Poznámky k přednášce
PROGRAMOVÁNÍ PRO FYZIKY

Úvod

Cílem přednášky je připravit studenta pro modelování fyzikálních problémů pomocí počítačů. Snažíme se předložit postupy charakterizované jako imperativní, strukturované a procedurální programování tak, aby jejich užitím byl student schopen zapisovat algoritmy (recepty, posloupnost kroků vedoucích od vstupních dat k výstupním) ve vhodném programovacím jazyku. Algoritmy čerpáme především z kolekce numerických metod potřebných v oboru. Pozornost věnujeme i postprocesingu a vizualizaci získaných výsledků.

Styly programování

Programování má řadu podob. Imperativní programování lze charakterizovat podrobným zápisem instrukcí (termín spojovaný s hardwarem) neboli příkazů (termín užívaný v programovacích jazycích), jimiž se mění stav dat uložených v paměti počítače. Na imperativní programování se časem nakladla omezení označovaná jako strukturované programování, spočívající v první řadě v sekvenčním užívání řídicích struktur (příkazových konstrukcí) se zřetelným vstupním a výstupním bodem; to má přispívat ke srozumitelnosti zdrojových textů programů pro oko programátora. Konstrukce strukturovaného programování se dají spočítat prsty jedné ruky (větvení a cykly), běžně jsou tolerované některé odchylky (skoky z konstrukcí, ošetření chybových stavů, tzv. výjimek); těžko odpustitelné jsou však obecné příkazy skoku (špagetový kód).

Dalšími paradigmaty (styly) programování jsou procedurální programování (PP) a objektově orientované programování (OOP). Procedurální styl zvýrazňuje akci s daty, kterou programátor vyjadřuje voláním procedury (odskokem s návratem); data v programu bývají organizována spíše jednoduše (pomocí datových struktur nazývaných pole a struktury neboli záznamy). Objektový styl se upíná k vytváření hierarchie tříd a objektů, vzájemně komunikujících posíláním zpráv; hlavní prací je zde příprava objektů zahrnujících jak data, tak metody (procedury) pro jejich zpracování. OOP přispívá ke zvládnutelnosti velkých projektů a je oblíbeno v informatické komunitě, PP jako starší paradigma je pro neprofesionála patrně přístupnější a bývá často užíváno pro menší projekty nebo programujícími experty v jiných oblastech, např. ve fyzice.

Do kontrastu k imperativnímu programování se klade deklarativní programování, kde důraz nespočívá na popisu algoritmů, ale ve vyjádření konečného záměru. S deklarativním stylem se lze potkat v numerických knihovnách, při paralelním nebo funkcionálním programování. Např. to upřednostňuje zápis programu jako obdoby matematické funkce, závislé pouze na svých parametrech a vracející návratovou hodnotu; do světa imperativních programovacích jazyků přesahuje funkcionální programování mj. rekurzivními funkcemi. Paralelní programování bývá v počítačové fyzice díky velikosti řešených problémů velmi aktuální; cílem je rozložit požadovanou práci i zpracovávaná data mezi více procesorů (jak mezi více hardwarových jader v rámci jednoho počítače, tak mezi více propojených počítačů), které pak práci provádějí souběžně v čase. Programátorovi zde pomáhají rozšíření programovacích jazyků pro vícevláknový běh a meziprocesovou komunikaci.

Programovací jazyky

Programovací jazyky běžně podporují více paradigmat, přičemž imperativní, strukturovaný a procedurální styl je minimem. Pro fyzikální aplikace se nejčastěji používají
– klasické programovací jazyky: Fortran (od 1957), C (od 1968), C++ (od 1983)
– skriptovací jazyky: Python, příkazový interpret Windows, shelly Unixu aj.
– systémy pro numerické modelování: Matlab, Octave, Mathematica, R apod., COMSOL, FEniCS apod.
a ty si fyzik dále rozšiřuje o
– knihovny pro numerické modelování: LAPACK a BLAS, mnoho algebraických řešičů, FFTW, IMSL, NAG aj.
– paralelizační systémy a knihovny: OpenMP, MPI, nástroje pro GPU aj.
– vizualizační software v 2D (gnuplot a mnoho jiných) a 3D (ParaView aj.)
Uvedený software bývá v dosahu jak pro osobní počítače s operačními systémy Windows nebo Linux, tak v počítačových centrech (př. www.it4i.cz/software), a C a Fortran se užívají i k programování moderních výpočetních koprocesorů (GPU). Programy psané v klasických programovacích jazycích se před spuštěním překládají do strojového kódu pomocí překladačů, skriptovací programovací jazyky se obvykle překládají po částech až za chodu svými interprety. Programátor může psát program v programátorském editoru (Visual Studio Code, Notepad++ aj.) a překládat voláním překladačů z příkazového řádku, nebo užívat k editaci, překladu i ladění integrované vývojové prostředí (Visual Studio, NetBeans, Lazarus, Spyder, Matlab aj.).

Programovací jazyk Pascal nemá podporu hlavních numerických a paralelizačních knihoven a v numerickém modelování se užívá zřídka. Byl (1970) navržen pro výuku (strukturovaného) programování a drží svou pozici právě jako takový. Aktuálně je udržován ve volně dostupné podobě Free Pascalu s vývojovým prostředím Lazarus pro Windows/Linux/Mac aj.

Terminologie ve světě informatiky klokotá a snadno se stane, že tatáž slova znamenají různé věci nebo že různými slovy je nazýváno totéž; to bývá ještě prohloubeno při hledání českých ekvivalentů k anglickým originálům. Už jen v předchozím textu tak stojí termín struktura pro příkazovou konstrukci i pro specifický typ datové struktury; dalším příkladem může být, že při procedurálním programování v jazyce C se nevolají procedury, ale funkce, a nejde přitom o funkcionální programování. Navíc, ne každý text o programování je psán odpovědně a může (a to asi i tento text) svádět nevinné dívky a chlapce natolik, až se v jejich programech blbě orientuje; hovoříme pak o blbě orientovaném programování (BOP).

První ukázky

Předkládáme ukázky s výpočtem součtu aritmetické posloupnosti imperativně i deklarativně, strukturovaně i špagetově, procedurálně i objektově, a také paralelně, a to úmyslně v jazyce Opravdového programátora (google: Real Programmer), ve Fortranu. Pověsti o jeho zastaralosti jsou zjevně zastaralé.

Matematická úloha

$$S_n = \sum_{i=1}^{n} i$$

Ukázka 1: Matematická optimalizace. Místo sumace o lineárním počtu operací, O(n), řešíme úlohu s konstantním počtem operací, O(1), co do výsledku ekvivalentní: $S_n = \frac{n(n+1)}{2}$. Sekvenčně vykonávaný program získá příkazem vstupu vstupní data, přiřazovacím příkazem přiřadí výsledek výrazu na pravé straně do paměťové proměnné na levé straně a příkazem výstupu vypíše řešení úlohy. Celý program je zde tvořen jedinou programovou jednotkou, a ta je zakončena koncovým řádkem. V programu nešetříme komentáři.

```
read *,n          ! prikaz vstupu
s=n*(n+1)/2       ! prirazovaci prikaz
print *,n,s       ! prikaz vystupu
end
```

Ukázka 2: Spaghetti code. Sčítáme řadu, tedy n-krát posuneme hodnotu indexu (řídicí proměnné) a n-krát přičteme aktuální člen řady k proměnné udržující součet. Místo příkazových konstrukcí jsou zde příkazy skoku na návěští, takže hlavně takhle už nikdy.

```
read *,n
i=0; s=0
111 if (i>=n) goto 999  ! radek s navestim, podmineny prikaz, prikaz skoku
i=i+1; s=s+i
goto 111
999 print *,n,s
end
```

Ukázka 3: Strukturované programování. Konstrukce pro cyklus s podmínkou a indexovaný cyklus (s řídicí proměnnou). Blok v cyklu do while se provádí tak dlouho, dokud je podmínka splněna. Indexovaný cyklus do koná totéž; prvotní nastavení indexu, jeho zvyšování o (zde jednotkový) krok a testování proti předepsané konečné (zde horní) mezi je skryto pod kapotou; indexovaný cyklus je tak o trochu víc deklarativní než cyklus s podmínkou.

```
read *,n
! cyklus s podminkou
i=0
s=0
do while (i<n)
  i=i+1
  s=s+i
end do
print *,n,s
```

```
! indexovany cyklus
s=0
do i=1,n
  s=s+i
end do
print *,n,s

end program
```

Ukázka 4: Procedurální programování. Některé programovací jazyky pracují s procedurami (podprogramy), jiné s funkcemi, některé s obojím. Pokud jazyk umí obojí, pak by se funkce měly podle jejich matematického vzoru užívat k umístění jejich návratové hodnoty na místo volání ve výrazu a jinak by neměly ovlivnit nic. Procedury toho mohou konat více, např. měnit hodnoty svých argumentů (parametrů). V obou případech jde o odskok s návratem za místo volání.

```
read *,n
call sum(n,s)          ! odskok do procedury, navratova hodnota se vraci jako argument
print *,n,s
print *,n,fsum(n)      ! odskok do funkce, navratova hodnota se dosazuje na misto volani
end program

! procedura
subroutine sum(n,s)
s=0
do i=1,n
  s=s+i
end do
end subroutine
```

```
! funkce
function fsum(n)
fsum=0
do i=1,n
  fsum=fsum+i
end do
end function
```

Ukázka 5: Funkcionální programování. Jednak, s trochou štěstí, můžeme v programovacím jazyce nalézt funkci pro požadovanou akci a ani nemusíme vědět, jak taková černá skříňka výsledek získá. Jednak lze mnohé algoritmy vyjádřit i jinak, někdy pohodlněji, než konstrukcemi strukturovaného programování. Zde velíme počítat naši úlohu podle předpisu $S_0 = 0,\ n > 0 :\ S_n = n + S_{n-1}$ a necháváme na překladači, aby se s tím nějak utkal.

```
read *,n
! standardni funkce jazyka
print *,n,sum([(i,i=1,n)])

! rekurzivni funkce
print *,n,RecursiveSum(n)

contains
```

```
recursive function RecursiveSum(n) result (r)
if (n>0) then
  r=n+RecursiveSum(n-1)
else
  r=0
endif
end function
end program
```

Ukázka 6: Objektově orientované programování. Vytvoří se třída (objektový typ) obsahující data a metody pracující s těmito daty; zde třída obsahuje horní mez řady a funkci pro výpočet součtu řady. Budování tříd bývá pracné, ale často je to už někde hotové a programátor to svému programu jen zpřístupní. Dál je to vlastně celkem pohodlné: programátor vytvoří objekt (proměnnou), konstruktorem třídy ho inicializuje, aplikuje na něj patřičnou metodu a výsledek je na světě.

```
module mCislo          ! modul
type tCislo            ! popis tridy
  integer n            ! data
contains
  procedure sum        ! metody
end type

contains

function sum(this)     ! popis metody
class(tCislo) this     ! popis argumentu
sum=0
do i=1,this%n          ! k datum tridy
  sum=sum+i
end do
end function
end module
```

```
! hlavni program
program p06
use mCislo             ! pripojeni modulu
type(tCislo) cislo     ! popis objektu
read *,n
cislo=tCislo(n)        ! volani konstruktoru
print *,n,cislo%sum()  ! volani metody
end program
```

Ukázka 7. Paralelní programování. Standardní (původně sekvenční) indexovaný cyklus je obložen direktivami systému OpenMP, které sdělují překladači, že se cyklus má rozdělit mezi několik softwarových vláken, o které se za běhu programu postará souběžně v čase několik hardwarových jader procesoru. Doplňkovou klauzulí se zajistí, aby se mezisoučty získané jednotlivými vlákny nakonec sečetly (redukovaly) do jediného výsledku. Direktivy OpenMP se řadí k deklarativnímu stylu; píšeme, co chceme získat, nepíšeme podrobně, jak.

```
read *,n
s=0
!$OMP PARALLEL DO REDUCTION (+:s)
do i=1,n
  s=s+i
end do
!$OMP END PARALLEL DO
print *,n,s
end program
```

Na webu přednášky jsou ke stažení tyto ukázky nejen ve Fortranu, ale i v C/C++, Pascalu, Pythonu, Matlab/Octave a příkazových interpretech Windows a Linuxu, pokud je v nich lze zapsat. Můžeme tak pozorovat, jak se konstrukce strukturovaného programování ve všech jazycích podobají; jak se v daném jazyce může prolínat více programovacích stylů; jak je třeba se rozmáchnout při psaní i malého objektového programu oproti procedurálnímu ekvivalentu; jak se OpenMP paralelizace zakousne do indexovaného cyklu nejen ve Fortranu, ale i v C. Stojí také za časnou zmínku, že klasické jazyky nutí programátora uvádět seznam používaných proměnných (Fortran tak lze také nastavit), zatímco skriptovací jazyky to obvykle nežádají.

Software a literatura

Viz geo.mff.cuni.cz/~lh/NOFY056/

Základní pojmy

Přehled pojmů

Programovací jazyky (Pascal, C, Fortran aj.) bývají kodifikovány mezinárodními normami (Pascal: ISO 7185, C: ANSI C89, C99, Fortran: 77, 95, 2003 aj.), které popisující syntaxi (formální pravidla pro zápis programů) a sémantiku (význam) jazykových konstrukcí. Ve větším či menším souladu s těmito normami softwarové systémy zvané překladače překládají zdrojové texty programů do strojového kódu.

K zápisu zdrojových textů se používá abeceda jazyka (velká i malá písmena, číslice, speciální znaky). Stavebními kameny zdrojových textů jsou klíčová slova jazyka (Pascal: program, begin, end, if, for, while aj.) a jména (kombinace písmen a číslic zahájená písmenem) veličin zaváděných programátorem či definovaných překladačem. Zápis zdrojového textu je vhodné pro zvýšení srozumitelnosti podřídit zavedeným konvencím (komentování, odsazování, mnemotechnická jména, používání velkých a malých písmen aj.).

