# Kapitola 3

# Podprogramy

Jak již název napovídá, podprogram je část programu („malý“ program), která může být kdykoliv v příkazové části programu (hlavního, „velkého“ programu) aktivována. V jazyce Pascalu rozeznáváme dva druhy podprogramů (podrobně je o nich pojednáno v [3]):

- Klasický podprogram, který nazýváme *procedura*. Jedná se v podstatě o příkaz.[1]
- Funkční podprogram, který nazýváme *funkce*. Jeho úkolem je poskytnout určitou hodnotu (funkční hodnotu).

Podprogramy (podalgoritmy) umožňují při algoritmizaci určitého problému použití systematické tvorby postupu řešení. Hovoříme o tzv. strukturovaném programování. Moderní postup algoritmizace vycházející z metody strukturovaného programování předpokládá rozčlenění (dekompozici) řešeného problému na podproblémy samostatně algoritmizovatelné. Podproblémy můžeme zapsat jako podprogramy, případně provést další stupeň dekompozice určitého podproblému (hovoříme o programování shora dolů). Podprogramy, na jejichž řešení se může podílet řada lidí, tvoří základní stavební prvky, z nichž komponujeme výsledný program (programování zdola nahoru).Podprogramy velmi napomáhají k čitelnosti programů.

Osvojení si způsobů jejich výroby (deklarace) a použití (volání) umožňuje aplikovat systematičnost a dělbu práce na tvorbě programů. Nezanedbatelná je rovněž možnost vhodně použitým podprogramem podstatně zkrátit celý program. Jak norma jazyka, tak především implementace jazyka přináší řadu standardních podprogramů, jejichž existence nám přináší zmíněné výhody: zkrácení a zpřehlednění textu programu. Vždyť bez procedur např. read, readln, write, writeln či funkcí např. sin, sqrt a dalších, si algoritmizaci v Pascalu nedovedeme ani představit.

Pronikněme nejprve do problematiky algoritmů vyřešením jednoduchého problému v následujícím Příkladě 3.1.

**Příklad 3.1:** Sestavte podprogram pro výpočet součtu celočíselných hodnot z intervalu <A,B> s přírůstkem C za předpokladu, že A,B jsou celočíselné hodnoty.

Úkolem je tedy vypočítat $S = \sum_{i=A}^{B} i$ , je-li přírůstek $\Delta i = 1$ a platí, že A<B.

Řešení podprogramu je, jak jsme již uvedli, buď v podobě procedury, nebo v podobě funkce. Podprogram, řešící určitý problém, lze sestavit vždy v podobě procedury. Uvedený problém řešíme deklarací procedury např. ve tvaru:

```
Procedure SUMA(FA,FB:integer; var FS:integer);
   var   I:integer;
   begin FS:=0;
         I:=FA;
         while I<=FB do begin FS:=FS + I;
                              I:=I + 1
                        end
   end;
```

[1] Celý program lze chápat jako jeden příkaz. Již jsme uvedli příkazy:„Najdi nejmenší hodnotu z řady čísel“, „Vyřeš kvadratickou rovnici“ atp. Čtenář asi cítí, že oba uvedené příkazy lze dekomponovat vždy na např. tři příkazy: „Sežeň (načti) potřebné hodnoty“, „Proveď zpracování vstupních hodnot“ a „Přehledně vytiskni požadované výsledky“. Řešení uvedených podproblémů může být v podobě podprogramů.

Poněvadž výsledkem požadovaného úkonu má být pouze jedna hodnota jednoduchého typu, je možno zadaný problém řešit také deklarací funkce např. ve tvaru:

```
Function SOUCET(FA,FB:integer):integer;
   var I,POM:integer;
   begin POM:=0;
         I:=FA;
         while I<=FB do begin POM:=POM+I;
                              I:=I+1
                        end;
         SOUCET:=POM
   end;
```

V obou předvedených podprogramech mohlo být místo cyklu **while–do** použito cyklu **repeat–until** a vzhledem k práci s proměnnými A,B ordinálního typu a přírůstku 1 (a jen právě proto) použito cyklu **for–to–do** (kdyby byl přírůstek různý od 1, popř. obě nebo některá ze vstupních proměnných A,B by byla typu desetinného čísla, pak by nebylo lze příkaz cyklu **for–to–do** použít).

