k dynamické proměnné, případně k jedné z jejích položek, se používá označení  $\sim$  (šipka). Jestliže P je proměnná typu ukazatel ukazující na nějakou dynamickou proměnnou, píšeme  $P^\sim$  pro celkové označení dynamické proměnné.

Příklad

```
type
    tab : array [1..N] of Real;
    Datum : record
        Den = 1..31;
        Mesic = 1..12;
    end;
var
    T : ^Tab;
    D : ^Datum;
    I : ^Integer;
```

Proměnné T, D a I označují proměnné typu ukazatel. Po dynamickém umístění všech proměnných (tj. po vykonání sekvence příkazů New(T), New(D) a New(I) ) jsou k disposici vždy dva druhy informací:

- proměnná ukazatel T a dynamická proměnná T^ typu Tab;
- proměnná ukazatel D a dynamická proměnná D^ typu Datum;
- proměnná ukazatel I a dynamická proměnná I^ typu Integer.

Přístup k dynamickým proměnným se zajišťuje např. takto:

```
I^ := 3;
T^[4] := 3.14 * I;
D^.Mesic := 11;
```

Typ dynamické proměnné je určen pomocí odkazovaného typu v deklaraci. Pro dynamickou proměnnou jsou přípustné všechny typy operací, které jsou možné pro proměnné stejného typu jako je typ odkazované proměnné.

5.4 Operace s dynamickými proměnnými a proměnnými ukazateli

Pro objekty typu ukazatel jsou definovány operace pro tvorbu, přiřazení, porovnávání a pro destrukci.

Pro dynamické proměnné: - všechny přípustné operace

vzhledem k odkazovanému typu,

- tvorba a destrukce.

Pro proměnné ukazatel: - přiřazení a

- porovnávání.

Tvorba - procedura New.

Pro tvorbu dynamické proměnné se používá standardní procedura NEW. Tato procedura se zapisuje takto:

New (P)

P je proměnná typu ukazatel, odkazující na typ T.

Tato procedura vytváří nezbytný prostor v paměti pro objekt typu T, přičemž se do P přiřazuje hodnota adresy paměťového místa počátku vyhrazeného prostoru pro dynamickou proměnnou P^.

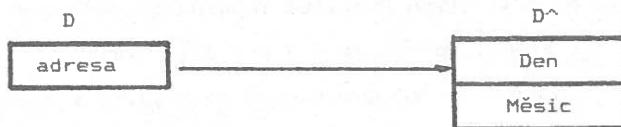
Příklad

```
type
    Pent = ^Integer;
var
    Ip : Pent;
```

Po vykonání příkazu New(Ip) se vyhradí v paměti prostor odpovídající proměnné typu Integer. Adresa tohoto prostoru je uložena do proměnné Ip.



Dle předchozí deklarace, vykonání příkazu New(D) zajistí vytvoření potřebného prostoru pro objekt typu Datum. Adresa tohoto prostoru je umístěna do proměnné D.



Přiřazení proměnným typu ukazatel

Proměnné typu ukazatel lze přiřadit hodnotu jiné proměnné typu ukazatel, nebo předdefinovanou konstantu NIL

(přesněji, jedná se o rezervované slovo, viz oddíl 2.1.2).

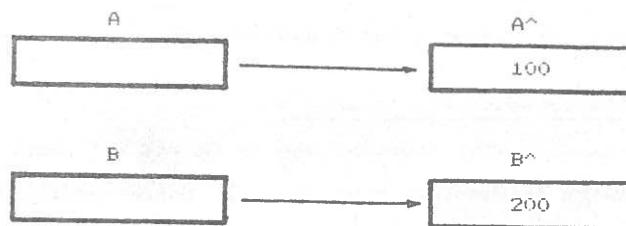
Přiřazení hodnoty NIL proměnné typu ukazatel znamená, že tato proměnná neukazuje na žádnou dynamickou proměnnou. Tato konstanta se využívá pro ukončení zřetězené struktury a na obrázcích bývá často označována jako uzemnění. Důležité je, že konstantu NIL lze přiřadit proměnné typu ukazatel libovolného doménového typu.

#### Příklad

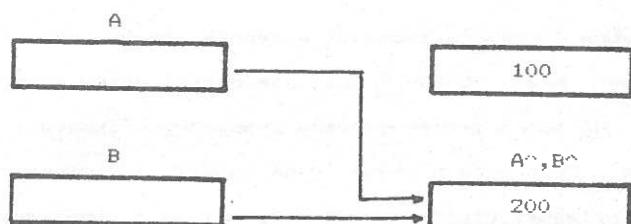
Je důležité rozlišit dva typy přiřazení :

Předpokládejme situaci v operační paměti při deklaraci:

```
var  
  A,B : ^Integer;
```

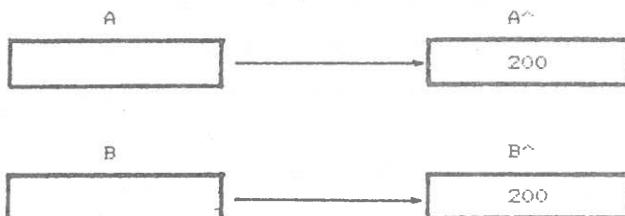


Po provedení příkazu  $A := B$ ; bude situace v paměti vypadat takto :



(Přiřazení mezi ukazateli).

Po provedení příkazu  $A^{\wedge} := B^{\wedge}$ ; takto :



(Přiřazení mezi dynamickými proměnnými).

#### Porovnávání proměnných typu ukazatel

Pro proměnné typu ukazatel lze použít jen operátory " $=$ " a " $\neq$ ". Pro vypsání hodnoty adresy uložené v proměnné typu ukazatel můžeme využít standardní funkci Ord. Výpis hodnot adres obsažených v ukazatelích je jediná možnost při ladění programu, jak získat alespoň částečný obraz stavu hromady.