Pro veličiny deklarované v programu jsou k dispozici standardní datové typy, charakterizující velikost veličiny v paměti počítače (1, 2, 4, 8, 16 bytů) a obor hodnot, kterých veličina může nabývat. Ke standardním datovým typům patří typ celočíselný (integer, C: int), reálný neboli s plovoucí tečkou (real, single, double, C: float, double), komplexní (Fortran: complex), logický (boolean, Fortran: logical) a znakový (char, Fortran: character). Pro příklad, 4bytový (32bitový) integer umožňuje zachytit $2^{32} \sim 4 \times 10^9$ různých celočíselných hodnot, volitelně se znaménkem nebo bez znaménka; 8bytový double poskytuje pro mantisu 15 platných míst a desítkové exponenty v rozsahu -324 až +308. Reálné typy zahrnují i speciální hodnoty pro nekonečno (±Inf) a pro matematicky nedefinované výsledky (NaN, Not-a-Number). Literály (doslovné hodnoty) jsou konstanty jednotlivých datových typů reprezentující samy sebe, např. integer: 1, -2, real: 1., -2.2, 3.402823e+38, fortranský complex: (0.,1.), boolean: false, true, char: 'a', '1', '+'. Z veličin o standardních datových typech lze vytvářet veličiny strukturovaných typů, v první řadě pole (typ složený z prvků téhož typu) a záznamy/struktury (typ složený z položek různých typů). Často je nutno provádět typové konverze, při nichž se veličina daného typu smysluplně překóduje do reprezentace odpovídající jinému typu; tyto konverze se někdy provádějí automaticky (při vyhodnocování výrazů, v přiřazovacích příkazech), jindy ne.

Pro pojmenované veličiny se kromě datového typu deklaruje i to, zda budou symbolickými konstantami (Pascal: const), tedy po počáteční inicializaci již neměnné, nebo proměnnými (Pascal: var), jimž náležející paměťová místa lze za chodu programu přepisovat. V četných kontextech jsou užitečné ukazatele, proměnné neobsahující ve svém paměťovém místě datovou hodnotu, ale adresu jiného paměťového místa, na které tak ukazují.

V programu jsou často vyhodnocovány výrazy, složené z operandů a operátorů. Výrazy číselných typů obsahují aritmetické operátory (aditivní +, −, multiplikativní *, /), k ruce jsou též operátory znakové (+, Fortran: //) a logické (not, and, or aj.); relační operátory (=, <>, <, <=, >, >=; první dva v C: ==, !=, ve Fortranu: ==, /=) porovnávají kompatibilní operandy všech standardních typů. Jsou definovány priority operátorů, v případě dílčích výrazů o totožných prioritách často i směr jejich vyhodnocování.

Program je v podstatě tvořen jednak výše zmiňovanými deklaracemi proměnných a symbolických konstant, dále mj. deklaracemi odvozených datových typů, jednak příkazy pro řízení chodu programu, které mohou nabýt i složitější podoby tzv. příkazové konstrukce. Jsou to především přiřazovací příkaz, podmíněné příkazy pro větvení programu do dvou (if) nebo více větví (case), příkaz cyklů s podmínkou (while, repeat) a indexovaného cyklu (for, Fortran: do), příkaz volání procedury (jejím jménem, Fortran: call) a příkazy obecného a speciálních skoků (goto; continue, break, exit, Fortran: cycle, exit, return). V Pascalu jsou deklarace a příkazy spojovány do tzv. bloků, které (spolu se svými hlavičkami) tvoří funkce a procedury korespondující s funkcemi v C a programovými jednotkami ve Fortranu. Funkce a procedury jsou typickou oblastí platnosti jmen v nich deklarovaných.

Zdrojový text programu (v Pascalu) je strukturován jednak jako sada procedur a funkcí vnořených do hlavního programu, jednak pomocí samostatných modulů obsahujících kromě modulových dat i vnořené funkce a procedury. Moduly se pak připojují k těm programovým jednotkám, které data, funkce a procedury obsažené v modulech potřebují. Programové jednotky a moduly mohou být rozloženy do více zdrojových souborů.

Jednotky informace

Veličina o velikosti 1 bit (binary digit, zkratka b) může nabývat 2^1 hodnot (0..1). Byte (zkratka B) se usadil na 8 bitech a může zachytit $2^8 = 256$ hodnot (0..255). Slovo je typicky velikost adresy v paměti počítače, a také velikost obvyklých číselných datových typů, a dnes je buď 32bitové (4bytové), umožňující popsat $2^{32} = 4 \times 2^{30} \sim 4 \times 10^9$ hodnot, nebo 64bitové (8bytové) pro 2^{64} hodnot. Bit i byte mohou být zvětšovány předponami kilo K (nebo k), mega M, giga G a tera T. Ty se mohou chápat buď jako 10^3, 10^6, 10^9 a 10^{12} násobky nebo jako násobky odvozené z dvojkové soustavy, $2^{10} = 1024 \sim 10^3$, $2^{20} \sim 10^6$, $2^{30} \sim 10^9$, $2^{40} \sim 10^{12}$, lišící se od předchozích o 2 až 10 %. V jejich druhém, dvojkovém významu se mají číst jako kibi (kilo-binary), mebi, gibi a tebi, ale kdo by to dělal. Spíš je třeba pokračovat k peta P pro 10^{15}, neboť touto příponou se doplňuje jednotka FLOPS (floating-point operations per second) při poměřování výkonu soudobých superpočítačů, en.wikipedia.org/wiki/TOP500.

Příkazy

Příkazy a složitější příkazové konstrukce se v programu psaném v duchu strukturovaného programování provádějí sekvenčně. Frekventovaný je přiřazovací příkaz, kterým se vyhodnotí výraz a uloží jeho výsledek. K dočasnému vytvoření dvou nebo více větví v programu jsou určeny podmíněné příkazy. Příkazy cyklu zajišťují opakování částí programu. Volání procedury provede odskok s návratem. Jiné příkazy skoku (až na několik výjimek) vnášejí do programu nepřehlednost a do strukturovaného programování nepatří. V programovacích jazycích se vyskytují i další specifické příkazy: prázdné a složené, vstupní a výstupní, alokační a dealokační a jiné. Příkazy se oddělují oddělovačem (často středník, někdy jen konec řádku). Různé programovací jazyky poskytují obdobné příkazy, samozřejmě každý jinak. Níže preferujeme ukázky v Pascalu (novější konstrukce ve Free Pascalu) a příležitostně zmiňujeme C, Fortran, Octave a Python.

Přiřazovací příkaz

Je určen pro vyhodnocení výrazu na pravé straně a přiřazení výsledku do cíle na levé straně. Cílem je obvykle jednoduchá (skalární) proměnná, cenná jsou zobecnění příkazu pro strukturované proměnné.
Syntaxe v Pascalu: levá strana := pravá strana;
V C, Fortranu a mnohých jiných jazycích postačí místo symbolu := pouhé rovnítko, =.
Při statických deklaracích datových typů proměnných

```
var n : integer; x : real; b : boolean; ch : char;
```

můžeme tyto proměnné inicializovat triviálním přiřazením literálů,

```
n:=1; x:=3.1415926; b:=true; ch:='A';
```

nebo můžeme zachytit výsledky složitějších výrazů,

```
n:=1+1; x:=arctan(1)*4.; b:=not true; ch:=succ(ch);
```

(Př. +1) Velmi častým přiřazením je zvětšení hodnoty proměnné o 1,

```
n:=n+1; x:=x+1.;
```

Pro tuto akci existují v jazycích různé zkratky: v Pascalu (pro integer, nikoliv pro real) inc(n), v C, Octave a Pythonu (pro integer i real) lze místo n=n+1 psát n+=1 a dokonce (v C a Octave) jen n++ nebo ++n.
Typové konverze. Zatím jsme na levé straně používali proměnnou datového typu identického jako typ výrazu na pravé straně. V klasických jazycích, kde proměnné mají staticky deklarovaný datový typ, je přiřazení schopno provádět implicitní konverzi datového typu výsledku z pravé strany na typ proměnné na levé straně. Přípustné konverze se shrnují pod pojem kompatibilita vzhledem k přiřazení a obvykle jsou to tyto: integer na real, malý integer na větší, real na complex, char na string apod. Pascal odmítne automatický převod real na integer, desetinnou část je nutno explicitně oříznout funkcí trunc nebo zaokrouhlit funkcí round:

```
x:=n; {nelze n:=x;} n:=trunc(x+0.5); n:=round(x+0.5);          // přiřazované hodnoty: 3.0, 3, 4
```

V jiných jazycích může být přiřazení n=x přípustné; v C a Fortranu to lze a implicitně se ořezává.
Dynamický datový typ. Ve skriptovacích jazycích proměnné běžně získávají datový typ dynamicky. Jednou z cest je přiřazovací příkaz: v Pythonu sekvence přiřazení a výpisů
(Python) a=1; type(a); a=1.; type(a); a=1+0j; type(a);
dokládá, jak s každým přiřazením nastane změna datového typu cíle, nikoliv konverze typu přiřazované hodnoty.
Zobecněná přiřazení. Dosud jsme na levé straně používali skalární proměnné. Uvidíme, že existují zobecněná přiřazení do cílů strukturovaných typů – běžně do prvků polí či položek záznamů, a někdy i do polí a záznamů jako celku nebo do jejich částí. Fortran a Octave mají při práci s poli obzvláštní sílu; např. pole tam lze sčítat příkazem c=a+b, zatímco v Pascalu je pro totéž nutný cyklus, for i:=1 to n do c[i]:=a[i]+b[i].
Symbolické konstanty. Přiřazenou hodnotu musí samozřejmě mít i symbolické konstanty, u kterých se to však děje nikoliv přiřazovacím příkazem, ale už pomocí inicializačního výrazu v deklaraci,

```
const z=0;
```

Řetězení. Někdy lze přiřazovací příkazy řetězit, např. v C, Octave či Pythonu příkaz b=c=d přiřadí hodnotu d do proměnných c i b. Jindy má podobná konstrukce význam porovnání c s d a přiřazení výsledku do b; to by však v Pascalu muselo být psáno při boolean b a real c,d jako b:=c=d; a ve Fortranu jako b=c==d.
Přiřazování adres. Přiřazovací příkaz může být pojat i jinak – nemusí přiřazovat do proměnné na levé straně hodnotu z pravé strany, ale paměťovou adresu, na které se ona hodnota nachází. Proměnná je tak ukazatelem na adresu s hodnotou. Někdy se toto alternativní pojetí přiřazovacího příkazu zvenku neprojevuje, jindy ano – třeba tehdy, když na jednu adresu ukazují dva ukazatele. Změna hodnoty pomocí jednoho ukazatele se pak projeví i při pohledu skrz druhý ukazatel. Např. statická pole v Pascalu se přiřazují hodnotou, sekvence a[0]:=0; b:=a; a[0]:=1; nezmění hodnotu 0 v b[0], zatímco dynamická pole se přiřazují adresou, po sekvenci c[0]:=0; d:=c; c[0]:=1; je v d[0] hodnota 1. Octave upřednostňuje přiřazování hodnot: a=[0]; b=a; a(1)=1; b ponechá v b nulu, Python přiřazování adres: a=[0]; b=a; a[0]=1; b ukáže v b jedničku.

Prázdný a složený příkaz

Objevují se pro uspokojení formálních či estetických požadavků programovacích jazyků. V Pascalu a C je prázdný příkaz téměř neviditelný a složený příkaz nevyhnutelný, ale jde to i jinak: ve Fortranu zní prázdný příkaz continue a složený příkaz neexistuje.

Syntaxe v Pascalu – prázdný příkaz: nic, složený příkaz: begin commands end. Složený příkaz v C: { }.

Př. `begin end` je složený příkaz obsahující jeden prázdný příkaz,
`begin ; end` je složený příkaz obsahující dva středníkem oddělené prázdné příkazy.

Př. Swap. Výměna obsahu dvou proměnných může nabýt podoby jednoho z následujících minialgoritmů, vložených do složených příkazů:

```
begin t:=x; x:=y; y:=t end;          // přesuny s pomocnou proměnnou
begin x:=x+y; y:=x-y; x:=x-y end;    // přesuny s aritmetickými operacemi
```

Podmíněný příkaz if

Slouží k podmíněnému provádění příkazů neboli k dvoucestnému větvení podle podmínky, tj. podle výsledku (true/false) logického (boolean) výrazu.

Syntaxe v Pascalu: plná `if expressionBoolean then commandTrue else commandFalse;`
zkrácená `if expressionBoolean then commandTrue;  // bez else-větve`

C, Fortran: obdobně, s uzavřením logické podmínky v kulatých závorkách. Fortran vyžaduje koncové endif.

Př. Minimum. Nalezení menší ze dvou hodnot, ekvivalent funkce min(x,y):

```
if x<y then min:=x else min:=y;
```

Větve. Větve příkazových konstrukcí obvykle zahrnují několik příkazů. V Pascalu i C přitom bývá ve větvi formálně přípustný jen jediný příkaz, tedy jím musí být složený příkaz. Ve Fortranu a Octave jsou větve ukončeny koncovými příkazy end, složený příkaz tak není potřebný. V Pythonu se větve vyznačují odsazením a nepotřebují ani složený, ani koncový příkaz.

(Pascal)	`if test then begin cmd1; cmd2 end;`	(C)	`if (test) {cmd1; cmd2};`
(Fortran)	`if (test) then; cmd1; cmd2; endif;`	(Octave)	`if test, cmd1; cmd2, end`
(Python)	`if test: || __cmd1 || __cmd2`	(zde || značí nový řádek a __ zvýrazňuje odsazení)	

Podmíněný výraz. Je určen pro větvená přiřazení do téže proměnné (hojný výskyt v jazycích, v Pascalu však není). K příkazu s podmíněným přiřazením do proměnné x, `if exprBool then x:=exprT else x:=exprF;` lze psát alternativu pomocí podmíněného operátoru `?:` (C), `x=exprBool ? exprT : exprF`, pomocí funkce `merge` (Fortran), `x=merge(exprT,exprF,exprBool)`, anebo ještě jinak (Python), `x=exprT if exprBool else exprF`.

Vnořování a řetězení. Vnořování podmíněných příkazů do větví nadřazených podmíněných příkazů je možné, ale spíše nepřehledné, zvláště zde bez odřádkování a odsazování:

```
if exprB1 then begin if exprB2 then cmdT2 else cmdF2 end else begin if exprB3 then cmdT3 else cmdF3 end;
```

Řetězení else-if je vnořování příkazu if do větve else předchozího if, tedy ještě dobře čitelná obecná konstrukce pro vícecestné větvení:

```
if exprB1 then cmdT1 else if exprB2 then cmdT2 else cmdF2;
```

Př. Sign. Pro získání znaménka numerického výrazu slouží funkce sign(x), vracející ±1 pro kladné či záporné hodnoty a 0 pro nulu. Implementace pomocí řetězení:

```
if x>0 then s:=1 else if x=0 then s:=0 else s:=-1;
```

Vícecestné větvení. Větvení podle více než dvou hodnot jedné proměnné lze zapsat pomocí řetězení s několika podmínkami,

```
if n=1 then cmd1 else if n=2 then cmd2 else cmd3;
```

ale v takovém případě je vhodnější užít následující specializovaný podmíněný příkaz.