Kompletní program s ukázkou použití deklarovaných podprogramů může mít např. následující podobu:

```
Program UKAZKA;
   var A,B,VYSL:integer;
   Procedure SUMA(FA,FB:integer;var FS:integer);
      var    I:integer;
      begin  FS:=0; I:=FA;
             repeat FS:=FS + I;
                    I:=I + 1
             until   I>FB;
      end;
   Function SOUCET(FA,FB:integer):integer;
      var   I,POM:integer;
      begin POM:=0; I:=FA;
            repeat POM:=POM+I;
                   I:=I+1
            until  I>FB;
            SOUCET:=POM;
       end;
   begin write('Zadej meze od - do: ');readln(A,B);
         SUMA(A,B,VYSL);
         writeln('Hodnota součtu od ',A,' do ',B,' je:',VYSL);
         VYSL:=SOUCET(A,B);
         writeln('Stejnou hodnotu bychom měli vytisknout i nyní: ',VYSL);
         .
         .
         .
   end.
```

konec Příkladu 3.1 konec Příkladu 3.1

Demonstrujme dále vliv podprogramů na zpřehlednění popř. i zkrácení zápisu programu.

## Příklad 3.2: Sestavme podprogram pro výpočet kombinačního čísla.

$$KC = \left( \begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{(n-k)!\ .k!}$$

Platí, že $X! = X * (X-1) * (X-2) * \ldots * 2 * 1 * 0!$, kde $0! = 1$.

Program, v podstatě bez myšlenky, s prokázáním pouze určité manuální zručnosti může mít tvar:

```
Program  KOMBCISLO;                          {Ve standardním Pascalu:           }
   var   P,KC:longint;                       {   Program KOMBCISLO(input,output);}
         N,K,I:byte;                         {       var P,KC,N,K,I:integer;    }
```

```
   begin write('Zadej hodnoty pro výpočet N nad K: ');
         readln(N,K);
         P:=1;
         for I:=2 to N do P:=P*I;
         KC:=P; P:=1;
         for I:=2 to K do P:=P*I;
         KC:=KC div P; P:=1;
         for I:=2 to (N-K) do P:=P*I;
         KC:= KC div P;
         writeln('Výsledek ',N,' nad ',K,' je: ',KC);
   end.
```

Realizací myšlenky zapsat obecně (formálně) algoritmus pro výpočet faktoriálu v podobě podprogramu a ten pak v programu třikrát použít, jistě dospějeme ke kratšímu zápisu programu. Poněvadž od podprogramu očekáváme poskytnutí *jedné* jednoduché (v našem případě celočíselné) hodnoty, budeme ho deklarovat v podobě funkce:

```
Program  KOMBCISLO;
   var   KC:longint;
         N,K:byte;
   Function FAKT(X:byte):longint;
      var I:byte; POM:longint;
   begin  POM:=1;
          for I:=2 to X do POM:=POM*I;
          FAKT:=POM
   end;
   begin write('Zadej hodnoty pro výpočet N nad K: ');
         readln(N,K);
         KC:=FAKT(N) div FAKT(K) div FAKT(N-K);
         writeln('Výsledek ',N,' nad ',K,' je: ',KC);
   end.
```

Z hlediska našeho zadání není nezbytné deklarovat funkci pro výpočet kombinačního čísla. Jistě si však dovedeme představit program, kde by bylo třeba v různých místech získávat hodnotu kombinačního čísla $\binom{n}{k}$.