#### Destrukce dynamických proměnných

Existují tři způsoby, jak se zbavit již nepoužívaných dynamických proměnných. Kdybychom tak totiž nečinili, brzo by se nám podařilo zcela zaplnit volnou paměť. První ošetření, velmi často používané, je programové. Jedná se o odkládání nepotřebných dynamických proměnných do "odpadkových košů". Potom před provedením procedury New se nejdříve podíváme do tohoto koše, zda tam není nějaká nepotřebná dynamická proměnná a teprve je-li koš prázdný, použijeme New. TURBO Pascal nám nabízí ovšem elegantnější řešení. Má totiž implementovány procedury Dispose, Mark a Release. Procedura Dispose je na rozdíl od dvojice procedur Mark a Release, které jsou obvyklé v jiných implementacích, ve standardu Pascalu.

Syntaxe je :              Dispose (uk1) ;  
                            Mark (uk2) ;  
                            Release (uk3) ;

Kde uk1, uk2 a uk3 jsou proměnná typu ukazatel.  
Hodnota uk3 musí být předem nastavena pomocí  
procedury Mark.

Standardní procedura Dispose nám uvolní paměť  
odpovídající dynamické proměnné, na kterou odkazuje ukazatel  
zadaný jako její parametr. Při používání Dispose tedy  
vznikají v paměti různě velké nevyužité díry (přesněji v  
části paměti označované jako Heap, což značí hromada). Při  
následujícím volání procedury New jsou nejdříve prohledány  
tyto díry odspoda, teprve není-li nalezena dostatečně velká  
pro vytvoření příslušné dynamické proměnné, je zabrán nový  
prostor na vrcholu hromady.

Místo procedury Dispose můžeme použít dvojici  
Mark/Release. Procedura Mark zkopíruje do svého parametru  
hodnotu systemového ukazatele na vrchol hromady (heap  
pointer). Po zavolání procedury Release na takto označenou  
proměnnou, dojde k uvolnění paměti zabrané hromadou nad  
adresou uloženou v proměnné včetně. To znamená, že jsou  
uvolněny všechny dynamické proměnné vytvořené pomocí New po  
označení našeho ukazatele procedurou Mark.

Je vhodné nepoužívat současně uvolňování dynamických  
proměnných pomocí procedur Dispose a Mark/Release. Pokud  
totiž velmi pozorně nezvážíme, jak bude asi vypadat hromada  
při zpracování našeho programu, začne se tento chovat  
nepředvídatelně. Vzhledem k tomu, že ošetřování hromady je  
závislé na použitém operačním systému, doporučujeme další  
studium (viz např. lit. [8]).

### 5.5 Príklad na používání dynamických proměnných

```
program PointerDemo;
type
  JmenoStr = String [20];
  ukClovek = ^Clovek;
  Clovek = record
    Jmeno : JmenoStr;
    Vyska : Integer;
    DalsiClovek:ukClovek;
  end;
var
  PrvniClovek:ukClovek;

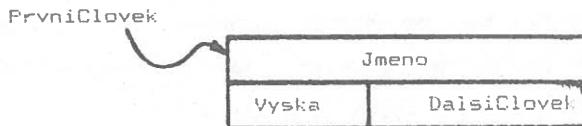
procedure NactiLidi ( var Prvni:ukClovek );
{ Uløí osobní karty do zasobníku zpristupnенého
  ukazatelem "Prvni".
  Cte ze vstupu dokud nenaøazi na jmeno "posledni" }
var
  jm : String(20);
  v : Integer;
  pom : ukClovek;
begin
  Writeln('Jmeno');
  Readln(jm);
  while jm<>'posledni' do
  begin
    Writeln('Vyska v centimetrech');
    Readln(v); Writeln;
    pom:=Prvni;
    New(Prvni);
    Prvni^.Jmeno:=jm;
    Prvni^.Vyska:=v;
    Prvni^.DalsiClovek:=pom;
    Writeln('Jmeno (napis "posledni" pro ukonèení vstupu)');
    Readln(jm);
  end;
end; { NactiLidi }

procedure VypisZasobnik (uk:ukClovek);
begin
  { vypise zasobnik zpristupneny ukazatelem uk }
  while uk<>nil do
  begin
    with uk^ do
      begin Writeln(Jmeno);
      Writeln(Vyska);
    end;
    Writeln;
    uk:=uk^.DalsiClovek;
  end;
end; { VypisZasobnik }
```

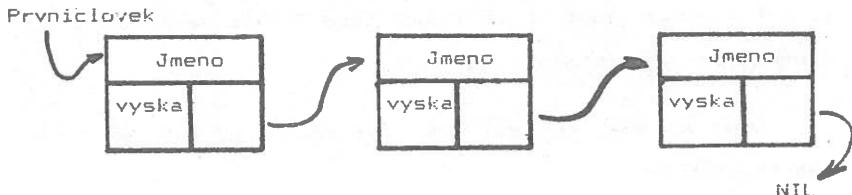
```
procedure NajdiNejvyssiho (uk:ukClovek);
{ najde nejvyssiho cloveka v zasobniku
  a vypise jeho jmeno a vysku }
var
  Max:Integer;
  ukmax:ukClovek;
begin
  Max:=0;
  while uk<>nil do
    begin
      if uk^.Vyska>Max then begin ukmax:=uk;
                                Max:=uk^.Vyska;
                                end;
      uk:=uk^.DalsiClovek;
    end;
  Writeln;
  Writeln('Nejvyssi clovek na seznamu se jmenuje
           ukmax^.jmeno');
  Writeln('a meri ',max,' centimetru.');
end; { NajdiNejvyssiho }

Begin
  PrvniClovek:=nil; { initialisace seznamu }
  NactiLidi(PrvniClovek);
  VypisZasobnik(PrvniClovek);
  NajdiNejvyssiho(PrvniClovek);
End. { hlavni program }
```

Pro představu o rozmištění dynamických proměnných při zpracovávání tohoto programu je možno vytvořit tato schémata:



Obr. 5.2 Stav po vykonání procedury New



Obr. 5.3 Zřetězení dynamických proměnných

Je vhodné, aby si čtenář uvědomil, že zápis identifikátoru poslední dynamické proměnné v seznamu na předešlém obrázku je

`PrvniClovek^.DalsiClovek^.DalsiClovek^` (jedná se o tzv. dereferenci). Tato proměnná je typu `Clovek` a jako s takovou s ní i můžeme zacházet, pokud není vytvořena, dojde k chybě při zpracování programu nebo k nedefinovatelnému chování programu. Výška tohoto člověka je

`PrvniClovek^.DalsiClovek^.DalsiClovek^.Vyska`, jedná se o proměnnou typu `Integer`.