Podmíněný příkaz (přepínač) case

Slouží k vícecestnému větvení podle hodnot výrazu ordinálního datového typu (integer, boolean, char, nikoliv real).

Syntaxe v Pascalu:
```
case expressionOrdinal of listOfConstants1: command1;
                          listOfConstants2: command2; ...;
else commands
end;                    // listOfConstants: konstanty nebo intervaly oddělené čárkou
```

V C a Octave se příkaz nazývá switch, ve Fortranu select case.

Větvení podle ordinálního výrazu umožňuje vyhodnotit výraz právě jednou a vskočit rovnou do vhodné větve. Totéž pomocí if je méně efektivní i méně přehledné.

Př.
```
case n of 1: cmd1; 2: cmd2; else cmd3 end;                  // vícecestné větvení (varianta příkladu s if)
case DoW of 1..5: VsedniDen; 6,7: Vikend; else Chyba end;   // day of week
case n of -MaxInt-1..-1: s:=-1; 0: s:=0; 1..MaxInt: s:=1 end;  // ekvivalent funkce sign(n) pro integer n
```

Příkazy cyklu s podmínkou while a repeat

Používají se pro opakované provádění (iterování) části programu – těla cyklu, přičemž počet opakování se odvíjí od hodnoty podmínky čili logického výrazu na začátku (příkaz while) nebo konci cyklu (příkaz repeat). Cyklus lze opustit nastavením patřičné hodnoty podmínky nebo předčasně pomocí příkazů skoku.

Syntaxe v Pascalu – varianty s podmínkou na začátku a na konci:

```
while exprBoolRun do command;          // dokud je exprBoolRun true, prováděj command
repeat commands until exprBoolBreak;   // opakuj commands, než bude exprBoolBreak true
skoky: continue; break;                // pokračuj novou iterací; skoncuj cyklus
```

C: while, do...while, Fortran: do while...enddo, Octave: while...end, do...until, Python: while...else. V C, Octave a Pythonu jsou příkazy skoku rovněž continue a break, ve Fortranu se používá cycle a exit.

Nekonečný cyklus. Někdy programátor chce zadržet program napořád, jindy si totéž způsobí neúmyslně – nekonečné cykly jsou denním chlebem. Postačí, když je v cyklu while podmínka stále true nebo v cyklu repeat false:

```
while true do;
repeat until false.
```

Obrácením logiky lze simulovat prázdný příkaz: while false do; repeat until true.

Př. Počítačové epsilon. Víme, že $1 + 1/2^n > 1$ pro konečné n. Počítači to je jedno, jeho procesor zvládá reálnou aritmetiku jen s omezenou přesností. S jak omezenou, to napoví počítačové epsilon neboli nejmenší kladné číslo, pro které 1+eps>1. Zjistíme ho cyklem s podmínkou, v jehož těle se půlí hodnota v proměnné eps:

(Pascal) `eps:=1; while 1+eps*0.5>1 do eps:=eps*0.5; writeln(eps);`

Je to spíš jen náhoda, že zrovna Pascal odpoví hodnotou přibližně 5×10^{-20}, což je nejmenší (a tedy nejlepší) možná odpověď na procesorech dnešních osobních počítačů. Ty nabízejí 3 úrovně přesnosti reálné aritmetiky (single, double, extended) a různé programovací jazyky implicitně zpřístupňují různé z nich. C, Octave a Python s podobným programem vrátí 1×10^{-16} a Fortran 6×10^{-8}. Horší výsledky než v Pascalu nejsou ostudou, prostě si jen programátor musí být vědom a případně nastavit, která přesnost reálných počtů se v jeho programu použije.

(Octave)
```
eps=1; while 1+eps*0.5>1, eps=eps*0.5; end; eps                 // dvojitá přesnost (double)
eps=single(1); while 1+eps*0.5>1, eps=eps*0.5; end; eps         // jednoduchá přesnost (single)
```

(Python)
```
eps=1
while 1+eps*0.5>1:
  eps=eps*0.5
print(eps)
```

Skoky. Iterace lze opouštět předčasně pomocí speciálních skoků. Skok continue ukončí aktuální průchod těla cyklu a pokračuje novou iterací, skok break přeruší celý cyklus a pokračuje za ním.

Př. repeat pomocí while. Ne každý z námi sledovaných jazyků se namáhá nabídnout cyklus s podmínkou na konci. Ten, podobně jako cyklus s podmínkou někde uvnitř, lze ovšem simulovat pomocí nekonečného cyklu while, podmíněného příkazu if a skoku break:

```
repeat cmd until exprBoolBreak;                        // opakuj cmd, než nastane exprBoolBreak
while true do begin cmd; if exprBoolBreak then break; end;   // alternativa pomocí while
```

Příkaz indexovaného cyklu (s řídicí proměnnou) for

Používá se pro opakované provádění těla cyklu, přičemž počet iterací se stanoví v hlavičce cyklu pomocí výchozí a konečné meze indexu (řídicí proměnné). Index je obvykle celočíselný, krok často jednotkový, kladný i záporný. Cyklus lze opustit předčasně pomocí příkazů skoku.

Syntaxe v Pascalu – varianty s rostoucím a klesajícím krokem 1:

```
for i:=lowerBound to upperBound do command;     // prováděj command pro i od dolní do horní meze
for i:=upperBound downto lowerBound do command; // prováděj command pro i od horní do dolní meze
skoky: continue; break;                         // pokračuj iterací s následujícím i; skoncuj cyklus
```

C: značně obecný for, Fortran: do...enddo s volitelným krokem (nejen ±1), Octave: for...end, Python: for...else. Příkazy skoku v C, Octave a Pythonu: continue a break, ve Fortranu: cycle a exit.

Př. for pomocí while. Pro zvýraznění, co všechno cyklus for vlastně dělá, zapíšeme příkaz for i:=lb to ub do cmd pomocí cyklu while. Není to přesný ekvivalent – obě varianty se mohou ve specifických situacích chovat různě.

```
i:=lb;                  // nastav dolní mez indexu
while i<=ub do begin    // opakuj, dokud je index menší než nebo stejný jako horní mez
  cmd;                  // proveď příkaz
  inc(i);               // přejdi k následující hodnotě indexu
end;
```

Hodnota indexu po ukončení cyklu. Různé jazyky a jejich překladače implementují cyklus for různě. Např. po cyklu s kladným krokem a lb>ub, jehož tělo se neprovádí, může index udržet svou předchozí hodnotu (Free Pascal) nebo

nabýt lb (příklad s while výše) nebo ub. Po řádném doběhnutí cyklu, tedy po ub–lb+1 iteracích, může mít index hodnotu ub (Free Pascal) nebo ub+1 (příklad s while výše). Po ukončení skokem index zachovává aktuální hodnotu.
Skoky. Iterace lze opouštět předčasně pomocí speciálních skoků. Skok continue ukončí aktuální průchod těla a pokračuje iterací s následující hodnotou indexu, skok break přeruší celý cyklus a pokračuje za ním. Není možné nebo vhodné snažit se o podobný efekt změnou hodnoty indexu (v jazycích často nepřípustné) nebo konečné meze (často neúčinné). Např. tělo cyklu lb:=1; ub:=10; for i:=lb to ub do begin ub:=0; writeln(i) end proběhne 10krát, bez přihlédnutí ke změně horní meze uvnitř cyklu; počet opakování cyklu byl totiž stanoven pevně podle hodnot mezí při vstupu do cyklu.
Reálný index. Některé jazyky umožňují použití nejen celočíselného indexu. Pascal připouští index ordinálního datového typu (integer, boolean, char): při var z : char lze vypsat abecedu cyklem for z:='a' to 'z' do write(z). Někdy se mohou hodit i cykly s reálným indexem, jsou však riskantní a v některých jazycích (v Pascalu) zakázané.

```
(C)        double x; for (x=0; x<=1; x=x+0.1) { printf("%25.20f\n",x); } // double precision
(Fortran)  real x; do x=0.,1.,.1; print *,x; enddo; end                  // single precision
(Octave)   format long; for x=0:.1:1; disp(x); end; for x=0:single(.1):1; disp(x); end;
```

Tyto cykly mají postupovat od 0 do 1 s reálným krokem 0.1. Bohužel proběhnou někdy 10krát, jindy 11krát, a poslední vypsaná hodnota může být větší než horní mez. Vidíme tu opět v akci reálnou aritmetiku s omezenou přesností: hodnotu 0.1 nelze reprezentovat přesně, platí pro ni: přesná 0.1 < 0.1 v přesnosti double < 0.1 v přesnosti single, a při kumulaci těchto 0.1 do proměnné x probíhá zaokrouhlování v různých přesnostech různě.
Vnořování cyklů. Cykly lze vnořovat. Časté je to při práci s vícerozměrnými poli, např. dvourozměrnou matici lze v Pascalu inicializovat takto: for i:=imin to imax do for j:=jmin to jmax do m[i,j]:=0. Konstrukce se záměnou vnějšího a vnitřního cyklu je svým výsledkem ekvivalentní: for j:=jmin to jmax do for i:=imin to imax do m[i,j]:=0, může zde však být podstatný rozdíl v efektivitě (době provedení). Při zacházení s vícerozměrnými poli je podstatné snažit se o sekvenční průchod pamětí, což v Pascalu a C upřednostňuje první variantu a ve Fortranu a Octave druhou (více v části o polích). Oč je jednodušší, když se takové věci dají zapsat bez cyklů, jako ve Fortranu a Octave: m(imin:imax,jmin:jmax)=0 nebo rovnou m=0.
Př. Kombinovaný cyklus. Pro cyklus omezený logickou podmínkou i limitem na počet průchodů máme na výběr:

```
for i:=lb to ub do begin if exprBoolBreak then break; cmd end;          // cyklus for a podmínka if
while not exprBoolBreak do begin cmd; i:=i+1; if i>ub then break end; //cykluswhile a ruční správa indexu
```

Příkaz cyklu for..in („foreach")

Používá se pro opakované provádění těla cyklu pro každý prvek daného pole nebo některých dalších datových typů. V těle cyklu lze použít symbolické jméno prvku, nikoliv indexu.
Syntaxe ve Free Pascalu: for x in a do cmd;
V C++ nalezneme zobecněný for, v Octave a Pythonu pokrývá syntaxe předchozí i tuto variantu.
Př. Průchod polem. Posuďme úspornost dvou variant, nepotřebujeme-li hodnotu indexu výslovně:

```
(Free Pascal)  for i:=1 to 5 do a[i]:=i; for i:=1 to 5 do writeln(i,a[i]);          for x in a do writeln(x);
(C++)          for (int i=1; i<6; i++) { a[i-1]=i; printf("%i %f\n",i,a[i-1]); }  for (float x : a) cout << x << endl;
(Octave)       a=1:5;                   for i=1:numel(a), disp([i,a(i)]); end;     for x=a, disp(x); end;
(Python)       a=range(1,6);            for i,x in enumerate(a): print(i,x,a[i]); for x in a: print(x);
```

Příkaz volání procedury

Poznávacím znamením procedurálního programování je rozprostření dílčích kroků programu do procedur. Příkazem volání procedury se do ní odskočí, po jejím provedení se chod programu vrací na příkaz následující za voláním.
Syntaxe v Pascalu: jmenoProcedury(argumenty);
V Pascalu (a ve Fortranu) se odskakuje i do funkcí, jejich volání však není příkazem, ale výrazem. Někdy procedury formálně splývají s funkcemi (v C, Octave a Pythonu jsou jen funkce) a volají se výrazovým příkazem (tj. pouhým jménem, jako v Pascalu); lze je pak použít i jako součást výrazu, neboť mají definovanou návratovou hodnotu.
Př. Přičtení 1, převod řetězce na číslo, výpis řádku – to vše jsou v Pascalu volání standardních procedur:

```
inc(n); val('3.14',x,ierror); writeln('pi ~ ',x);
```

Předčasné ukončení. Z procedur se vrací po provedení jejich posledního příkazu nebo předčasně pomocí skoku exit, pomocí halt se předčasně ukončí chod celého programu. (C: return a exit, Fortran: return a stop.)

Příkazy skoku. Výjimky

Obecný příkaz skoku je ve strukturovaném programování hlavním otloukánkem. Jazyky ho poskytují v podobě příkazu goto label, kde návěští label je (v Pascalu) identifikátor nebo číselný kód, označující řádek v aktuální programové jednotce. Na ten se má nesekvenčně přeskočit. Je radno se skoku goto vyhýbat a ve specifických kontextech jej nahrazovat vhodnějším příkazem. Viděli jsme skoky pro předčasné ukončení cyklu (continue, break) a předčasné opuštění procedury (exit).

Další situací vyvolávající potřebu někam si odskočit je vznik chybového stavu programu neboli vyvolání softwarové výjimky (exception). Překladače vkládají do programů kód se standardním ošetřením softwarových výjimek, nejčastěji vedoucí k ukončení programu, některé programovací jazyky však programátorovi dovolují reagovat na výjimky vlastním kódem. V následujících konstrukcích se provádějí příkazy v části try, a nastane-li chyba (zachytí-li se výjimka), přeskočí se na příkazy v části except, resp. catch:

```
(Free Pascal)   try commandsTry except commandsExcept end;          // konstrukce může být složitější
(Octave)        try, commandsTry, catch, commandsCatch, end
(Python)        try: || __cmdsTry || except exception: || __cmdsExc ... || else: || __cmdsEls || finally: || __cmdsFin
```

Poté program pokračuje sekvenčně příkazem za konstrukcí try.