Podprogram (funkce) pro poskytnutí kombinačního čísla pak bude mít zřejmě v sobě deklarovanán další podprogram – funkci pro výpočet faktoriálu tak – jak ukazuje následující řešení:

```
Program  KOMBCISLO;
   var   KC:longint;
         N,K:byte;
   Function KOMB(H,D:byte):longint;
      Function FAKT(X:byte):longint;
      var    I:byte; POM:longint;
      begin  POM:=1;
             for I:=2 to X do POM:=POM*I;
             FAKT:=POM
      end;                                          {*  Konec funkce FAKT  *}
   begin KOMB:=FAKT(H) div FAKT(D) div FAKT(H-D);
   end;                                             {*   Konec funkce KOMB *}
begin write('Zadej hodnoty pro výpočet N nad K :');
      readln(N,K);
      KC:=KOMB(N,K);
      writeln('Výsledek ',N,' nad ',K,' je: ',KC);
end.
```

Příkazová část výše uvedeného programu může být pochopitelně též ve tvaru

```
begin write(' Zadej hodnoty pro výpočet N nad K :');
      readln(N,K);
```

```
      writeln('Výsledek ',N,' nad ',K,' je: ',KOMB(N,K))
end.
```

Uvedenou funkci FAKT pro výpočet faktoriálu je možno zapsat také jako funkci rekurzivní, tj. funkci, která volá sama sebe. Vycházíme při tom z definice faktoriálu podle vztahu: $x! = x \cdot (x-1)!$, přičemž $0! = 1$.

```
Function FAKT(X:byte);longint;
   begin if N<2 then FAKT:1
                else FAKT:=N*FAKT(N-1)
   end;
```

Těm čtenářům, kteří si činí ambice na tvorbu rekurzivních podprogramů doporučujeme porovnat rekurzivní i nerekurzivní funkce FAKT, které jsou výše uvedeny a uvědomit si, že každý cyklus lze převést na rekurzi a naopak. Tak, jako cyklus

**for** I:=N **downto** 2 **do** POM:=POM∗I resp.

```
I:=N;
while I>1 do begin POM:=POM*I;
                   I:=I–1
           end;
```

resp.

```
I:=N;
repeat  POM:=POM*I;
        I:=I–1
until   I< 2;
```

(všechny tři zápisy vedou ke stejnému efektu) má svoji koncovou podmínku
I<2 resp. **not**(I>1) resp. I<2
má i rekurze svoji podmínku (N<2) při jejímž splnění již není dále rozvíjena.

Ve výše uvedených algoritmech je jedno úzké místo: výpočet faktoriálu i pro poměrně malá čísla dává vysokou hodnotu výsledku. Tak např. již 8! je 40320, což je hodnota větší, než je v Turbo Pascalu maximální zobrazitelná hodnota typu integer. A již 13! je 6 227 020 800, což je hodnota větší, než v Turbo Pascalu maximální zobrazitelná hodnota (typu longint) 2 147 483 647 (je to totéž, jako ve škále implementací PASCALu hodnota integer). Tuto nevýhodu odstraňuje následující strategie algoritmizace výpočtu hodnoty $X!$, kdy výsledná hodnota není vytvářena v (malé) proměnné jednoduchého typu, ale v poli číslic, které obsahuje výslednou hodnotu:

```
const MAX = 30;                              {*   maximální délka výsledku *}
type  CISLICE = array [1..MAX] of 0..9;
procedure FAKT(X: byte; var F: CISLICE);
var P,                                       {*   přenos                   *}
    I,                                       {*   index                    *}
    POM  :byte;                              {*   mezivýsledek násobení    *}
begin
   for I:= 1 to MAX - 1 do  F[I] := 0;
   F[MAX] := 1;
   while X > 1 do                            {*   cyklus 1*2*...*X         *}
      begin P := 0;
            I := MAX;
            repeat POM  := F[I] * X + P;
                   F[I] := POM mod 10;       {*   jedna řádová číslice     *}
                   P := POM div 10;          {*   přenos do vyššího řádu   *}
                   I := I - 1
            until I = 0;
            X := X - 1;
      end;
end;                                         {*   konec procedury FAKT     *}
```

Závěrem tohoto příkladu uveďme ještě vskutku již fundovaný způsob řešení výpočtu kombinačního čísla, při kterém se zároveň vyhneme výpočtům faktoriálů. Z uvedeného vzorce pro výpočet kombinačního čísla lze odvodit vztah

$$\left(\begin{array}{c} n \\ k \end{array}\right) = \frac{n}{k}\left(\begin{array}{c} n-1 \\ k-1 \end{array}\right) \text{ přičemž platí, že } \left(\begin{array}{c} n \\ 0 \end{array}\right) = 1.$$