#### 5.6 Význam dynamických proměnných a typů ukazatel

Dynamické proměnné a ukazatelé nám dávají možnost vytvářet abstraktní struktury dat (viz např. lit. [11]) a další zajímavé možnosti. Domníváme se, že se jedná o nejsilnější a také o nejjazdimavější stránku jazyka Pascal. Bylo by vhodné, aby se čtenář studiu této problematiky věnoval až po dobrém osvojení jazyka Pascal. Zvládnutí dynamických proměnných vyžaduje praxi a i po delší době se nám budou do jisté míry jevit jako "černá skříňka".

Ukazatelé řeší částečně i problematiku rekurzivních definic dat, tj. struktura typu `T` může obsahovat jako položku odkaz opět na strukturu typu `T` (viz např. proměnná `JedenClovek` v příkladu v oddíle 5.2).

Doporučujeme zkušenějšímu čtenáři zamyslet se nad těmito problémy:

Předávání hodnot mezi procedurami se v jazyce Pascal uskutečňuje pomocí parametrů, globálních proměnných nebo

dynamických proměnných.

Dynamické proměnné alokované (tj. vytvořené na hromadě) v podprogramu (proceduře) porušují pravidla o viditelnosti a o fyzické existenci lokálních proměnných. Nedochází totiž k jejich zániku po ztrátě aktivity podprogramu, ve kterém došlo k jejich alokaci.

Díky ukazatelům můžeme pracovat i s tak velkými objekty (týká se to zejména matic), které se nám nevejdou do operační paměti obhospodařované Turbo Pascalem. K vytvoření těchto objektů v paměti dojde až při běhu programu a ne již při jeho překladu. Tato situace nastává typicky u počítačů typu IBM PC s 256 či 640 KB operační paměti a při použití verze překladače Turbo Pascal nižší než 4.0.

## 6. Několik rad k programování

Jedním z hlavních cílů, který si vytýčili autoři Pascalu, je plná podpora správného strukturování a účelné organizace programů. Tento cíl je částečně dosažen navrženou syntaxí deklarací a příkazů jazyka.

Samotná syntaxe jazyka, ač je dobře propracovaná, není dostatečná. Pro správné programování je třeba zvládnout různé sémantické mechanismy, které jazyk dává k disposici.

V tomto oddíle bude ukázáno, jak používat jisté sémantické charakteristiky Pascalu, které umožňují lepší organizaci programů.

Nejde o výklad programovacích technik, který by přesáhl rozsah tohoto učebního textu, ale pouze o uvedení několika užitečných zásad, které pomáhají při praktickém programování.

Tyto zásady se dají shrnout do těchto vět:

- pojmenujte vše, co může být pojmenováno;
- systematicky inicializujte konstanty;
- rozložte program na nezávislé moduly tak, jak je to jen možné;
- definujte přesně vztahy mezi moduly.

### 6.1 Používání identifikátorů

V běžné programátorské praxi se uznává, že k programu musí být vždy vyhotovena dokumentace, která se využívá především pro údržbu programů. Nevýhodou tohoto způsobu je, že musí existovat dva texty místo jednoho (tj. program a jeho dokumentace) a že je třeba neustále udržovat jejich

logičnost (soulad).

Jedno z řešení tohoto problému spočívá ve vytváření programů, které jsou čitelné bez komentářů. Nejde však o úplné potlačení komentářů, které jsou nutné pro vysvětlení složitých myšlenek nebo obtížných algoritmů, ale o vyloučení komentářů, které parafrázují text programu a tím jej zatečujuji.

Je třeba, aby text programu byl čitelný bez velké námahy. K dosažení tohoto cíle existuje několik možností. Zvláště dobrá indentace (odsazování či zoubkování) textu přispívá k větší čitelnosti programu (viz příklady v tomto učebním textu).

Jiným důležitým prostředkem pro zlepšení čitelnosti je systematické používání vhodných mnemotechnických identifikátorů.

Proto je doporučováno používat i delší identifikátory.  
Např. příkaz:

```
For IndexSt := 1 to MaxPocetSt do
    VypoctiPrumer (Znamky [IndexSt]);
```

nepotřebuje žádný komentář.

Poznamenejme, že v tomto příkladu jazyk Pascal umožnil pojmenování dvou druhů objektů:

- konstanty: MaxPocetSt;
- proměnných: IndexSt, Znamky a
- příkazu procedury: VypoctiPrumer.

Je zřejmé, že podobně lze pojmenovat i používané typy dat.

### 6.1.1 Označování konstant

Náhrada explicitních konstant programu pomocí symbolických konstant přináší výhody jak z hlediska čitelnosti, tak z hlediska úč.

Předpokládejme, že je třeba vypracovat program pro vyhodnocení výsledků studia a využití času studentů v jednotlivých studijních skupinách.

Jestliže maximální počet studentů ve skupině bude 24, potom příkaz:

```
For IndexSt := 1 to 24 ...  
je meně obecný než příkaz:
```

```
For IndexSt := 1 to MaxPocetSt. ... , který následuje  
po deklaraci:
```

```
Const  
  MaxPocetSt = 24  
    (* max. pocet studentu ve st. skupine*)
```

Je dosti pravděpodobné, že tato konstanta bude v programu využívána několikrát. Např. takto:

```
type  
  KlasStup = 1..4;  
Var  
  Znamky : array [1..MaxPocetSt] of KlasStup;  
  Stari : array [1..MaxPocetSt] of Integer;
```

Je jisté, že konstanta MaxPocetSt se bude v příkazech programu vyskytovat také několikrát. Např. v cyklu For v podmíněném příkazu apod.