Př. Dělení nulou. Dělení reálné nenuly nulou vrátí jako výsledek znaménkové nekonečno (±Infinity, ±Inf) a dělení nuly nulou nečíslo (Not-a-Number, NaN); obojí patří mezi hardwarové výjimky zachycované procesorem (floating-point exceptions), které mohou a nemusí vyvolat softwarovou výjimku. V Octavu hardwarové výjimky nevadí a program pokračuje bez softwarové výjimky, v Pascalu a Pythonu se softwarová výjimka vyvolá:

```
(Free Pascal)   for i:=-1 to 1 do writeln(1/i:4:1);       // výpis -1.0 a chyba Runtime error 200 Division by zero
(Octave)        for i=-1:1, 1/i, 0/i, end                 // výpis -1 -0, Inf NaN, 1 0
(Python)        for i in (-1,0,1): print(1./i)            // výpis -1.0 a chyba ZeroDivisionError: float division by zero
```

Nevyhovuje-li standardní ošetření softwarové výjimky, může programátor připravit vlastní postup pomocí try. V Octave jsou hardwarové výjimky neškodné a větev catch je nečinná, v Pascalu (z příkazového řádku, Lazarus situaci komplikuje) a Pythonu větev except už v akci uvidíme:

```
(Free Pascal)   for i:=-1 to 1 do try writeln(1/i:4:1) except writeln('error') end;  // výpis -1.0, error, 1.0
(Octave)        for i=-1:1, try, 1/i, 0/i, catch, 'error', end, end                  // výpis -1 -0, Inf NaN, 1 0
(Python)        for i in (-1,0,1):                                                   // výpis -1.0, error, 1.0
                  try: print(1./i)
                  except: print('error')
```

Příkazy pro vstup a výstup dat

Vstupní data lze do programu vpravit pomocí přiřazovacích příkazů ve zdrojovém textu nebo pomocí příkazů pro vstup dat z klávesnice nebo z diskového souboru. Výstupní data mohou pomocí příkazů pro výstup mířit na obrazovku nebo do diskového souboru. Zde se soustředíme na vstup z klávesnice a výstup na obrazovku, ke čtení a zápisu souborů se vrátíme později. Frekventovanými slovy programovacích jazyků pro příkaz vstupu jsou read, scan a input, pro výstup se užívá print, write, disp aj. Někdy (Pascal, C) je vstup a výstup formálně chápán jako příkaz volání příslušné standardní procedury či funkce.

Syntaxe v Pascalu – vstup: read(data), readln(data), výstup: write(data), writeln(data). Varianty s koncovkou -ln provedou odřádkování (čteme read-line, write-line).

V C se nabízí dvojice scanf a printf, ve Fortranu read a print/write, v Octave input a disp a v Pythonu input a print. V interaktivním režimu Octave a Pythonu je běžné vypsat hodnotu výrazu pouhým jeho zápisem.

Př. Vstup a kontrolní výpis dat. Na výzvu odpovíme zadáním čísla z klávesnice a údaj vypíšeme s implicitním i explicitním formátováním. Pascal může explicitně formátovat jednoduchým uvedením délky výpisu za výraz, např. writeln(1:2,3.:4:1) pro 2znakový výpis celočíselné 1 a 4znakový výpis reálné 3. s 1 znakem za desetinnou tečkou (tj. _1_3.0). V jazycích dominuje formátování inspirované funkcí printf jazyka C. Ta obsahuje formátové specifikace, pomocí nichž se formátují výstupy celočíselné (%d, %nd), reálné (%f, %nf, %n.nf, %n.ne), znakové (%s, %ns) ad. Free Pascal poskytuje v knihovně sysutils funkci format, jejímž prvním argumentem je řetězec s obdobnými formátovými specifikacemi. Fortran používá specifikace I pro integer, F a E pro real, L pro logical a A pro znaky.

```
(Pascal)    write('Zadej n: ');     readln(n);      writeln('n = ',n);      writeln(format('%s%d',['n = ',n]));
(C)         printf("Zadej n: ");    scanf("%d",&n);                         printf("%s%d\n","n = ",n);
(C++)       cout << "Zadej n: ";    cin >> n;       cout << "n = " << n;
(Fortran)   print *,'Zadej n: ';    read *,n;       print *,'n = ',n;       print '(A,I0)','n = ',n
(Octave)    n=input('Zadej n: ');                   n,                      printf('%s%d\n','n = ',n))
(Python)    n=input('Zadej n: ');                   print('n = ',n);       print('%s%d'%('n = ',n))
```

Př. Pozastavení programu. Volání readln bez argumentů slouží k pozastavení programu. Pokračuje se stisknutím klávesy Enter: program prg; begin write('Press Enter to continue . . . '); readln; end.

Komentáře

Pestře volené jsou symboly pro vyznačování komentářů ve zdrojových textech různých programovacích jazyků. Obecně může programátor použít řádkový komentář, platný od příslušného symbolu do konce řádku, nebo blokový komentář, platný od příslušného symbolu do jeho párového protějšku. Řádkové komentáře v Pascalu jsme výše použili, jsou uvozeny dvojitým lomítkem //. Totéž platí v C99, ve Fortranu uvozuje ! (a kdysi C v prvním sloupci), v Octave %, v Pythonu #. Blokové komentáře: (* Pascal *), { Pascal }, /* C */, %{ Octave %}, ''' Python '''.

STANDARDNÍ DATOVÉ TYPY

Datový typ charakterizuje vlastnosti proměnných, konstant a dalších veličin. Několik datových typů reprezentujících celá čísla a reálná čísla je standardizováno a v soudobém hardwaru zakódováno, a je proto standardně zpřístupněno v programovacích jazycích. Pro pohodlí programátora jazyky poskytují i různá, rovněž standardní, rozšíření nebo omezení těchto typů (komplexní, logické a znakové typy).

Celočíselné typy (integer)

Jsou vhodné pro reprezentaci matematicky celočíselných hodnot a zpravidla nutné pro indexy cyklů a prvků polí. V paměti mívají vyčleněnou velikost obvykle 4 byty (32 bitů) a dovolují tak udržet celkem 2^{32} různých celočíselných hodnot; jazyky běžně nabízejí i jiné velikosti (1, 2 a 8 B). Bývá volitelné, zda rozsah hodnot pokrývá přirozená čísla (neznaménkový integer) nebo čísla po obou stranách nuly (znaménkový integer). Velikost v paměti implikuje rozsah dostupných hodnot v desítkové soustavě: 1bytový (8bitový) znaménkový typ zahrnuje 2^8 = 256 různých hodnot -2^7..2^7-1 čili -128..127, 2bytový typ snese -2^{15}..+2^{15}-1 čili -32768..32767 a 4bytový typ -2^{31}..+2^{31}-1 čili něco přes ±2 miliardy.

Názvy celočíselných typů. Pascal rozumí znaménkovým typům: shortint (1 B), smallint (2 B), integer (implicitní volba, tzv. default, 4 B, totéž co longint) a int64 (8 B), i neznaménkovým typům: byte (1 B), word (2 B), longword (4 B). Celočíselné typy v C se deklarují pomocí slov [signed] char, [signed] short [int], [signed] int, [signed] long [int], unsigned [int] aj., znaménkové typy ve Fortranu pomocí integer, integer(4), integer(8) aj. Octave preferuje 8bytový reálný typ a s nadhledem dovoluje jeho použití i při indexování: příkaz n=2 vytvoří a inicializuje reálnou proměnnou a zápisy prvku pole a(2), a(2.), a(n) jsou pak všechny rovnocenné; k mání jsou ovšem i „čisté" typy int32, int64, uint32 ad. Python má znaménkový typ int (4 nebo 8 B) a navíc paměťově neomezený typ long s celočíselnou aritmetikou libovolné přesnosti; např. pro 2^{1000} lze prostě napsat 2**1000.

Celočíselné literály. 0, 1, -2, +3 apod. Default velikost bývá 4 B, v 64bitových systémech často 8 B. C vyznačí long literál jako 1L, Fortran svůj integer(8) jako 1_8, Octave nemá int literály, jen konverzní funkce, int32(1), a v Pythonu je long jednička 1L. Jazyky umožňují zápis literálů i v šestnáctkové a dalších soustavách:

(Pascal) write(15,%1111,&17,$f); // výpis: 4x 15
(C) printf("%d %d %d %d",15,0b1111,017,0xf); (Fortran) print *,15,b'1111',o'17',z'f' // boz literály
(Python) 15,0b1111,0o17,0xf, int('15',10),int('1111',2),int('17',8),int('f',16) // literály a konverzní funkce

Integer přetečení. Výsledek mimo pokryté celočíselné hodnoty běžícímu programu často nevadí (nevyvolá výjimku), i když je chybný. Následující ukázky se pohybují na horní hranici typu, tedy 2^{31}-1, resp. 2^{63}-1.

(Pascal) var n : integer; n:=high(integer); write(n,n+1) // 2147483647 a -2147483648 čili low(integer)
(C s limits.h) int n; n=INT_MAX; printf("%d %d",n,n+1); // výpis jako výše
(Fortran) integer n; n=huge(0); print *,n,n+1 // výpis jako výše
(Octave) 2**31, int32(2**31), int32(2**1000) // vypíše reálné číslo a 2krát liché číslo 2^{31}-1
(Python 64bit) n=2**62; n-1+n, n+n-1, n+n // vypíše 1krát int a 2krát long (s příponou L)

Uzavřenost. Jazyky poskytují celočíselné dělení (podíl po odřezání neceločíselného zbytku) a celočíselná aritmetika je tak uzavřená, výsledky operací s celočíselnými operandy jsou celočíselné.

Interní reprezentace. Uložení celých čísel v paměti připomíná zápis v dvojkové soustavě, např. 0101 = 2^2+2^0 = 5. Bity f_k o hodnotách 0 nebo 1 se pro p-bitový neznaménkový integer interpretují podle vzorce $\sum_{k=1}^{p} f_k \cdot 2^{k-1}$ a pro p-bitový znaménkový integer při $f_p = 0$ jako nezáporná čísla $\sum_{k=1}^{p-1} f_k \cdot 2^{k-1}$ a při $f_p = 1$ jako záporná čísla $-2^{p-1} + \sum_{k=1}^{p-1} f_k \cdot 2^{k-1}$. V příkladu výše jsou číslice f_k ukládány po řadě zprava doleva.

Reálné typy (real, floating point)

Jsou vhodné k aproximaci reálných čísel. Jejich podobu ukotvila norma IEEE 754, v procesorech respektovaná. Popisuje čísla v semilogaritmickém tvaru o dvojkovém základu $(-1)^s \cdot M \cdot 2^E$, kde s je znaménkový bit o hodnotách 0 nebo 1, mantisa $M = 1 + \sum_{k=1}^{p-1} f_k \cdot 2^{-k}$ a E je celočíselný exponent v mezích $\langle E_{\min}, E_{\max} \rangle$; bity f_k nabývají hodnot 0 nebo 1, f_0 je vždy 1 a je $1 \leq M < 2$. Reprezentovaná čísla mají omezenou přesnost mantisy a omezený rozsah exponentu, zachycují tedy jen některá reálná, přesněji racionální čísla, a někdy se jim raději než reálná říká „floating point", čísla s plovoucí (binární) tečkou. Norma definuje několik formátů (přesností), daných celkovým počtem bitů čísla, počtem p bitů mantisy a rozsahem exponentů. Procesory nejčastěji implementují přesnost jednoduchou (4bytovou) a dvojitou (8bytovou) a zatím zřídka čtyřnásobnou (16bytovou), někdy je dostupná i dvojitá rozšířená (10bytová) přesnost. Stojí za zvýraznění, že v reálných typech lze ukládat přesně zlomky (a součty zlomků) s mocninami dvou ve jmenovateli, např. 1/2, 3/4, nikoliv však např. 1/3 nebo 1/10.

Názvy reálných typů. V Pascalu je zvykem mít typy single (4 B) a double (8 B) pro jednoduchou a dvojitou přesnost a extended (10 B) pro rozšířenou přesnost; obecný real se ztotožňuje s 8bytovým double. C zavádí typy float, double a long double pro 4B, 8B a 16B přesnost, Fortran real(4), real(8) a real(16) pro totéž. Octave má typy single a double a Python jen float pro 8bytovou přesnost.
Reálné literály. Obsahují desetinnou tečku nebo v semilogaritmickém tvaru o desítkovém základu symbol exponentu: 0., 1.1, -0.2, 1e0, +1.e+0, 3.14e0, 6.67e-11 (pro $6.67 \cdot 10^{-11}$). V C, Pascalu, Octave a Pythonu je jejich default přesnost 8 B, ve Fortranu 4 B. V C mají float literály příponu f a long double L, Fortran používá podtržítka: 1._8, 1._16, Octave je bez single literálů, lze jen single(1).

```
(Pascal)    writeln(1.,12345678901234567890.0);  // 10bytové literály; pozor: lze psát 1., nelze (1.), lze (1.0)
(C)         printf("%f %f %f %f %Lf %Lf",1.,1e0, 1.f,1e0f, 1.L,1e0L); // 2x 8bytová, 4bytová a 16bytová 1.
(Fortran)   print *,1.,1._4,1e0,1e0_4,real(1,4), 1d0,1._8,1e0_8,real(1,8)    // 5x 4bytová 1., 4x 8bytová 1.
(Octave)    1,1.,1e0,double(1), single(1)              // 4x 8bytová 1., 1x 4bytová 1.
(Python)    1.; 1e0; float(1)                          // 3x 8bytová 1.
```

Meze reálných typů. Přesnost mantisy a rozsah exponentu jsou definovány v dvojkové soustavě, člověk je pozoruje v desítkové soustavě:

	celkem		mantisa			exponent	
precision	bytů	bitů	p bitů	relativní přesnost	platných míst	dvojkově	desítkově
single	4	32	24	$2^{-23} \approx 1.2 \cdot 10^{-7}$	7,22	-126..127	-45..38
double	8	64	53	$2^{-52} \approx 2 \cdot 10^{-16}$	15,95	-1022..1023	-324..308
double extended	10	80	64	$2^{-63} \approx 1 \cdot 10^{-19}$	19,27	-16382…16383	-4951..4932
quadruple	16	128	113	$2^{-112} \approx 2 \cdot 10^{-34}$	34,02	-16382…16383	-4951..4932