Pak řešení v podobě rekurzivní funkce je

```
Function KOMB(H, D : byte) : longint;
   begin if D=0 then KOMB  := 1
                else KOMB := KOMB(H-1, D-1)*H div D;
   end;
```

Funkce KOMB uvedená výše nám poslouží stejnými výsledky, jako funkce KOMB uvedená na straně 37. Porovnání přehlednosti a úspornosti necháváme na čtenáři.

konec Příkladu 3.2 konec Příkladu 3.2

Snad první úloha, se kterou se každý začínající student – programátor setká, je úloha výpočtu hodnot podle nějakého vzorce. Vždyť jak nás dokáže potrápit zpracování provedených měření v rámci laboratorních cvičení z fyziky. Výpočet podle určitého vzorce je však pouze jednou stránkou problému. Neméně důležitá je však i úprava výstupní sestavy — protokolu z měření. V následujícím příkladu uvedeme problém tabelace funkce s předpokládanými výstupy na obrazovku. Přesměrování případných výstupů na tiskárnu je již jen rutinní záležitostí s uvědoměním si, že zatímco jedna obrazovka pojme např. 25 řádků po 80 znacích, na jeden list papíru můžeme umístit např. 60 řádků po 60 znacích.

## Příklad 3.3: Tabelujte funkci y= f(x) pro všechna $x \in \langle A, B \rangle$, s přírůstkem $\Delta x = C$.

Řešením pro funkci např. W=mgh (potenciální energie) pro výšku h od A (např. 1m) do B (např. 10m) po kroku $\Delta$h = C (např. 0.1m) je tabulka čítající cca X(100) řádků s výsledky. Vyřešení hlavičky tabulky je dílem vhodně poskládaných příkazů tisku, vlastní výpočet provedeme přiřazovacím příkazem a tisk každého z výsledkových řádků tabulky příkazem procedury writeln. Nelze však chaoticky vytisknout např. na obrazovku všech X řádků za sebou, neboť bychom po ukončení viděli na obrazovce pouze posledních 24 řádků (ostatní řádky by „utekly“). Tuto skutečnost je třeba ošetřit.

```
Program TABELACE;
  const G=9.80665;
  var   A,B,C,W,M,H : real;
        PR          : byte;
        Z           : char;
  Procedure PODTRH(ZNAK:char; POCET:byte);
     var   I : integer;
     begin for I:=1 to POCET do write(ZNAK);
           writeln;
     end;
  Procedure HLAVICKA;
     begin writeln('    Tabulka výpočtu potenciální energie');
           write(' ':4);
           PODTRH('-',35);
           writeln;
           PODTRH('*',49);
           writeln('|   výška   | hmotnost  |     g     |  energie  |');
           PODTRH('*',49);
     end;
  begin write('Zadej hmotnost tělesa: '); readln(M);
        write('Zadej meze výšky pro měření (OD - DO - PO): '); readln(A,B,C);
        writeln('***  Pozor, první výsledky tabelace přichazejí  ***');
        writeln;
        HLAVICKA;
        H:=A; PR:=0;
        while H<=B do
```

```
        begin W:=M*G*H;
              PR:=PR+2;
              if (PR mod 18) = 0 then
                 begin write('Až budeš chtít pokračovat, odešli znak: ');
                       readln(Z);
                       HLAVICKA;
                       writeln('| ',H:8:2,'  | ',M:8:4,'  | ',G:9:5,' | '
                                   ,W:9:4,' |');
                       PODTRH('-',49)
                 end
                                 else
                 begin writeln('| ',H:8:2,'  | ',M:8:4,'  | ',G:9:5,' | '
                                   ,W:9:4,' |');
                       PODTRH('-',49)
                 end;
              H := H + C;
        end;
      writeln;
      write('   ***   konec tabelace   ***  -  odešli znak:');
      readln(Z);
end.
```

konec Příkladu 3.3 konec Příkladu 3.3

**Příklad 3.4:** Vytvořte podprogram pro výpočet funkce sin(x)

Pro výpočet aproximace funkcí $\sin(x)$, $\cos(x)$, $\exp(x)$ (tj. $e^x$), aproximace integrálů aj. využíváme rozvoje daných funkcí v nekonečnou řadu. Tak např.