Každý výskyt konstanty bude srozumitelnější, když použijeme symbolické označení.

V případě modifikace programu pro jinou maximální velikost studijní skupiny stačí změnit v deklaraci konstantu MaxPocetSt. V ostatním textu programu není třeba dělat další úpravy.

Jestliže konstanta zůstává anonymní, tj., bez jména, potom se musí program opravit ve všech bodech, kde se anonymní konstanta používá. Počet oprav může být značný a přitom mohou vznikat další chyby, např. tím, že se zapomene opravit některý výskyt dané konstanty.

Tento problém může být jěště větší, když se v programu objevuje několik rozdílných anonymních konstant, které mají stejnou hodnotu.

V uvedeném zadání úlohy bylo řečeno, že je třeba vyhodnotit i využití času studentů. Takže např. se budou používat tyto deklarace:

```
type
    Aktivita = (prednasky, cviceni, jidlo,
                  studium, odpocinek);
var
    Rozvrh : array [1..24] of Aktivita;
```

a potom se v programu mohou objevit příkazy:

```
For Hodina := 1 to 24 do
```

Jestliže se použijí anonymní konstanty, pak je možné, že se zamění význam konstanty 24 jako maximálního počtu studentů ve skupině s významem poslední hodiny dne.

Tento problém zmizí, když se budou používat identifikátory konstant dle této deklarace:

```
MaxPocetSt = 24;
ZacDne = 1;
KonecDne = 24;
```

Poznamenejme, že používání deklarovaných konstant je poměrně časté v publikovaných programech, neboť pomocí nich jsou zaváděny do programu hodnoty, které je možno změnit v závislosti na typu počítače, který je k disposici.

### 6.1.2 Označování typů

Typy se v Pascalu používají pro popis povahy a vlastností proměnných a hodnot, které se objevují v programu.

Typ může být pojmenován v úseku deklarace typů a potom se využívá pomocí svého jména.

#### Příklad

```
type
  tJmeno = array [1..MaxZn] of Char;
  tVeta = record
    Jmeno : tJmeno;
    kusy : Integer
  end;
  tStavSkladu = array [1..MaxStav] of tVeta;
```

V tomto příkladu se definují tři typy, které se budou využívat v programu.

Typ lze používat také anonymně, tzn., že jeho popis je používán přímo při deklaraci proměnných.

Místo deklarace:

```
var
  Sklad : tStavSkladu;
```

je možno zapsat:

```
Sklad : array [1..MaxStav] of record
  Jmeno : array [1..MaxZn] of Char;
  kusy : Integer
end;
```

Tento zápis je méně vhodný a používání anonymních typů není praktické.

Např. po deklaraci:

```
var
  A, B : record
    položka : Integer
  end;


příkaz A := B; je správný, zatímco po deklaraci:


var
  A : record
    polozka : Integer
  end;
  B : record
    polozka : Integer
  end;


příkaz A := B; způsobí chybu.


```

Při praktickém programování se doporučuje přijat určitou konvenci pro popis typů, aby se předcházelo omylům a chybám.

Např. po deklaraci:

```
type
  Barvy = (Modra, Zelena, Červena);
var
  Barva : Barvy;


je po určitém čase dosti obtížné si vzpomenout, jestli Barva znamená typ či proměnnou. Jednou z možností při řešení tohoto problému je zavedení konvence, kdy každému typu předchází malé písmeno t.


```

Zápis:

```
type
  tBarva := (Modra, Zelena, Červena);
var
  Barva : tBarva;
```

je přehlednější a jasnější.

#### 6.1.3 Označování proměnných

Úplné proměnné, tj. proměnné, které jsou popsány v deklaraci proměnných, jsou touto deklarací plně popsány.

Pro proměnné se doporučuje vybírat taková jména, která jasně ukazují na význam proměnné a zvyšují čitelnost programu.

Při deklaraci proměnných dodržujme zásadu:

Stejná proměnná nesmí mít v jednom programu dva různé a nezávislé způsoby využívání, i když tyto způsoby využívání nebudou současné.

Je proto výhodnější deklarovat dvě proměnné, než deklarovat jednu proměnnou pro dva způsoby využití.

Je samozřejmé, že každé jméno vysvětluje význam proměnné a tím se snižuje riziko možných chyb.

Lze namítnout, že toto pravidlo nešetří místem v paměti. Ale pozor, např. kompilátor Turbo Pascalu má účinné prostředky pro usporu paměťového místa (lokální proměnné dvou nezávislých podprogramů sdílejí stejný prostor paměti).

Porovnejte dva zápisy při řešení kvadratické rovnice:

```
- Koren := ( -B + Sqrt (B * B - 4 * A * C)) / 2 * A;  
- Delta := B * B - 4 * A * C;  
Koren := ( -B + Sqrt (Delta)) / 2 * A;
```

Je zřejmé, že druhý zápis je čitelnější.

#### 6.1.4 Označování podprogramů

V oddíle 4.1 bylo ukázáno, že je vhodné rozdělovat řešenou úlohu na dílci části, podprogramy.

Je samozřejmé, že vhodný výběr jmen podprogramů,

zajišťuje větší čitelnost té části programu, kde se tyto podprogramy vyvolávají.

Příklad

```
Begin
    TvorbaMenu;
    Zpracovani;
    KonecProgramu;
End;
```

I když tento program používá jen tři příkazy, je rozdělení úlohy na tři dílčí části evidentní.

6.2 Používání příkazu skoku

Zkušenosti programátorů a některé teoretické práce dokázaly, že anarchické používání příkazu skoku snižuje čitelnost programu a komplikuje vytvářený program. To platí pro jakýkoliv programovací jazyk.

Používání příkazu skoku v Pascalu je výjimečné a je jen málo situací, kde je vhodné.

V Pascalu příkaz skoku je nahrazován speciálními příkazy, které umožňují logičtější konstrukci programu:

- jsou to příkazy cyklu: While, Repeat, For a
- možnosti práce s podprogramy, příkaz procedury.