Relativní přesnost (též počítačové epsilon) chápeme jako nejmenší číslo, jehož přičtením či odečtením lze změnit jedničku, a při p-bitové mantise se rovná 2^{1-p}. Počtem platných míst v desítkové soustavě pak myslíme hodnotu $P = p \log_{10} 2$ plynoucí ze vztahu $10^{-P} = 2^{-p}$. Největší single hodnota je $\sum_{k=0}^{23} (\frac{1}{2})^k \cdot 2^{127}$, přibližně 3.4e38, a pro double $\sum_{k=0}^{52} (\frac{1}{2})^k \cdot 2^{1023}$ ~ 1.8e308, nejmenší řádnou kladnou hodnotou je single 2^{-126} ~ 1.2e-38 a double 2^{-1022} ~ 2.2e-308. Lze pracovat i s čísly s menší než řádnou kladnou hodnotou, tj. s $f_0 = 0$. Nejmenším kladným z nich je v single přesnosti $(\frac{1}{2})^{23} \cdot 2^{-126}$ ~ 1.4e-45, v double $(\frac{1}{2})^{52} \cdot 2^{-1022}$ ~ 4.9e-324.
Běžně se počítá v 8bytové přesnosti s 16místnou mantisou, přičemž procesory mohou interně vyčíslovat ve vyšší přesnosti (proto má mantisa 10bytového typu právě 64 bitů). 16bytová přesnost s 34místnou mantisou se užívá jen výjimečně, bývá softwarově emulovaná a velmi pomalá, pokud je vůbec překladačem nabízena. S rozmyslem se musí zacházet se 4bytovou přesností o 7místné mantise, která může snadno narazit: tři pokusy o výpis 8ciferné hodnoty, (C) printf("%f\n",16777217.f); (Fortran) print *,16777217., (Octave) single(16777217), vedou k výpisu hodnoty 16777216.0. To také dokumentuje, že existují 4bytová integer čísla, která nelze přenést do 4bytového reálného typu přesně (sekvence x=n; n=x může při single x a integer n změnit hodnotu n); 8bytový reálný typ už pojme všechna 4bytová celá čísla bez ztráty platných míst.
Přetečení, podtečení a uzavřenost. Rozsah exponentů uvedený v tabulce (vždy mocnina 2 minus 2) napovídá, že 2 hodnoty exponentu jsou rezervovány pro speciální hodnoty. Patří k nim dvě znaménková nekonečna (+Inf, –Inf) pro čísla s příliš velkou absolutní hodnotou, dvě znaménkové nuly, subnormální čísla neboli nenulová čísla s $f_0 = 0$ a tzv. nečísla (NaN, Not-a-Number), kterými se nahrazují matematicky nedefinované výsledky, např. 0./0., sqrt(-1.) apod. Výsledky operací s reálnými operandy tak mohou zůstat v oboru reálného typu, reálná aritmetika je uzavřená. Na závěr paradox: „celých čísel je více než reálných čísel", přesněji celočíselný datový typ reprezentuje více různých číselných hodnot než reálný typ o témže počtu bytů, už jen kvůli existenci dvojice reálných nul, hlavně však nečísel.

Komplexní typy (complex)

Jsou vhodné k aproximaci komplexních čísel. Interně jsou reprezentovány dvěma čísly reálného typu pro reálnou a imaginární část. Ve standardním Pascalu komplexní typ není, ve Free Pascalu ho přináší modul ucomplex. C získalo komplexní aritmetiku normou C99, Fortran ji má od kolébky, Octave a Python také.
Př. Odmocnina z -1. Jazyky preferují udržování výsledků matematických funkcí s reálnými argumenty v reálném oboru; výpočet $\sqrt{-1}$ voláním funkce sqrt(-1) tak zpravidla vede k NaN (Octave je výjimkou). Pro počty v komplexním oboru bývá třeba používat funkce se specifickým názvem.

```
(Pascal)        uses ucomplex; var c : complex; c:=cinit(-1,0); c:=csqrt(c); write(cstr(c),' ** 2 = ',cstr(c**2,6,2));
(C s complex.h) double complex c; c=-1+0*I; c=-1; c=csqrt(c); printf("I*I = %f+%fI",creal(c*c),cimag(c*c));
(Fortran)       complex c; c=(-1,0); c=cmplx(-1,0); c=-1; c=sqrt(c); print *,(0,1),' ** 2 = ',c**2
(Octave)        c=-1+0i; c=complex(-1); c=-1; c=sqrt(c); [num2str(1i) ' ^ 2 = ' num2str(c^2)]
(Python)        import cmath; c=-1+0j; c=complex(-1); c=-1; c=cmath.sqrt(c); print(1j,' ** 2 = ',c**2)
```

Logické typy (boolean, logical)

Jsou vhodné pro reprezentaci logických hodnot pravda/nepravda. Ty se nejčastěji objevují při rozhodování v podmíněných příkazech jako výsledky relačních výrazů. V Pascalu se logický typ nazývá boolean (literály: true, false), v C je zvykem reprezentovat ho číselnými hodnotami (0 pro false, nenula pro true), od C99 lze užít typ bool (literály: true, false), Fortran má typ logical (.true., .false.), Octave logical (true, false) a Python bool (True, False). Octave a Python rozumí po vzoru C i číselným hodnotám.

Př. Větvení podle podmínky. Opakujeme podmíněný příkaz if, zde s jedinou větví, a podmíněný výraz, pokud v jazyce existuje.

```
(Pascal)        var b : boolean; b:=true; if b then write(b);                                  // výpis TRUE
(C)             int b; b=1; if (b) printf("%d ",b); printf("%s",b ? "true" : "false");         // výpis 1 true
(C s stdbool.h) bool b; b=true; if (b) printf("%d ",b); printf("%s",b ? "true" : "false");     // výpis 1 true
(Fortran)       logical b; b=.true.; if (b) print *,b; print *,merge('true ','false',b)        // výpis T true
(Octave)        b=true; if b, b, end; b=1; if b, b, end                                        // výpis 1 1
(Python)        b=True || if b: print(b) || b=1 || print('true' if b else 'false')             // výpis True true
```

Znakové typy (char, character)

Jsou určeny pro reprezentaci znaků, v programovacích jazycích obvykle 1bytových (v kódování podle tabulky ASCII), občas i 2bytových (kódování UTF-8, Unicode). 1bytové kódování umožňuje reprezentovat 256 znaků, z nichž pevně daná první polovina (tzv. 7bitové ASCII kódy 0..127) zahrnuje 10 číslic, 2krát 26 znaků pro velká a malá písmena bez diakritiky a několik desítek dalších symbolů (mezera, !, ", # atd., nikoliv však např. ≠, ≤, ≥); druhá polovina tabulky se znaky národních abeced je přepínatelná. Dá se zapamatovat několik ASCII kódů a v příhodné okamžiky jich využít: souvisle řazené číslice začínají na kódu 32*3/2 (např. jednička na 48+1), velká písmena na 32*2+1 a malá písmena na 32*3+1. Ve Windows přežívá starodávný postup pro zadávání znaků pomocí jejich ASCII kódů: podržet klávesu Alt, na numerické klávesnici (klávesy vpravo, ne nahoře) zadat kód a uvolnit Alt (např. Alt+64 pro @, Alt+92 pro \).

Znakový typ v Pascalu je char, v C rovněž, ale jde o 1bytový celočíselný typ (pro ASCII kód znaku), Fortran má character, Octave char a Python str. Znakové literály se uzavírají mezi apostrofy, př. 'a', někdy jsou povoleny i uvozovky, "a"; literál obsahující apostrof se píše někde zdvojeně, '''', jinde pomocí tzv. escape sekvence, '\''. Ke znakovým literálům lze řadit i prázdný řetězec ''. Jméno char či chr typicky nese i konverzní funkce ASCII kódu na příslušný znak.

Př. ASCII tabulka. Třířádkový výpis klíčové části tabulky se znaky s kódy 33-64, 65-96, 97-128 (128 je bonus).

```
                !"#$%&'()*+,-./0123456789:;<=>?@   ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`   abcdefghijklmnopqrstuvwxyz{|}~⌂Ç
(Pascal)        var r,c : integer; for r:=1 to 3 do begin for c:=1 to 32 do write(char(r*32+c)); writeln end;
(C)             int r,c; for (r=1; r<=3; r++) { for (c=1; c<=32; c++) { printf("%c",r*32+c); } printf("\n"); }
(Fortran)       integer r,c; do r=1,3; do c=1,32; print '(a$)',char(r*32+c); enddo; print *; enddo
(Octave)        for row=1:3, disp([char((1:32)+row*32)]); end
(Python)        for row in range(1,4): || __for col in range(1,33): || ____print(chr(row*32+col)), || __print
```

VÝRAZY

Výraz je předpis pro získání hodnoty vyhodnocením operací kombinujících operandy (proměnné, konstanty, návratové hodnoty funkcí). Výraz lze použít na pravé straně přiřazovacího příkazu, jako argument při volání procedury, jako součást jiného výrazu a na dalších místech. Aritmetické výrazy se skládají z matematických operací na numerických operandech, v relačních výrazech jsou operandy porovnávány, v logických výrazech se operandy kombinují podle pravidel Booleovy algebry. Pro vyhodnocování složitějších výrazů jsou stanoveny priority operátorů určující pořadí, v jakém se vyhodnocují dílčí výrazy. Není-li v jazyce k dispozici jinak běžný operátor, je patrně realizován funkcí.

Aritmetické výrazy

Elementární operace. Jejich operandy i výsledná hodnota jsou celočíselného, reálného nebo komplexního typu. Zahrnují sčítání +, odčítání –, násobení *, dvě dělení a někdy umocňování. Datový typ výsledku se snaží udržet datový typ operandů: jsou-li operandy celočíselné, výsledek je celočíselný (s možnou výjimkou dělení), jsou-li operandy reálné, výsledek je reálný (nekonečna a nečísla patří mezi reálná čísla) a podobně pro operandy komplexní.
Př. 1+1 je celočíselná 2, 1.0+1.0 je reálná 2.0, fortranský výraz (1.,0.)+(1.,0.) se vyhodnotí jako komplexní (2.,0.).
Typová konverze. Jsou-li operandy různého typu, provede se nejprve typová konverze, konkrétněji povýšení typu operandu o menším typu, přičemž celočíselné typy se chápou jako menší než reálné a ty jako menší než komplexní. Výsledek je pak téhož typu jako oba operandy po typové konverzi, s výjimkou dělení. Konverze při umocňování jsou atypické, umocňování na (malý) celočíselný exponent je jistě vhodnější realizovat násobením než logaritmováním.
Př. 1+1.0 se stejně jako 1.0+1 povýší na 1.0+1.0.
Komutativita a asociativita. Počítačové operace + a * jsou jako v matematice komutativní, platí: 0+1 je totéž co 1+0, 0.*1 je 1*0., high(integer)+1 je 1+high(integer) (tj. low(integer) nebo přetečení nebo výsledek povýšený do 8bytového typu) a 1e200*1e300 je 1e300*1e200 (tj. +Inf). Operace + a * však někdy nejsou asociativní, a to jako důsledek omezeného rozsahu a omezené přesnosti reálných typů: při vyhodnocování v 8bytovém typu (1e300*1e300)*1e-300 je +Inf, zatímco 1e300*(1e300*1e-300) je 1e300, a (-1.0+1.0)+1e-16 je 1e-16, zatímco -1.0+(1.0+1e-16) je 0.0.
Dělení. V jazycích bývá dvojí dělení, celočíselné s celočíselnými operandy i výsledkem (tj. s celou částí matematického výsledku) a reálné (resp. komplexní) s reálným výsledkem. Některé jazyky (C, Fortran) mají jeden operátor / a rozhodují se kontextově podle typu operandů (jsou-li oba operandy celočíselné, jde o dělení celočíselné, jinak reálné), jiné jazyky mají různé operandy pro různá dělení: např. (Pascal) div pro dělení celočíselné a / pro reálné. Některé jazyky šly oběma cestami: v Pythonu 2 operátor / dělí podle typu operandů, v Pythonu 3 je to reálné dělení. K celočíselnému dělení pak patří i operátor pro zbytek po dělení čili modulo, v Pascalu mod, jinak (C, Python) často %, případně (Fortran) funkce mod.
Př. C, Fortran, Python 2: 1/3 je integer 0; 1./3, 1/3. a 1./3. jsou real 0.333…; Pascal, Python 3: 1/3 i 1./3. je 0.333…. Pascal: 1 div 3 je 0 a 1 mod 3 je 1. Fortran: mod(1,3).
++ a --. Oblíbené jsou unární operátory inkrementace ++ a dekrementace --: (C, Octave) výrazy ++i a i++ zvyšují hodnotu proměnné o 1 a --i a i-- hodnotu proměnné o 1 snižují.
Umocňování. Pro umocňování se někdy (Fortran, Octave, Python) používá operátor **, jindy (Matlab, Octave) ^, s knihovnou math umí ** i Free Pascal.
Př. 2**2 je celočíselná 4 a 2.0**2 je reálná 4.0, ale spočtená patrně úsporněji než 2.0**2.0.
Rychlost. Elementární operace +, - a * mají vysokou rychlost, odpovídá frekvenci procesoru udávané v GHz, za sekundu se jich tedy stihne řádově miliarda. Dělení bývá několikanásobně pomalejší. Umocnění (až na situaci s malým celočíselným exponentem) sdílí náročnost výpočtu s logaritmem a exponencielou, je řádově pomalejší než elementární operace.
Priority. Různé operace mohou mít rozdílnou prioritu: umocnění má obvykle nejvyšší prioritu, pak přijdou na řadu multiplikativní operace (násobení a dělení) a nakonec aditivní operace (sčítání a odčítání). Pro více operací stejné priority může být stanoveno pořadí vyhodnocování, obvykle zleva doprava, s možnou výjimkou pro umocňování. Požadované pořadí vyhodnocování lze zvýraznit závorkami.
Př. 1+2*3 je totéž co 1+(2*3), tedy 7. Výraz 2**3**2 je někdy 64 (Pascal, Octave), jindy 512 (Fortran, Python).
Strašidelné příklady uplatnění priorit s kontextovým lomítkem: 1/3+1/3+1/3 je 0, 1./3+1/3+1/3 je 0.333…, až 1./3+1./3+1./3 je 1.0. Podobně: 1/4*4 je 0, 1/4*4. je 0.0, až 1./4*4 je 1.0. Dělení může být skryto v umocňování se záporným exponentem: 10**(-1) se může chápat jako 1/(10**1), tedy 0 (Pascal, Fortran), ale může být i 0.1 (Octave, Python).
Poznamenejme, že + a - jsou i unární operátory, např. -(1), +x. (Minus v -1 je součást literálu, nikoliv operátor.)