$$\begin{aligned}
\sin(x) &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots \\
\cos(x) &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \cdots \\
e^x &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots \\
\int_0^x e^{-u^2} du &= x - \frac{x^3}{3.1!} + \frac{x^5}{5.2!} - \frac{x^7}{7.3!} + \frac{x^9}{9.4!} - \cdots
\end{aligned}$$

Řešení: Součet řady je dán obecně

$$S = (\text{např.}\sin(x)) = T_0 + T_1 + T_2 + T_3 + \cdots$$

Nejjednodušší způsob sestavení algoritmu představuje tedy definování součtu jednotlivých členů řady. Uvědomíme-li si však, že např. hodnota $9! = 362880$ je zhruba o řád větší, než maximální zobrazitelné číslo celé (u mikropočítačů zpravidla 32767) a že při práci s desetinnými hodnotami počítáme na poměrně malý počet platných cifer (vzniká nežádoucí zaokrouhlování) je vhodné zamyslet se nad jiným postupem řešení.

Vycházíme z předpokladu, že se dá vyjádřit vztah mezi dvěma sousedními členy řady, tj. že platí

$$T_i = f(T_{i-1}) \text{ pro } \; i = 1, 2, 3, \ldots$$

Tak např. pro funkci $\sin(x)$ může platit, že např.

$$T_3 : T_2 = -\frac{x^7}{7!} : \frac{x^5}{5!}$$

a tedy

$$T_3/T_2 = \frac{-x^2}{7.6}$$

což zobecníme na

$$T_i/T_{i-1} = -x^2/(2i\ .\ (2i+1))$$

neboli

$$T_i = T_{i-1}\ .\ -x^2/(2i\ .\ (2i+1)).$$

Uvědomíme-li si, že význam posledního zápisu je postihnutelný zápisem v podobě přiřazovacího příkazu

$$\mathrm{T} := -\mathrm{T} * \mathrm{sqr(X)}/(2{*}\mathrm{I}{*}(2{*}\mathrm{I}{+}1));$$

(uvědomme si způsob zpracování přiřazovacího příkazu počítačem – nejprve je zpracováván výraz, nějaká hodnota v proměnné T (stará hodnota), po vyčíslení celého výrazu je nová hodnota umístěna opět do proměnné T (stará (předchozí) hodnota přepsána novou hodnotou)), pak sestavení úplného algoritmu je již manuální záležitostí.

Podprogram, samozřejmě v podobě funkce, může mít následující podobu:

```
Function SINUS(X:real) : real;
  const EPS = 0.00001;
  var SUMA, T : real;
      I       : byte;
  begin I := 0;   SUMA := X;   T := X;
        repeat I := I + 1;
               T := -T * sqr(X)/(2*I*(2*I+1));
               SUMA := SUMA + T
        until abs(T) < EPS;
        SINUS := SUMA;
  end;
```

Podotkněme ještě, že deklarace funkce s názvem SIN má za důsledek znepřístupnění standardní funkce *sin*.

konec Příkladu 3.4 konec Příkladu 3.4

**Příklad 3.5:** Deklarujte proceduru, která ze standardního textového souboru čte po znacích číslice a převádí je na hodnotu typu integer (do vnitřní reprezentace). Číslu v textovém souboru může předcházet libovolný počet mezer. Před prvou číslicí čísla může být znaménko. Poté následuje souvislá posloupnost číslic, ukončená mezerou.

```
procedure CTICISLO(var CISLO:integer);
   var ZNAK:char;
       ZNAMENKO:-1..1;
       {*DEKLAROVALI JSME DVE LOKALNI PROMENNE*}
   begin ZNAMENKO :=1;
         CISLO :=0;
         read(ZNAK);
         while ZNAK = ' 'do read(ZNAK);
         while ZNAK <>' ' do
            begin case ZNAK of
                       '+'   : ;
                       '-'   : ZNAMENKO:=-1;
                       '0'   : CISLO :=CISLO*10;
                       '1'   : CISLO := CISLO*10 + 1;
                       '2'   : CISLO := CISLO*10 + 2;
                       '3'   : CISLO := CISLO*10 + 3;
                       '4'   : CISLO := CISLO*10 + 4;
                       '5'   : CISLO := CISLO*10 + 5;
                       '6'   : CISLO := CISLO*10 + 6;
                       '7'   : CISLO := CISLO*10 + 7;
                       '8'   : CISLO := CISLO*10 + 8;
                       '9'   : CISLO := CISLO*10 + 9;
                  end;
```

```
                read(ZNAK)
            end;
        CISLO :=ZNAMENKO * CISLO
    end;
```