Oprávněné použití příkazu skoku je především v těchto situacích:

- cykly s několika násobnými východy a
- zpracování nenormálních situací v programech.

U všech cyklů v Pascalu je možný jen jeden východ. Např. v situaci dále uvedené je použití příkazu skoku oprávněné:

```
While Podminka do
begin
    Prikaz1;
    if Podminka2 then
        begin
            Prikaz2;
            GoTo Konec
        end;
    Prikaz3
end;
Konec: Prikaz4;
```

Při zpracování podprogramů se často stává, že je třeba přerušit jejich zpracování, jestliže se objeví chyba. Použití příkazu skoku je možné, ale v Turbo Pascalu je lze nahradit vyvoláním příkazu Exit.

```
Procedure Chyba (Hodnota : Real;
                  var Ok : Boolean);
begin
    Prikaz1;
    Ok := true;
    if hodnota < MaxHod then
        Prikaz2
    else
        begin
            Ok := false;
            Exit
        end;
    Prikaz3
end;
```

### 6.3 Organizace programů

Již několikrát bylo zdůrazněno, že při návrhu programu je vhodné řešenou úlohu rozdělit do dílčích částí – podprogramů.

Dekompozice úlohy se zajišťuje postupně na několika úrovních, až do dosažení požadované elementární úrovně, kdy dílčí části již jsou snadno programovatelné.

Jestliže při dekompozici úlohy se postupuje shora dolů, tak při realizaci programu se postupuje zdola nahoru. Nejdříve se realizují dílčí elementární programy, které po

odladění a ověření jsou postupně slučovány do větších celků. Takto se postupuje až do konečného vytvoření celého programu.

#### Lokální deklarace

Jestliže jeden podprogram je využíván jen v podprogramu A, je třeba, aby tento podprogram byl deklarován lokálně v rámci A.

Každý čtenář programu bude vědět, že z důvodu existence pravidla viditelnosti je tento podprogram použitelný jen v podprogramu A. To platí pro deklarace všech objektů, které jsou využívány v podprogramu A a jeho lokálních podprogramech.

#### Modularizace

Stává se často, že část zpracování není zajištována jen jedním podprogramem, ale skupinou podprogramů, které využívají společnou deklaraci.

V těchto případech se skupiny podprogramů označují jmény. Pro takto označené skupiny podprogramů se používá název modul. Doporučuje se, aby tyto skupiny (moduly) byly ukládány do knihovny programů.

Poznámka: - v Turbo Pascalu je možno tyto moduly považovat za vkládané soubory a dle potřeby je vkládat do programu pomocí direktivy I {\$I Jmeno.xxx}.  
Přesnější popis viz oddíl 7.3.1.

## 7. Turbo Pascal

V této kapitole bude uvedena jak obecná charakteristika Turbo Pascalu, tak stručný popis tří základních složek této úspěšné implementace jazyka Pascal: editoru, kompilátoru a prostředků ladění. Poznamenejme, že tyto složky vytváří integrovaný a interaktivní celek.

Při hodnocení funkcí Turbo Pascalu vycházíme jak z vlastních zkušeností, tak z lit. [1].

### 7.1 Charakteristika Turbo Pascalu

Turbo Pascal dosáhl mimořádného úspěchu především tím, že stejně jako zkušenému programátorovi, tak i začátečníkovi nabízí kvalitní prostředek pro vývoj aplikací. Turbo Pascal není jen programovací jazyk. Je to integrovaný celek, který má tři výkonné složky: editor, kompilátor a prostředky pro ladění (tzv. debugger).

Turbo Pascal může být využíván jak na 8-bitových, tak na 16-bitových mikropočítáčích, a to pod operačními systémy CP/M, CP/M-86 a MS-DOS. Při použití na 8-bitovém mikropočítáci je nutné, aby tento mikropočítáč byl zkonstruován na bázi mikroprocesoru Z-80. Turbo Pascal tedy nelze využívat na 8-bitových mikropočítáčích s mikroprocesorem Intel 8080.

Turbo Pascal může pracovat na mikropočítáčích různé konfigurace, a to díky instalaci programu Tinst, který umožňuje definovat např. typ klávesnice či rozměry obrazovky. Proto je možné využítí Turbo Pascalu na mikropočítáčích, které jsou dosti rozšířené v našem

zemědělství (TNS, Robotron 1715 aj.).

Jazyk Turbo Pascal vychází z mezinárodní normy, od které se liší jen málo. Hlavní odchylky jsou tyto:

- nejsou realizovány procedury vstupu/výstupu Get a Put. Funkce těchto procedur jsou obsaženy v předdefinovaných procedurách Read a Write;

- není realizována procedura Page, a to především proto, že v operačním systému CP/M ji nelze definovat;

- rezervované slovo Packed nemá žádný účinek. Zhušťování se zajišťuje automaticky všude tam, kde je možné. Z tohoto důvodu nejsou realizovány funkce Pack a Unpack;

- jako parametry nelze použít procedury a funkce (viz oddíl 4.2).

Turbo Pascal přináší řadu rozšíření vzhledem k mezinárodní normě. Většina z nich již byla uvedena a proto je možno ta nejdůležitější rozšíření jen zrekapitulovat:

- soubory s přímým přístupem (tzv. typové soubory);
- práce s řetězci znaků;
- typové konstanty;
- předdefinovaný typ Byte;
- další předdefinované funkce a podprogramy.

Turbo Pascal ve verzi 3.0, na mikropočítacích typu IBM/PC, umožňuje práci s barevnou obrazovkou, vytváření okének, grafické výstupy a také ovládání zvukových signálů.

Mezi výhody Turbo Pascalu je možno zařadit i skutečnost, že programové nadstavby jsou dodávány ve zdrojovém kódu. Pro uživatele i programátora jsou k disposici:

- Turbo Database Toolbox - programové moduly pro práci s

indexsekvenčními soubory a pro třídění;

- Turbo Graphics Toolbox - úplná knihovna procedur a funkcí pro práci s grafickou obrazovkou. Tyto podprogramy umožňují kreslení různých křivek, histogramu, geometrických útvarů ato.;

- Turbo Editor Toolbox - programové moduly pro práci s textem.