Řetězcové výrazy

Práce s daty znakových typů se obvykle koná pomocí funkcí a procedur. K dispozici bývá jen operátor pro řetězení ("sčítání") znaků a řetězců: (Pascal, Python) + či (Fortran) //.
Př. 'A'+'BC' pro 'ABC'. V Octavu je 'A'+'B' rovno 131 (součet ASCII kódů 65+66).

Relační výrazy

Relace slouží k porovnání operandů kompatibilních typů. Vždy lze porovnávat na rovnost a nerovnost, pro operandy datových typů, pro které je definováno pořadí, lze porovnávat i velikost. (Standardní datové typy, s obvyklou výjimkou pro typ complex, mají definováno pořadí.) Výsledkem bývá hodnota logického typu, v některých jazycích celočíselná hodnota s logickou interpretovatelností.
Operátory. Pro test rovnosti se užívá někdy = (Pascal), jindy dvojice == (C, Fortran, Octave, Python), operátor pro nerovnost je ještě mnohotvárnější: <> (Pascal), != (C, Octave, Python), /= (Fortran). Operátory pro porovnání velikosti jsou jednoznačné: >, >=, <, <=. Fortran má i ekvivalentní zkratky: .eq., .ne., .gt., .ge., .lt., .le..
Př. (Pascal) 1=1, 1<>2, 1<2, 1.<=2., 1<2., false<true, '1'<'2' pro 7krát true.
Test rovnosti v reálné aritmetice. Zvýrazněme nevhodnost až nebezpečnost zjišťování přesné rovnosti dvou reálných operandů, jsou-li výsledkem nepřesných operací reálné aritmetiky. Vhodné je stanovit si potřebnou přesnost a porovnávat absolutní hodnotu rozdílu reálných operandů s touto přesností.
Př. 0.1+0.1==0.2 vrací true, zatímco 0.1+0.1+0.1==0.3 vrací false (pro 8bytový real). Robustní modifikace: abs(0.1+0.1-0.2)<1e-15 a abs(0.1+0.1+0.1-0.3)<1e-15 pro 2krát true.

Logické výrazy

Operandy logického typu lze skládat podle pravidel Booleovy algebry. K dispozici bývají operátory pro negaci (not, !), konjunkci neboli logický součin (and, &&), disjunkci neboli logický součet (or, ||) a exkluzivní disjunkci (xor).

operandy...		výsledky operací...				
a	b	not a	a and b	a or b	a xor b	not a or b (=>)
false	false	true	false	false	false	true
false	true	true	false	true	true	true
true	false	false	false	true	true	false
true	true	false	true	true	false	true

Bývá možné nastavit zkrácené vyhodnocování logických výrazů: ve výrazech false and X a true or X není třeba pro zjištění výsledku vyhodnocovat operand X. To však může v některých situacích způsobit odlišné chování programu než při úplném vyhodnocení všech operandů.

Priority

Pravidla pro pořadí vyhodnocování aritmetických výrazů jsme již uvedli. Po vyhodnocení dílčích aritmetických výrazů se obvykle vyhodnocují relace a ty se nakonec poskládají pomocí logických operátorů. Pascal má podivnou výjimku: relace se vyhodnocují jako poslední. Každopádně, když standardní priority nevyhovují nebo nejsou zřejmé, lze závorkovat.
Př. (C, Octave) 1<2 && 3<4, (Fortran) 1<2 .and. 3<4, (Python) 1<2 and 3<4, ale (Pascal) (1<2) and (3<4).

A jiné

Jazyky mívají bitové operátory, operující na integer operandech bit po bitu podle pravidel Booleovy algebry. Je-li v jazyce zaveden datový typ množina, jsou tam i množinové operátory (průnik, sjednocení ad.). Při práci s ukazateli bývá potřeba unární referenční a dereferenční operátor (Pascal: @ a ^, C: unární * a &). Oblíbený bývá podmíněný výraz (jediný ternární, se třemi operandy) vracející podle logické interpretace jednoho operandu hodnotu druhého nebo třetího operandu: (C) 1 ? "ano" : "ne", (Python) 'ano' if 1 else 'ne' pro 'ano'. Vzácností je operátor pro faktoriál: (gnuplot) 5!.

JEDNODUCHÉ DATOVÉ TYPY

Vedle standardních datových typů (Pascal: integer, real, boolean, char) jsou k dispozici ještě další jednoduché datové typy a také strukturované typy složené z veličin již známých typů. K jednoduchým typům se řadí výčty (dostupné i v C a Fortranu) a intervaly, strukturovanými typy jsou především pole složená z prvků téhož typu, záznamy/struktury složené z položek obecně různých typů a množiny. Pole a záznamy (i výčty) se naleznou i v C a Fortranu a zdatnost v zacházení s nimi je nevyhnutelná (pole jsou běžná např. k popisu fyzikálních polí, zobecněné záznamy jsou klíčovým typem objektově orientovaného programování). Strukturovaným typem jsou rovněž řetězce neboli pole znaků. Pascal a C považují za strukturovaný typ i soubory, ve Fortranu se při práci se soubory užívá jiná terminologie.

Ordinální typy

Termín používaný v Pascalu pro typy, jimiž reprezentované hodnoty lze (rozumně) očíslovat pořadovým, tj. ordinálním číslem; to tvoří vazbu ordinálního typu na typ integer. Jde tedy o celočíselné typy (Int64 s omezeními), boolean, char, výčty a intervaly, nejde o typy reálné. Funkce Ord(X) zjistí ordinální číslo výrazu X ordinálního typu, funkce Succ(X), resp. Pred(X) vracejí hodnotu následující po, resp. předcházející výrazu X a procedury Inc(X,n) a Dec(X,n) mění hodnotu proměnné X o n pozic. Proměnná ordinálního typu může indexovat cyklus i pole.

Př. Inc(X) je totéž co X:=Succ(X), Dec(X) totéž co X:=Pred(X).
Ord(0), Ord(false), Ord('0'), Ord('A'), Ord('a') vrací 0, 0, 48, 65, 97.

Intervaly

Interval (subrange) je ordinálním typem pro omezený počet po sobě jdoucích hodnot s předepsanou dolní a horní mezí téhož bázového (samozřejmě ordinálního) typu. Ordinální čísla intervalu jsou rovna ordinálním číslům hodnot bázového typu, dolní a horní mez vracejí funkce Low a High. Interval se může objevit v samostatném popisu typu (pojmenovaný typ) nebo rovnou v deklaraci proměnné (anonymní typ). Dva různé typy interval jsou kompatibilní, jsou-li odvozeny od téhož bázového typu, a kompatibilní jsou pak i příslušné proměnné. Typ interval je užíván v deklaracích statických polí pro rozsah jejich mezí a také pro automatickou kontrolu povoleného rozsahu hodnot prováděnou překladačem nebo za chodu programu (pokud je zapnuto Range checking).

Syntaxe v Pascalu

```
type Interval = LowerBound .. UpperBound;                                  // pojmenovaný typ
var v1 : Interval;
    v2 : lb2 .. u2;                 // jsou-li všechny meze kompatibilních typů, jsou kompatibilní i v1 a v2
```

Př.
```
type tDen=1..31; tMesic=1..12; tRok=1900..2099;
var den : tDen; mesic : tMesic; rok : tRok;
var ch1 : 'a'..'z'; ch2 : 'A'..'Z';
den:=1; mesic:=den; rok:=low(tRok)+110; writeln(den,mesic:2,ord(rok):5);     // vypíše 1 1 2010
ch1:=high(ch1); ch2:=low(ch2); if ch1=ch2 then writeln('To nejde.') else writeln(ch1,ch2);   // z A
```

Výčty

Výčet (enumeration) je ordinálním typem pro omezený počet (ne nutně navazujících) hodnot reprezentovaných symbolickými jmény. Legálními hodnotami daného výčtu jsou i přeskočená ordinální čísla. Není-li ordinální číslo přiřazeno explicitně, čísluje se od 0 nebo od předchozího ordinálního čísla. Symbolická jména výčtových hodnot, obdobně jako návratové hodnoty funkcí Succ, Pred, Low a High, nemohou být používána v číselných výrazech; takto zacházet však lze s příslušnými ordinálními čísly. Dva různé výčtové typy nejsou kompatibilní a kompatibilní nejsou ani příslušné proměnné. (Předchozí dvě věty platí jen v Pascalu, jehož pravidla pro kompatibilitu typů jsou extrémní; v C i Fortranu jsou jména výčtových hodnot celočíselnými symbolickými konstantami.) Podobně jako interval, výčet může být pojmenovaný nebo anonymní. Vhodné použití výčtového typu je tam, kde ordinální čísla přípustných hodnot mají pojmenovatelný význam (Leden=1, Unor=2 atd.) nebo kde na konkrétních hodnotách ordinálních čísel nezáleží (Club=1, Diamond=2, Heart=3, Spade=4).

Syntaxe v Pascalu

```
type tEnum1 = (val0, val1, val2, ..., valn);                    // ordinální čísla od 0
     tEnum2 = (val0=ord0, val1=ord1, ..., valn=ordn);           // přiřazená ordinální čísla
```

C: typedef enum {val0,...} tEnum; Fortran: enum,bind(c); enumerator :: val0,...; end enum.

Př.
```
type MyBoolean=(false,true);                                    // boolean konstanty tímto zastíněny
var b1 : boolean; b2 : MyBoolean;
b1:=boolean(true); b2:=true; writeln(b1,ord(b1),ord(b2));       // tzv. typecasting; vypíše se TRUE11
type tRoman=(I=1,V=5,X=10,L=50,C=100,D=500,M=1000);
writeln(ord(M)+ord(M)+ord(X),ord(high(tRoman))-ord(low(tRoman))+1);        // vypíše 2010 a 1000
```

Ukazatele

Ukazatel je proměnnou neobsahující datovou hodnotu, ale paměťovou adresu (na níž se očekává datová hodnota). V programovacích jazycích je užíván např. k dynamické alokaci paměti, tj. k alokování paměti za chodu programu pro ty proměnné, pro které nebyla paměť alokována staticky předem. Ukazatele mohou být deklarovány explicitně, často však i skrytě jako např. dynamická pole nebo řetězce. Ukazatele jsou také nezbytnou součástí tzv. dynamických datových struktur (spojový seznam, binární strom aj.) používaných v algoritmech běžných spíše v informatice než fyzice. Při práci s ukazateli je snadné dopouštět se nesnadno lokalizovatelných chyb s vážným dopadem na chod programu.

Syntaxe v Pascalu

– deklarace typu ukazatel na cílový typ: `type pTyp = ^Typ;` // stříška zleva pro deklaraci typu
– deklarace ukazatele: `var pointer : pTyp;`
– prázdný ukazatel pro nulování ukazatelů: `nil`
– adresní/referenční operátor pro adresu proměnné: `@variable`
– dereference ukazatele (cílová hodnota): `pointer^` // stříška zprava pro dereferenci
– dotaz na nenulovost ukazatele: `Assigned(pointer)` nebo `pointer<>nil`

Př. deklarace a inicializace ukazatele:

```
var p : ^integer = nil;
```

V C se místo ^ používá * a místo @ referenční operátor &; oblíbená je tzv. ukazatelová aritmetika pro pohyb v polích. Ve Fortranu jsou ukazatele realizovány odchylně (opatrněji), mj. dereference je automatická a programátor nemá přístup k adresám uloženým v ukazatelích.

Ukazatele a jejich cíle

Ukazatel na nepojmenovaný, dynamicky alokovaný cíl: paměť pro ukazatel (adresu) je připravena deklarací ukazatele, v pravou chvíli se voláním procedury New dynamicky alokuje paměť pro cíl ukazatele a do ukazatele se uloží adresa této paměti (ukazatel se přiřadí, asociuje), po použití se paměť dealokuje voláním procedury Dispose a přiřazením nil se ukazatel vynuluje. Přesměrování nebo přiřazení hodnoty nil do ukazatele s dynamicky alokovanou pamětí (nezpřístupněnou jiným ukazatelem) vede k tzv. úniku paměti, alokovanou paměť již v programu nelze uvolnit. To se nestane dynamickému poli, jemuž alokovaná paměť je automaticky dealokována nulováním (nil) dynamického pole nebo opuštěním jeho oblasti platnosti.

Pascal: alokace paměti a zamíření ukazatele `New(pointer);`
uvolnění paměti a nulování ukazatele `Dispose(pointer); pointer:=nil;`

Př.
```
var pInt : ^integer; New(pInt); pInt^:=1; writeln(pInt^); Dispose(pInt); pInt:=nil;
```

Ukazatel na pojmenovaný, existující cíl (alias, zkratka): ukazatel získá přiřazovacím příkazem adresu existující proměnné odpovídajícího typu nebo adresu uloženou v jiném ukazateli téhož typu, po použití je ukazatel přesměrován jinam nebo vynulován. Lze tak např. krátit cestu k součástem proměnných strukturovaných typů.

Pascal: zamíření na existující proměnnou `pointer := @promenna;`
zamíření na cíl jiného ukazatele `pointer := otherPointer;`
nulování ukazatele `pointer := nil;`

Př.
```
var i : integer=1; p,r : ^integer; p:=@i; r:=p; write(p^,r^); p^:=2; writeln(i:2,p^,r^); // vypíše 11 222
```

Problematické situace

neinicializovaný ukazatel (wild pointer) – nedefinovaný stav ukazatele

```
var p : ^real; if (Assigned(p)) ...                    // ukazatel může a nemusí být nil
```

únik paměti (memory leak) – ukazatel na alokovaný cíl přesměrován bez dealokace cíle

```
var p : ^real; New(p); p:=nil;                         // ztraceno 8 B
```

visící ukazatel (dangling pointer) – ukazatel míří na neexistující cíl

```
var p,r : ^real; New(p); r:=p; Dispose(p); p:=nil;     // Assigned(r) je true, cíl není
```

dvojí dealokace (double free) – dealokování už dealokované paměti

```
var p,r : ^real; New(p); r:=p; Dispose(r); Dispose(p); // runtime error
```

STRUKTUROVANÉ DATOVÉ TYPY

Typ pole

Pole (array) je strukturovanou proměnnou (nebo konstantou) složenou z prvků (elements) stejného bázového typu. Typu této proměnné se říká typ pole, pole je tedy proměnnou typu pole. (Říká se prostě pole a podle kontextu je to proměnná nebo typ.) Prvky pole jsou značeny celočíselnými (v Pascalu ordinálními) indexy. Jednorozměrná (1D) pole (vektory) jsou indexována jedním indexem, dvourozměrná (2D) pole (matice) dvěma indexy atd. V každém rozměru jsou meze indexů a tím i jejich počet (velikost rozměru) dány typem interval.
Potřebná velikost pole může být známa předem, v čase překladu, nebo vyplyne až za chodu programu. Polím deklarovaným včetně konstantních mezí se říká pole statická. Polím bez mezí uvedených v deklaraci se říká pole dynamická a místo v paměti se jim přiděluje za chodu programu dynamicky, i opakovaně. Při práci s dynamickými poli je v C nezbytné zacházet s ukazateli, Pascal i Fortran mají možnosti to zakrýt. C i Fortran mají pro zacházení s poli mocný aparát, každý jiný.