Vidíme, že použití příkazu **case** vede k potřebnému cíli přes řadu příkazů majících stejný význam. Příkazy, které jsou součástí příkazu **case**, je možno jistě zjednodušit (např. násobit deseti můžeme až po příkazu **case**, pokud příkaz **case** použijeme pouze k definování hodnoty pomocné proměnné typu 0..9), avšak podstata zůstává stejná – realizujeme řadu příkazů, které jsou natolik „podobné“, že musí existovat možnost podstatného zkrácení celého algoritmu.

Jistě je čtenáři již známo, že podmnožina znaků ’0’, ’1’, až ’9’ je uspořádaná a souvislá. Zná také význam standardní funkce *ord* a tedy může zkonstruovat příkaz

$$\text{CISLO:= CISLO}*10 + (\text{ord(ZNAK)} - \text{ord('0')});$$

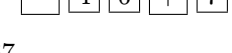
přičemž výraz ord(ZNAK) - ord(’0’) poskytuje celočíselnou hodnotu z intervalu 0..9 podle hodnoty v proměnné ZNAK, která je z intervalu ’0’..’9’ a je typu char. Pak příkaz **case** může mít tvar:

```
case ZNAK of '+' : ;
             '-' : ZNAMENKO:=-1;
             '0','1','2','3','4','5','6','7','8','9':
                          CISLO:=CISLO*10+(ord(ZNAK)-ord('0'))
end;
```

Označující konstanta (možnost, případ) je konstanta popřípadě seznam konstant. Řada implementací však umožňuje jako označující konstantu použít i interval, tedy např.

```
case ZNAK of '+'      : ;
             '-'      : ZNAMENKO := -1;
             '0'..'9' : CISLO:=CISLO*10+(ord(ZNAK)-ord('0'))
end;
```

Uvedenou strategií není mj. ošetřen případný výskyt znaménka mezi číslicemi. Tvar vstupu




vede k vytvoření celočíselné hodnoty +467.

Správným řešením je algoritmus ve tvaru:

```
begin ZNAMENKO :=1;
      CISLO :=0;
      read(ZNAK);
      while ZNAK = ' 'do read(ZNAK);
      if ZNAK='+' then read(ZNAK)
                  else if ZNAK='-' then begin ZNAMENKO:=-1;
                                              read(ZNAK);
                                        end;
      while ZNAK <>' ' do begin CISLO:=CISLO*10+(ord(ZNAK)-ord('0'));
                                read(ZNAK)
                           end;
      CISLO :=ZNAMENKO * CISLO
end;
```

Případné doplnění algoritmu o příkazy potřebné k vyřešení situace, která nastane po přečtení nějakého jiného znaku než mezery, znaménka nebo číslice, necháváme na úvaze studenta.

Mechanismus převodu vnější (textové) reprezentace celého čísla na vnitřní (binární) reprezentaci je součástí procedury read resp. readln. Součástí těchto procedur pracujících mj. i nad textovým souborem, je i algoritmus převodu vnější reprezentace desetinného čísla do vnitřní reprezentace a v mnoha implementacích i algoritmus čtení několika znaků (řetězce) do příslušné proměnné. Zpětné algoritmy převodu celočíselných, reálných i řetězcových hodnot vnitřní reprezentace do vnější (člověku čitelné) reprezentace, obsahují podprogramy write resp. writeln pracující nad textovým souborem.

konec Příkladu 3.5 konec Příkladu 3.5