Turbo Pascal je možno považovat za otevřený systém. Neustále je rozšiřován o nové programové nadstavby. Průběžně dochází i k jeho inovaci. Ve verzi 4.0, která se objevila v listopadu 1987, je možno využívat separátní komplikaci jednotlivých modulů a další užitečné vlastnosti, které přináší vývoj v oblasti programového vybavení.

V příručce (lit. [8]) jsou popsány též specifické možnosti práce pod různými operačními systémy. Uvedené pokyny umožňují zkušenému programátorovi více využívat technického vybavení počítače, např. zařazování podprogramů ve strojovém kódu.

Výklad v dalších oddílech této kapitoly bude zaměřen jen na práci pod operačním systémem CP/M.

### 7.2 Editor

Jednou z podstatných složek Turbo Pascalu je celobrazovkový editor, který je určen pro vytváření zdrojového textu programu.

Práce v tomto editoru je velmi praktická a logika příkazů je shodná s tou, která byla zavedena u textového editoru WordStar. (Textový editor WordStar byl uveden na trh v roce 1980 a je považován za standard programů pro práci s

anglickým textem).

Turbo editor je v podstatě velmi jednoduchý a při praktické činnosti rychle osvojiteLNý. Zdrojový program se vytváří obdobným zpùsobem jako kdyby se používal běžný psací stroj. Ale na rozdíl od psacího stroje poskytuje editor řadu praktických funkcí, které umožňují snadnější manipulaci s textem. Lze říci, že tvorba zdrojového textu programu je mnohem snadnější než při běžném zápisu na papír.

Při práci v Turbo editoru je možno využívat celkem 45 různých příkazů. Tyto příkazy lze rozdělit do těchto čtyř skupin:

- příkazy pro posun kurzoru;
- příkazy pro vkládání a destrukci;
- příkazy pro práci s bloky a
- ostatní příkazy.

Jednotlivé příkazy se spouštějí stisknutím jednoho nebo několika tlačítka na klávesnici. Standardní kombinace stisků tlačítka jsou dobře ergonomický navrženy, ale přesto je možno pomocí instalačního programu vytvořit vlastní kombinace tlačítka, pomocí nichž budou jednotlivé příkazy řízeny.

#### Příkazy pro posun kurzoru

Tyto příkazy dělíme na základní a rozšiřující.

Základní příkazy umožňují posun kurzoru na libovolné místo na obrazovce.

Rozšiřující příkazy umožňují rychlejší posun kurzoru po obrazovce, posun kurzoru na začátek či konec obrazovky, bloku případně i souboru.

### Příkazy pro vkládání a destrukci

Tyto příkazy umožňují jak vkládat tak rušit znaky, slova případně i řádky. Pomocí těchto příkazů lze nastavit režim vkládání (insert) či přepisování (overwrite).

### Příkazy pro práci s bloky

Pod pojmem blok se při práci s editorem rozumí libovolná část textu (od jednoho znaku až do několika stran textu). Začátek a konec bloku musí být označen speciálními příkazy. Takoto označený blok může být přesunut (případně zkopirován) na jiné místo, zrušen či zapsán do souboru na disk. Existuje též příkaz, který umožňuje čtení externího souboru do textu jako jeden blok.

### Ostatní příkazy

Mezi ostatní příkazy je možno zařadit příkaz, který umožňuje vyhledání libovolného řetězce znaků. Někdy se využívá také příkaz, který umožňuje nalezení a záměnu zvoleného řetězce znaků.

## 7.3 Kompilátor

Mezi základní praktická řešení, která výrazně usnadňují práci s Turbo kompilátorem, je možno zařadit:

- vkládané soubory,
- systém segmentace a
- způsoby komplikace

### 7.3.1 Vkládané soubory

Skutečnost, že editor umožňuje editaci jen v operační paměti, limituje rozsah zpracovávaného zdrojového textu

programu. Ale text programu může být rozdělen na menší části a v jeden celek spojen až při komplikaci. Toto dělení zdrojového textu zjednodušuje programátorskou práci, neboť odladěné podprogramy mohou být ukládány do souborů na disku. Tak lze vytvářet vlastní "knihovny" podprogramů, či modulů nebo jen deklaracních částí. Soubor na disku může být tvořen libovolnou částí zdrojového textu programu - blokem (viz oddíl 7.2).

Vložení takovéto části zdrojového textu se zajišťuje pomocí direktivy I. Tato direktiva má tuto syntaxi:

```
{$I JménoSouboru}
```

kde JménoSouboru je jméno souboru na disku.

Příklad

```
{$I E1Proc.gen}  
{$I Input.Sys}  
{$I Vypis.pas}
```

Je třeba upozornit, že vkládaný soubor se nesmí odvolávat na jiný vkládaný soubor.

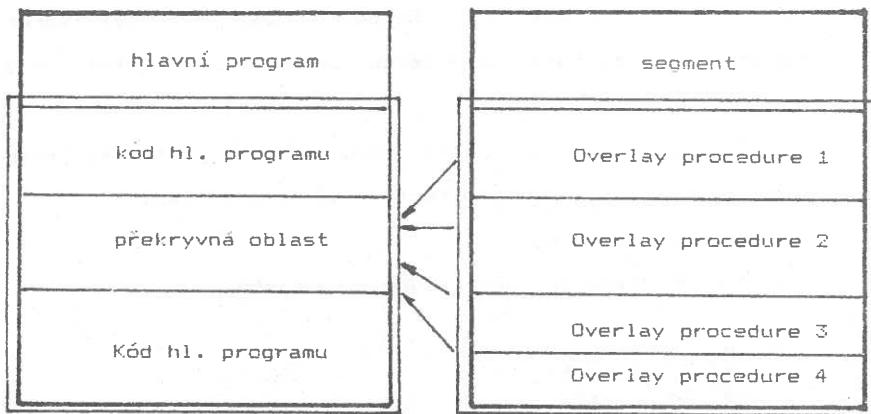
### 7.3.2 Systém segmentace

Systém segmentace (překrývání) umožňuje vytvářet programy, které budou větší než dostupný rozsah operační paměti. Tento systém se hlavně využívá při rozsáhlých aplikacích tak, že jednotlivé podprogramy jsou vykonávány ve stejném prostoru operační paměti.