Syntaxe v Pascalu

– deklarace typu statického pole:

```
1D:   array [LowerBound..UpperBound] of BaseTyp;        // interval s konstantními mezemi
2D:   array [lb1..ub1,lb2..ub2] of BaseTyp;
      array [lb1..ub1] of array [lb2..ub2] of BaseTyp;  // alternativní deklarace matice
```

– deklarace typu dynamického pole:

```
1D:   array of BaseTyp;                                 // bez uvedení mezí
2D:   array of array of BaseTyp;
```

– alokace a dealokace paměti pro dynamické pole:

```
1D:   SetLength(pole,velikost); Finalize(pole);         // udává se velikost, nikoliv meze
2D:   SetLength(pole,velikost1,velikost2); Finalize(pole);  // Finalize(pole) a pole:=nil znamená totéž
```

dolní mez v každém rozměru dynamického pole je 0

– dotazovací funkce: funkce Low a High vracejí hodnoty dolní a horní meze prvního rozměru pole a funkce Length vrací velikost prvního rozměru; pro dynamické pole tedy vrací Low vždy 0 a High totéž co Length-1

– přístup k poli: jménem pole, přístup k prvku pole: jménem a indexy v [ ], indexy matice: [i1,i2] nebo [i1][i2]

– konstruktor pole: literál typu pole, seznam hodnot v ( ), v Pascalu jen v inicializačním výrazu statické deklarace

C: dolní mez 0, indexy v [ ], dynamická pole explicitními ukazateli, ukazatelová aritmetika

Fortran: meze libovolné (dolní implicitně 1), indexy v ( ), sekce pole, operátory a funkce zobecněné pro pole

Př. statické pole: deklarace typu, deklarace proměnné, inicializace

```
type tVector = array [0..10] of integer; tMatrix = array [-5..5,1..3] of real;
var v : tVector; m : tMatrix;
for i:=0 to 10 do v[i]:=i; for i:=low(m) to high(m) do for j:=low(m[0]) to high(m[0]) do m[i,j]:=i+j;
```

Př. dynamické pole: deklarace typu, deklarace proměnné, alokace paměti, inicializace

```
type tVectorD = array of integer; tMatrixD = array of tVectorD;
var v : tVectorD; m : tMatrixD;
SetLength(v,11); for i:=0 to length(v)-1 do v[i]:=i; Finalize(v);
SetLength(m,11,3); for i:=-5 to 5 do for j:=1 to 3 do m[i+5,j-1]:=i+j; Finalize(m);
```

Př. deklarace statického pole s inicializací konstruktorem pole

```
var v : array [1..3] of real = (1,2,3);             // inicializace konstruktorem pole
```

Poznámky

– Meze statických polí: Meze statických polí musí být literály nebo symbolické konstanty. Z historických důvodů není typová konstanta skutečnou konstantou, použity proto mohou být jen netypové konstanty.

– Pole ve výrazech a přiřazeních: V obecných výrazech mohou vystupovat jen prvky pole, nikoliv celá pole nebo jejich sekce. (Ve Fortranu mohou.) V přiřazeních mohou současně vystupovat celá pole, jen jsou-li identických typů, př. v1:=v2. Nelze v1:=skalar. (Ve Fortranu lze.)

– Vícerozměrná pole: Statická i dynamická pole mohou samozřejmě mít i více než 2 rozměry, př. t[i,j,k].

– Průchod vícerozměrným polem: Prvky pole jsou v paměti uloženy v souvislém prostoru, lineárně jeden za druhým. (Platí pro pole statická, nemusí platit pro vícerozměrná pole dynamická.) Prvky matice se ukládají po řádcích; je-li při průchodu matice možnost si vybrat, je efektivnější procházet ji po řádcích, tj. ve vnořených cyklech měnit první index nejpomaleji. (V C také, ve Fortranu naopak.) Příklad výše.

– Dynamická pole a ukazatele: Proměnná dynamického pole je (implicitním) ukazatelem. Zatímco přiřazení v1:=v2 pro statická pole vyjadřuje kopírování všech prvků pole, totéž pro dynamická pole je přesměrování proměnné (ukazatele) v2 na proměnnou v1. Budou tak pak v paměti existovat dvě různá statická pole, ale jen jedno dynamické (pod dvěma jmény). Přiřazení mezi prvky dynamických polí vyjadřují běžné kopírování.

Př.: (as, bs statická pole, ad, bd pole dynamická, v as, ad uloženy 1, v bs a bd 2)
as:=bs; bs[0]:=3; ad:=bd; bd[0]:=3; writeln(as[0],bs[0],' ',ad[0],bd[0]); // vypíše 23 33

– Neobdélníkové matice: dynamické matice mohou mít proměnnou délku řádku. K tomu je třeba použít postupu obvyklého v C, totiž alokovat nejprve vektor ukazatelů na řádky a pak ke každému prvku tohoto vektoru alokovat řádek samostatně.

Př.: (trojúhelníková matice) var tm : tMatrixD; SetLength(tm,11); for i:=0 to 10 do SetLength(tm[i],i+1);

– Ukazatelová aritmetika: Ukazuje-li ukazatel na prvek pole, lze jej přičtením či odečtením celého čísla posunout k sousedním prvkům. V C je příslušná syntaxe úsporná, v Pascalu nikoliv.

Př. (Pascal) var a : array of real; p : pReal; p:=@a; p:=pReal(pChar(p)+sizeof(real)); // jen pro char pointers
(C) float a[1],*pa; p=&a[0]; p++; // ++ posouvá o prvek

Typ řetězec

Řetězec (string) je strukturovaným typem pro práci se znaky. Řetězce lze chápat pole znaků (array of char), navíc je pro ně definována sada specifických funkcí a procedur a zobecněn operátor +.

Syntaxe v Pascalu

– deklarace řetězců s dynamickou a statickou délkou: var s : string; ss : string[MaxDelka];

– dynamická délka se nastavuje přiřazením výrazu nebo procedurou SetLength a zjišťuje se funkcí Length

Řetězec lze interpretovat jako skalár i jako pole znaků, i-tý znak lze psát jako s[i].

př. skalár: s:='abc'; s:=s+'def'; writeln(s);
př. pole znaků: s:='aBc'; s[2]:='b'; for i:=1 to Length(s) do write(s[i]); writeln;

Relace =, <>, >, >=, <, <= definovány na základě porovnávání ordinálních čísel jednotlivých znaků zleva; nejmenší řetězec je '' (prázdný řetězec)

Standardní procedury a funkce: Concat pro řetězení (totéž jako +), Copy pro získání podřetězce, Delete pro zrušení podřetězce, Insert pro vložení podřetězce, Pos pro nalezení podřetězce, Str pro konverzi čísla na řetězec a Val pro konverzi řetězce na číslo.

Speciální typy v Pascalu:

ShortString: statická (alokovaná) délka 255 nebo MaxDelka znaků, „dynamická" délka se mění podle okolností: s:='ahoj' má tak statickou velikost stále 256 B, zatímco funkce Length(s) vrátí hodnotu 4, uloženou v s[0]; Low(s) vrací 0 a High(s) statickou délku

AnsiString: nemá statickou velikost, alokovaná velikost se mění spolu s dynamickou délkou, kterou vrací rovněž funkce Length(s), nikoliv však už pouhý přístup k s[0]; Low(s) a High(s) nejsou použitelné

WideString: podpora 16bitových (2bytových) (tzv. Unicode) znaků

Typ záznam neboli struktura

Záznam (record) je strukturovanou proměnnou složenou z položek (fields) obecně různého typu. Typu této proměnné se říká typ záznam (Pascal), struktura (C) nebo odvozený typ (Fortran). Položky jsou pojmenovány, jejich jména mají význam jen v kontextu daného typu; stejná jména mohou být použita v jiném typu i úplně jinde.

Syntaxe v Pascalu

– deklarace typu: type tRecord = record field1 : Type1; field2 : Type2; ... end;
př. type tDatum = record den : tDen; mesic : tMesic; rok : tRok end;

– deklarace záznamu: var r : tRecord; datum : tDatum;

– použití záznamu (oddělovačem jmen záznamu a položky je tečka): r.Field1:=…; r.Field2:=…;
př. datum.den:=25; datum.mesic:=11; datum.rok:=2014;

– alternativa s příkazem with (není nutno opakovaně uvádět jméno záznamu):
with datum do begin den:=25; mesic:=11; rok:=2014 end;

Záznamy lze větvit podle položky ordinálního typu (deklarace variantní části syntakticky podobná příkazu case),
type tRecord = record f1 : T1; case tag: ordinalT of constantList1: (variant1); …; end;

Položky záznamu nemusejí být v paměti uloženy v souvislém prostoru (ale mohou, někdy to lze vynutit). Položky variantní části se v paměti mohou překrývat.

Kombinace polí a záznamů:

– pole záznamů: var PZ : array of record x : real end; SetLength(PZ,n); PZ[0].x:=…;

– záznam s poli: var ZP : record x : array of real end; SetLength(ZP.x,n); ZP.x[0]:=…;

Př. hmotný bod: var B : record m : real; x,v : array [1..3] of real end; with B do begin m:=1; x[1]:=0; ... end;

Objektově orientované programování

Zobecněním typu záznam o vnořené procedury (metody), pomocí kterých se pak přistupuje k položkám (vlastnostem) záznamu, vzniká základní typ objektově orientovaného programování: třída. Zobecněným záznamem je

pak objekt (instance objektu). Omezení přístupu k položkám objektů pouze prostřednictvím metod se říká zapouzdření. Dědičností tříd se míní zavádění tříd přejímáním vlastností a metod rodičovských tříd.

Typ množina

Množina (set) je strukturovaným typem realizujícím vlastnosti matematických množin. V C a Fortranu není.
Syntaxe v Pascalu
– deklarace množiny prvků bázového (ordinálního) typu: var s : set of BaseType;
– konstruktor množiny: [seznam prvků, použitelné i intervaly], prázdná množina: [ ]
Př. var Letters, ULetters, LLetters : set of char; … ; ULetters:=['A' .. 'Z']; LLetters:=['a'.. 'z'];
množinové operace: sjednocení +, průnik *, rozdíl –, relační operace, test existence prvku in
Př. Letters:=ULetters + LLetters; writeln('c' in Letters);

Typ soubor

soubor (File): textový (Text), s udaným typem, bez udaného typu (dána velikost bloku v bytech)
procedura Assign(var f:file; name:string) neboli AssignFile nahrazuje přiřazovací příkaz pro proměnné typu soubor
name 'Input', resp. 'Output' pro standardní vstup, resp. výstup
otevření existujícího souboru: procedury Reset (když Text, tak pouze pro čtení), Append (pouze Text a pro zápis)
otevření nového souboru (po předchozím smazání stejnojmenného): procedura Rewrite (Text pouze pro zápis)
uzavření otevřeného souboru: procedura Close
zápis: procedury Writeln pro Text, Write pro Text a typové soubory, BlockWrite pro beztypové soubory
čtení: procedury Readln pro Text, Read pro Text a typové soubory, BlockRead pro beztypové soubory
standardní přístup sekvenční, k netextovým souborům i přímý přístup pomocí procedury Seek
test dosažení konce souboru: funkce Eof(file)
dotaz po velikosti, resp. pozici v (ne Text) souboru: funkce FileSize(file), resp. FilePos(file)
ošetření chybových stavů: po direktivě {$I+} nebo {$IOCHECKS ON} ošetřuje (tvrdě) chybové stavy překladač,
po {$I–} se programátor může doptat u funkce IOResult a pak se rozhodnout samostatně
Př. Zápis do souboru

```
var soubor : text; ...                  // popis souborove promenne
assignFile(soubor,'soubor.dat');        // prirazeni jmena souboru promenne
rewrite(soubor);                        // otevreni pro zapis
writeln(soubor,1,true,'1');             // zapis do souboru
close(soubor);                          // zavreni souboru
```

Př. Čtení ze souboru

```
var soubor : text; line : string; ...   // popis souborove promenne
assignFile(soubor,'soubor.dat');        // prirazeni jmena souboru promenne
reset(soubor);                          // otevreni pro cteni
while not eof(soubor) do begin          // cteni radku do konce souboru
  readln(soubor,line);                  // precteni radku
  writeln(line);                        // vypis 1TRUE1
end;
close(soubor);                          // zavreni souboru
```

PROCEDURY A FUNKCE

Pro lidské oko je stravitelnější zdrojový kód členěný do menších úseků. Funkce se podobně jako v matematice používají pro získání jedné návratové hodnoty, procedury jsou určeny pro ostatní účely. V každém případě se chod programu po vykonání funkce a procedury vrací zpět k místu volání. Mnoho tzv. standardních (vnitřních) funkcí a procedur je už součástí programovacího jazyka. S volající programovou jednotkou procedura (vše platí i pro funkce) komunikuje jednak prostřednictvím svého rozhraní, tj. svých argumentů (parametrů), jednak prostřednictvím dat známých oběma jednotkám (globálních dat). Data deklarovaná v proceduře jsou vně procedury nedostupná (lokální data). Argumenty jsou vstupní, výstupní nebo obojí; to se vyjadřuje předáním argumentu proceduře buď kopírováním datové hodnoty (předání hodnotou) nebo poskytnutím adresy argumentu, tj. ukazatele (předání odkazem). Procedury pro příbuzné účely se často sdružují do modulů (knihoven). Procedury odvolávající se na sebe se nazývají rekurzivní, procedury s týmž jménem, ale různým rozhraním se nazývají přetížené.