Před komplikací tyto podprogramy musí mít v klavičce před jménem podprogramu uvedeno rezervované slovo Overlay (viz oddíl 2.1.2). Podprogramy se slovem Overlay jsou zkompilovány do samostatného souboru na disku (segmentu). V operační paměti se pro podprogramy vyhrazuje prostor

(překryvná oblast) dle rozsahu největšího podprogramu Overlay.

Situaci při vytvoření jednoho segmentu a jedné překryvné oblasti lze znázornit takto:



Obr. 7.1 Princip segmentace

Pro vytvoření jednoho segmentu a tím jedné překryvné oblasti je třeba, aby Overlay procedury byly uvedeny bezprostředně za sebou.

Technika segmentace vyžaduje, aby rozdělení programu do jednotlivých segmentů bylo pozorně zváženo, neboť se může stát, že časté vyvolávání Overlay procedur do překryvné oblasti bude časově příliš náročné.

### 7.3.3 Režimy komplikace

V Turbo Pascalu lze zvolit jeden ze tří možných režimů komplikace, a to:

- režim M, který je implicitní. Výsledek komplikace (program ve strojovém kódu) je umístěn v operační paměti a lze jej

spustit příkazem R:

- režim C, který je možno zvolit. Výsledek komplikace se ukládá na disk pod jménem původního souboru, ale s příponou .COM. Takto vytvořený program lze spustit přímo z operačního systému;

- režim H, který je také možno zvolit. Výsledek komplikace se ukládá na disk pod jménem původního souboru, ale s příponou .CHN. Tento program nelze spustit přímo, ale jen z jiného programu v Pascalu pomocí procedury Chain.

#### 7.4 Ladící prostředky

Mezi ladící prostředky Turbo Pascalu je možno zařadit režim práce s komplikovaným souborem, direktivy komplikátoru a způsob ošetření chyb.

##### 7.4.1 Režimy práce s komplikovaným souborem

Při komplikaci jsou k disposici dva druhy souborů:

Hlavní soubor (Main file), který obsahuje odkazy na jiné zdrojové soubory pomocí direktivy {\$I };

Pracovní soubor (Work File), který je vkládán do paměti a se kterým lze běžně pracovat (tzn., že jej lze editovat, komplikovat i spouštět).

Při menších aplikacích se používá jen pracovní soubor. Pracovní soubor lze po odladění uložit na disk.

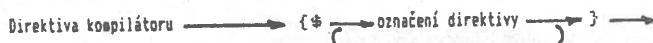
Použití hlavní souboru umožňuje, aby pro komplikaci byl připraven jeden hlavní soubor a v případě potřeby bylo možno pracovat s libovolným pracovním souborem. Pracovním souborem bude soubor, na který je v hlavní souboru učiněn odkaz pomocí direktivy {\$I }. Toto řešení zjednodušuje ladění

programů, neboť při zjištění chyby v některém z vkládaných souborů, se tento soubor automaticky vyvolává jako pracovní soubor do paměti a lze jej pomocí editoru opravit. Po ukončení opravy se pouze spustí komplikace. Opravený soubor je totiž automaticky ukládán na disk a také komplikace hlavního souboru se zahajuje automaticky.

#### 7.4.2 Direktivy kompilátoru

Činnost kompilátoru je řízena pomocí direktiv. Direktiva je dů programu vkládána jako poznámka se speciální syntaxí. Direktivu lze zapisovat do stejných míst jako poznámku.

Direktiva kompilátoru má tuto syntaxi:



Obr. 7.2 Syntaktický diagram direktivy kompilátoru

Všechny direktivy mají implicitní hodnotu, kterou není třeba uvádět. Výjimku tvoří direktiva I pro vkládané soubory.

Direktiva B ..... implicitně B+

řídí vstup/výstup. Když je aktivní, {\$B+}, tak se pro standardní soubory Input a Output používá zařízení CON. Když je pasivní, {\$B-}, tak TRM. Tuto direktivu lze použít jen pro celý program.

Direktiva C ..... implicitně C+

kontroluje interpretaci řídícího znaku při vstupu a výstupu z klávesnice. Když je aktivní, {\$C+} a při vstupu pomocí procedur Read nabo Readln se zapíše Ctrl-C, tak se přeruší zpracování. Když je pasivní, {\$C-}, pak řídící znak nemá význam. Tato direktivu lze použít jen pro celý program.

Direktiva I (vstup/výstup) ..... implicitně I+

řídí ošetření chyb vstupu/výstupu. Když je aktivní, {\$I+}, tak všechny operace vstupu/výstupu jsou automaticky kontrolovány. Když je pasivní, {\$I-}, tak kontrolu těchto chyb zajišťuje programátor pomocí standardní funkce IOResult (viz 7.4.3).

Direktiva I (vkládané soubory) .....

Jestliže direktiva I obsahuje jméno souboru, potom kompilátor zařazuje do komplikace vkládaný soubor daného jména (viz 7.3.1).

Direktiva R ..... implicitně R-

kontroluje hodnoty indexů během zpracovávání. Když je aktivní, {\$R+}, tak všechny indexy polí jsou ověřovány, zda jsou ve správném rozsahu. Když je pasivní, {\$R-}, tak se kontrola nezajišťuje.

Direktiva V ..... implicitně V+

řídí přenos parametrů typu řetězec předávaných odkazem. Když je aktivní, {\$V+}, tak délky řetězců v obou parametrech musí být stejné. Když je pasivní, {\$V-}, tak délka řetězců může být různá.

Direktiva U ..... implicitně U-

řídí uživatelská přerušení. Když je aktivní, {\$V+}, tak uživatel může přerušit zpracování programu kdykoliv pomocí Ctrl-C. Když je pasivní, {\$U-}, tak nemá žádný účinek.

Direktiva A ..... jen pro CP/M ..... implicitně A+

řídí vytváření absolutního (nerekurzivního) kódu. Když je aktivní, {\$A+}, tak absolutní kód je vytvářen. Když je pasivní, {\$A-}, tak kompilátor vytváří kód, který dovoluje rekurzivní volání.

Direktiva W ..... jen pro CP/M ..... implicitně W2

řídí hierarchickou úroveň příkazů With, tj. počet "otevřených" záznamů uvnitř bloku. Za W je možno uvést jednu číslici z intervalu 1 až 9.

Direktiva X ..... jen pro CP/M ..... implicitně X+

řídí optimalizaci polí. Když je aktivní, {\$X+}, tak vytvářený kód pro pole je optimalizován.. Když je pasivní, {\$X-}, tak kompilátor minimalizuje vytvářený kód.

7.4.3 Ošetření chyb

Při ladění programu je velmi důležité, jakým způsobem jsou indikovány chyby.

V Turbo Pascalu jsou rozlišovány tři skupiny chyb:

- chyby při kompliaci;
- chyby při zpracování a
- chyby při vstupu/výstupu.

Ke každé skupině chyb je k disposici samostatný chybník, i když většina chyb je jasné hlášena a srozumitelná bez dalšího výkladu. Turbo Pascal umožňuje překlad chybových

hlášení do libovolného jazyka, proto chybová hlášení se objevují v češtině.

#### Chyby při komplikaci

Při zjištění syntaktické chyby je komplikace přerušena a objeví se chybové hlášení. U většiny druhů chyb lze stisknut tlačítka Esc a potom se kurzor přemístí na místo v programu, kde byla indikována chyba.

#### Chyby při zpracování

Vyskytne-li se při zpracovávání programu neodstranitelná chyba, potom program je ukončen a objeví se chybové hlášení

Run-time error NN, PC = addr

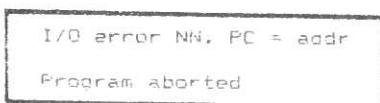
Program aborted

kde NN je číslo chyby a addr je je adresa místa, kde chyba byla indikována. Pozor, obě čísla jsou v hexadecimální soustavě. Po stisknutí tlačítka Esc je v některých případech možno místo chyby ve zdrojovém textu vyhledat a na toto místo umístit kurzor.

#### Chyby při vstupu/výstupu.

Při zpracování programu se může tež vyskytnout chyba vstupu/výstupu.

Je-li direktiva kompilátoru I aktivní, {#I+}, tak se při výskytu chyby program ukončí a objeví se chybové hlášení:



Kde NN je číslo chyby a addr je je adresa místa, kde chyba byla indikována. Pozor, obě čísla jsou v hexadecimální soustavě. Po stisknutí tlačítka Esc je v některých případech možno místo chyby ve zdrojovém textu vyhledat a na toto místo umístit kurzor.

Jestliže direktiva kompilátoru I je pasivní, {#I-}, tak potom chyby vstupu/výstupu může ošetřovat programátor pomocí funkce IDResult. Teto možnosti se využívá například pro zjištění, zda na disku je požadovaný soubor či nikoliv (viz 3.7.3).

## Literatura

- [1] Barbier, P.: Turbo Pascal: nejrychlejší na trhu.  
Microsystèmes č. 65, 1986, s. 142-145.
- [2] Dohnal, J. - Picková, E.: Základy výpočetní techniky.  
1. vyd. Praha, VŠZ 1986. 238 s.
- [3] Fišer, M. - Pátková, Z.: Mikro Pascal (Popis jazyka a jeho prostředí). Praha. Agrodat 1986. 417 s.
- [4] Havlíček, Z. - Vaněk, J.: Programování a programovací jazyky (Pascal) - cvičení. V přípravě.
- [5] Honzík, J.: Programovací techniky. JZD Slušovice 1986.  
357 s.
- [6] Jinoch, J. - Müller, K. - Vogel, J.: Programování v jazyku Pascal. 1. vyd. SNTL 1985. 263 s.
- [7] Kolektiv : Pascal. 1. vyd. Paříž, AFNOR 1983. 217 s.
- [8] Kolektiv : Turbo Pascal 3.0. Scotts Valley,  
Borland International Inc. 1985. 376 s.
- [9] Kolektiv : TNS - TURBO, CPM. JZD Slušovice 1986. 213 s.
- [10] Mašková, H.: Základy programování Pascal.  
1. vyd. Praha, SPN 1984. 160 s.
- [11] Müller, K.: Programovací jazyky. 1. vyd. Praha,  
ČVUT 1985. 168 s.
- [12] Pišvejc, J.: Programování a programovací jazyky I  
(Fortran). 1. vyd. Praha, VŠZ 1982. 283 s.
- [13] Remmeli, W.: Pascal systematisch. Berlin,  
Springer Verlag 1983. 237 s.
- [14] Vardell, L.: Pascal as second language. New York,  
Prentice-Hall 1984. 194 s.
- [15] Wirth, N.: Systematické programovanie (překlad).  
Bratislava, Alfa 1981. 200 s.



Název: Programování a programovací jazyky - PASCAL  
Autor: Doc. Ing. Zdeněk Havlíček, CSc.  
Určeno: Všem zájemcům o programování v jazyku Pascal  
Vydání: První  
Výtisků: 1000

Příručka neprošla jazykovou úpravou

Slušovice, 1988

Vydáno prostřednictvím podniku ČO ČSTV Sportpropag

Vytiskl METASPORT Ostrava 0684/88

JZD AGROKOMBINAT SLUŠOVICE  
nositel Řádu práce

ZAVOD APLIKOVANÉ KYBERNETIKY 2  
763 15 SLUŠOVICE  
CZECHOSLOVAKIA                    TELEX: 87 279