Deklarace a volání

Deklarace funkcí a procedur jsou tvořeny hlavičkou a tělem (blokem) a vkládají (vnořují) se do těla programu, modulu nebo jiných funkcí a procedur.

```
function jméno funkce(formální argumenty) : typ návratové hodnoty; tělo funkce;
procedure jméno procedury(formální argumenty); tělo procedury;
```

Formální argumenty jsou jména lokálně reprezentující data předávaná z místa volání (skutečné argumenty); nemusejí být žádné. Slova argument a parametr jsou někdy záměnné, jindy se parametrem myslí formální parametr/argument a argumentem skutečný parametr/argument. Funkce vrací návratovou hodnotu, musí být tedy deklarován její typ; návratovou hodnotu je třeba přiřadit proměnné téhož jména jako funkce nebo proměnné jménem result. Tělo funkce a procedury obsahuje podobně jako tělo hlavního programu deklarace lokálních dat a složený příkaz; může obsahovat další (vnořené) funkce a procedury.

Př.:

```
function pi: real;                                  begin pi:= 3.141592653589793 end;
function sqr(x: extended): extended;                begin result:=x*x end;
procedure prumery(a,b : real; var arit,geom : real); begin arit:=(a+b)/2; geom:=sqrt(a*b) end;
```

Volání funkcí a procedur provedou odskok z místa volání s návratem po provedení. Volání funkce vrací návratovou hodnotu a je tedy výrazem, funkci lze volat kdekoliv, kde lze použít výraz, např. jako součást jiného výrazu. Volání procedury je samostatným příkazem.

```
jméno_funkce(skutečné argumenty);
jméno_procedury(skutečné argumenty);
```

Skutečné argumenty jsou buď obecným výrazem, který se před předáním nejprve vyčíslí, nebo jménem proměnné (nebo prvku pole nebo položky struktury), nebo jménem jiné procedury (procedurální argument). Jejich počet musí odpovídat počtu formálních argumentů.

Př.: přiřazovací příkaz s voláním funkce

```
x:=pi;
```

volání procedury s argumenty, jimiž jsou standardní funkce: první bez argumentu, druhá s jedním atd.

```
writeln(pi, exp(1), LogN(10,100), LogN(exp(1),sqr(exp(1))));
```

V dalším textu této části se píše o procedurách, vše však platí i pro funkce.

Poznámka o volání v OOP (objektově orientovaném programování)

Procedury (metody) vázané na strukturovaný typ (třídu) a potažmo na proměnnou tohoto typu (objekt) s touto proměnnou obvykle pracují. Je zažitá následující syntaxe pro volání metod:

místo `procedure(object,other_arguments)`
lze volat `object.procedure(other_arguments)`.

Předávání argumentů

Úkolem je zajistit jednak jednosměrné předávání vstupních argumentů (z místa volání do procedury) a výstupních argumentů (z procedury ven), jednak obousměrné předávání argumentů, které tak jsou vstupně-výstupní. Snahou též bývá zajistit efektivní předávání rozměrnějších dat, tj. polí a záznamů. Vše se děje tzv. předáváním hodnotou nebo odkazem (ukazatelem).

Předávání argumentů hodnotou (by value, default). Formální argumenty jsou nově alokovanými lokálními proměnnými, do kterých jsou zkopírovány hodnoty skutečných argumentů. Aktuální hodnoty formálních argumentů

se při návratu zpět nekopírují. Vhodné pro vstupní skalární argumenty; kopírování vstupních argumentů o značné velikosti (polí, záznamů) může být plýtváním. Skutečnými argumenty mohou být obecné výrazy.

Př.:
```
procedure ParamVal(i : integer);
begin i:=9 end;
begin i:=1; write(i); ParamVal(i); writeln(i); end        =>   11   (skutečný argument je proměnná)
begin i:=1; write(i); ParamVal((i)); writeln(i); end      =>   11   (skutečný argument je výraz)
```

Předávání argumentů odkazem (by reference, klauzule var). Formální argumenty jsou ukazateli na skutečné argumenty. Nekopírují se tedy hodnoty skutečných argumentů, ale předávají se jejich adresy. Změny na formálních argumentech se tak provádějí v místě skutečných argumentů. Skutečné argumenty musí být adresovatelné, musí tedy být např. proměnnou, nikoliv obecným výrazem. Vhodné pro vstupně-výstupní argumenty libovolné velikosti. Pro vstupní argumenty je vhodnější použít předání hodnotou nebo konstantní argumenty.

Př.:
```
procedure ParamRef(var i : integer);
begin i:=9 end;
begin i:=1; write(i); ParamRef(i); writeln(i) end         =>   19
begin i:=1; write(i); ParamRef((i)); writeln(i) end       nelze, skutečný argument není adresovatelný
```

Předávání konstantních argumentů (klauzule const). Konstantní formální argumenty nesmí být v proceduře změněny, mohou tedy reprezentovat jen vstupní argumenty. Příslušným skutečným argumentem však může být proměnná (s adresou) i obecný výraz (bez adresy) a podle toho se automaticky provede předání buď hodnotou nebo odkazem. Jde tak o vhodnou volbu pro předávání rozměrných vstupních argumentů.

Př.:
```
procedure ParamConst(const i : integer);
begin { NELZE: i:=9} end;
begin i:=1; write(i); ParamConst(i); writeln(i) end       =>   11
begin i:=1; write(i); ParamConst((i)); writeln(i) end     =>   11
```

Předávání výstupních argumentů (klauzule out). Výstupní formální argumenty by měly v proceduře získat hodnotu, neboť jejich vstupní hodnota nemusí být převzata z volající procedury. Prakticky však jde o argumenty předané odkazem.

Př.:
```
procedure ParamOut(out i : integer);
begin i:=9 end;
begin i:=1; write(i); ParamOut(i); writeln(i) end         =>   19
```

Předávání polí bez udání velikosti (otevřená pole, open array parameters). Z předchozího plyne, že pole se obvykle předávají s klauzulemi const nebo var, tedy odkazem. Skutečné a formální argumenty předávané odkazem však musí být identického typu, který pro statická pole zahrnuje i popis mezí, a to je přílišné omezení. Je proto umožněno deklarovat formální argument typu pole bez uvedení velikosti; formální argument pak převezme velikost od skutečného argumentu. Bázový typ a počet rozměrů se shodovat musí. Syntaxe

```
procedure ParamArr(const a : array of BaseType);
```

je stejná jako při deklaraci dynamických polí, nejde však o dynamické pole, ale o popis formálního argumentu. Otevřeným polím mohou být předávána pole statická i dynamická. Otevřená pole mají dolní mez (funkce Low) vždy 0 a horní mez (funkce High) tedy rovnu Length-1; v tom jsou podobná polím dynamickým, předaná statická pole jsou tak však přeindexována.

Př.: nulování statického nebo dynamického 1D pole libovolné délky
```
procedure Clear(var a: array of real);                    // pole předané odkazem
var i: integer;
begin for i :=Low(a) to High(a) do a[i]:=0 end;
```

Přiřazení skutečných argumentů formálním je poziční, tj. přiřazuje se první skutečný argument prvnímu formálnímu, druhý druhému atd. Počty skutečných a formálních argumentů se proto musí shodovat. Předávání argumentů hodnotou umožňuje předávat argumenty neidentických typů podle pravidel o kompatibilitě vůči přiřazení, tedy s možností typové konverze jako v přiřazovacích příkazech. Při předávání odkazem však skutečné a formální argumenty musí být identického typu, s uvedenou výjimkou pro otevřená pole.

Poznámka. Jazyky (nikoliv Pascal) někdy nabízejí syntaxi pro volitelné argumenty. K takto deklarovanému formálnímu argumentu nemusí existovat příslušný skutečný argument. Volitelný argument bez přiřazeného skutečného argumentu musí mít uvedenou defaultní hodnotu nebo nesmí být za běhu programu používán. Variantou volitelných argumentů je možnost deklarovat proměnný počet argumentů.

K volitelným argumentům se váže možnost předávat argumenty nikoliv pozičně, ale přiřazením jménu formálního argumentu. Např. ve Fortranu lze standardní funkci pro vyjádření podmíněného výrazu merge(tsource,fsource,mask) volat s pozičním přiřazením, merge(1/i,0,i/=0), i s přiřazením pomocí formálních jmen, merge(mask=i/=0,tsource=1/i,fsource=0).

Procedurální argumenty

Bývá žádoucí, aby procedura mohla přijmout jako argument jméno jiné procedury. Procedury a funkce jako formální argumenty musí mít procedurální typ, popisující jejich rozhraní a u funkce návratovou hodnotu.

Př.
```
type tFun = function (x : real) : real;
function rtbis(f : tFun; x1,x2,xacc : real) : real;
```
```
type tProc = procedure (x : real; var f,df : real);
function rtnewt(p : tProc; x1,x2,xacc : real);
```

Procedura jako skutečný argument musí mít rozhraní odpovídající typu příslušného formálního argumentu a měla by být uvozena adresním operátorem @.

Př.
```
function f1(x : real) : real; ...
function f2(x : real) : real; ...
writeln(rtbis(@f1,a,b,eps),rtbis(@f2,a,b,eps));
```
```
procedure p1(x : real; var f,df : real); ...
procedure p2(x : real; var f,df : real); ...
writeln(rtnewt(@p1,a,b,eps),rtnewt(@p2,a,b,eps));
```

Rekurzivní procedury

Rekurzivní neboli na sebe se odvolávající zápisy jsou v matematice běžné s účelem převést úlohu na obdobnou, ale jednodušší.

Př. n-faktoriál: $\begin{cases} 0! &= 1 \\ N! &= N(N-1)! \text{ pro } N = 1, \ldots \end{cases}$

V leckterých případech (i zde) existuje sice i alternativní nerekurzivní zápis, $N! = \prod_{n=1}^{N} n$, ne však vždy. Rekurzivní algoritmy se často používají např. při práci s dynamickými datovými strukturami nebo při řešení úlohy půlením. (Př. Setřídit pole? Setřiď obě poloviny a vhodně je sluč.) Programovací jazyky rekurzivní volání procedur obvykle umožňují, přímo (procedura volá sama sebe) i nepřímo (první procedura volá druhou, druhá volá první). Nezbytnou součástí rekurzivní procedury je podmíněný příkaz s testem ukončení (př. nerekurzivní větev pro 0-faktoriál).

Př. n-faktoriál:

```
// rekurzivní funkce
function FactRec(const n : integer) : real;
begin
  if n<=0 then result:=1        // test ukončení
  else result:=n*FactRec(n-1);  // rekurzivní volání
end;
```
```
// nerekurzivní funkce
function Fact(const nmax : integer) : real;
var n : integer;
begin
  result:=1;
  for n:=1 to nmax do result:=result*n;
end;
```

Citlivé je užívání procedur s vícenásobným rekurzivním voláním. Je to v pořádku, pokud vícenásobná rekurze odpovídá vnitřní struktuře algoritmu nebo dat (např. při průchodu binárním stromem). Vícenásobná rekurze nad lineární strukturou (např. nad posloupností) ovšem hrozí plýtváním.

Př. Fibonacciho posloupnost

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_N &= F_{N-1} + F_{N-2} \text{ pro } N = 2, \ldots \end{aligned}$$

Při volání podle definice, result:=FibRec(N-1)+FibRec(N-2), roste výpočetní složitost s N exponenciálně, při volání podle odvozených vzorců

$$\begin{aligned} F_{2N-1} &= F_N^2 + F_{N-1}^2 \\ F_{2N} &= F_N(F_N + 2F_{N-1}) \end{aligned}$$

složitost roste jen lineárně, stejně jako při výpočtu cyklem.

Přetížené procedury

Často se vyskytují procedury s obdobným úkolem, ale různým rozhraním – různým počtem nebo datovým typem formálních argumentů. Takové procedury mohou mít stejné jméno a říká se jim pak přetížené (nebo také generické, zastupující specifické procedury). Syntaxe: za hlavičky přetížených procedur se přidá klauzule overload.

Př.: funkce rozhodující se pro celočíselné a reálné dělení podle typu argumentů

```
function Divide(X, Y: Real): Real; overload;
begin Result:= X/Y end;
function Divide(X, Y: Integer): Integer; overload;
begin Result:= X div Y end;
begin writeln(Divide(1,2),Divide(1.,2),Divide(1,2.),Divide(1.,2.)) end.
```

Lokální a globální proměnné

Rozsah platnosti proměnné: od deklarace do konce bloku
tj. při deklaraci proměnné nad deklarací procedury/funkce: proměnná vzhledem k proceduře globální
při deklaraci proměnné v bloku procedury/funkce: proměnná v proceduře lokální
při deklaraci proměnné za blokem procedury/funkce: proměnná v proceduře nedostupná
Při shodě jmen globální proměnné a lokální proměnné nebo formálního argumentu:
globální proměnná v proceduře zastíněná
Lokální proměnné si po opuštění bloku neuchovávají definovanost.

Kam s procedurami

Některé jazyky umožňují poskládat procedury do zdrojového kódu sekvenčně v libovolném pořadí, jindy je možné procedury vnořovat do těla programu (nebo jiné procedury), jindy sdružovat do samostatných modulů, které se pak k programům explicitně připojují. Pascal nabízí vnořování a moduly.

Vnořování procedur do bloku vnější programové jednotky: deklarace procedury se umístí pod deklarace globálních proměnných (dostupných v proceduře) a nad deklarace lokálních proměnných volající programové jednotky (nedostupných v proceduře). Procedura musí být (v Pascalu) deklarována nad místem svého volání.

Př.
```
program prog;          // hlavička programu
var ...                // globální proměnné programu
procedure proc(...);   // hlavička vnořené procedury
var ...                // lokální proměnné procedury
begin ... end;         // tělo procedury
var ...                // lokální proměnné programu
begin ... end.         // tělo programu
```

Moduly (units) slouží jako kolekce (souvisejících) proměnných a procedur, které lze volat po připojení modulu (příkaz uses) z různých programů. Moduly mohou obsahovat veřejná i (v modulu) soukromá data a procedury. Veřejná data a procedury se vyjmenují v rozhraní modulu (interface), deklarace všech procedur se pak umístí do implementační části modulu (implementation).

Př.
```
// modul
unit m;              // hlavička modulu

interface            // rozhraní modulu
var ...;             // veřejné deklarace
procedure proc(...);
...
implementation       // implementační část
var ...;             // lokální data modulu
procedure proc(...); // implementace procedur
begin ... end;
...
end.                 // konec modulu
```

```
// program
program prog;
uses m;              // připojení modulu
var ...;             // lokální proměnné programu
begin ... end.       // blok programu
```