

Začínáme programovat v jazyce FORTRAN 95

Jan Celý

Brno 2006
(stav k 18.12.2006)

Obsah

1	Začínáme	3
1.1	Překládáme první program	3
1.2	Jak psát programy	4
1.3	Co jsme naprogramovali	4
1.4	Začínáme počítat	5
1.4.1	Deklarace numerických proměnných	5
1.4.2	Přiřazovací příkaz a převod typů	6
1.4.3	Výpočet numerických výrazů – operandy a operátory	7
1.4.4	Deklarace s počátečními hodnotami a konstanty	8
1.5	Textové konstanty a proměnné – řetězce a podřetězce	8
1.6	Logické proměnné, relační operátory, rozhodovací příkaz	9
1.6.1	Logické konstanty a proměnné, logické operátory	9
1.6.2	Relační operátory	9
1.6.3	Rozhodovací příkaz a konstrukce IF	10
1.7	Vstup z klávesnice a jednoduchý cyklus DO	11
1.7.1	Vstup z klávesnice	11
1.7.2	Jak zajistit kontrolu vstupních dat v programu	12
1.7.3	Opakování bloku příkazů – jednoduchý cyklus DO a příkaz EXIT	13
1.8	Procedury a funkce	13
1.9	Moduly	16
1.9.1	Vytvoření modulu, generická jména procedur	16
1.9.2	Překlad modulu a programu s voláním modulu	19
1.10	Uživatелеm definované typy	19
2	Pole – vektory, matice a jim podobné objekty	23
2.1	Deklarace polí	23
2.2	Konstantní pole, počáteční hodnoty a funkce reshape	23
2.3	Subpole – výřezy z polí	24
2.4	Konformní pole, výrazy obsahující pole, konstrukce where	25
2.5	Alokovatelná pole	26
2.6	Textové řetězce a znaková pole	28
2.7	Další zabudované funkce pro pole	29
2.8	Možnosti příkazů PRINT, WRITE (nejen) pro výstup polí	32
3	Řídící konstrukce	35
3.1	Rozhodovací konstrukce IF	35
3.2	Konstrukce CASE	35
3.3	Cykly – konstrukce DO	38
3.3.1	Řízení cyklu, příkazy exit a cycle	38
3.3.2	Příkaz go to, vložené řídicí konstrukce	39
3.4	Příkaz a konstrukce FORALL	41

4	Vstupy a výstupy	43
4.1	Několik terminologických doplňků	43
4.2	Formátovaný vstup dat	44
4.3	Příkazy pro zápis a čtení	44
4.3.1	Propojení se souborem, specifikátor UNIT	45
4.3.2	Chyby v/v operací, konec souboru a záznamu – IOSTAT	45
4.3.3	Specifikátory ADVANCE, SIZE	45
4.4	Otevření a zavření souboru	46
4.4.1	Příkaz OPEN	46
4.4.2	Příkaz CLOSE	47
4.5	Vše o v/v prozradí příkaz INQUIRE	48
4.6	Neformátované v/v operace	49
4.7	Nastavení souborového ukazatele: BACKSPACE, REWIND, ENDFILE	49
5	Procedury, funkce a moduly	54
	Dodatky	55
A	Instalace kompilátoru G95 a jeho základní použití	56
A.1	Instalace kompilátoru G95	56
A.2	Instalace FortranTools	56
A.3	Základy práce s kompilátorem G95	57
A.3.1	Kompilace	57
A.3.2	Vytvoření vlastních knihoven a jejich použití	57
B	Editor SciTe	59
B.1	Instalace	59
B.2	Spolupráce editoru SciTe s G95	59
C	Zobrazení čísel v počítači a jejich typy v F95	61
D	Dodatek D	62
E	Grafika pro G95	63
E.1	Gnuplot	63
E.2	Grafická knihovna DISLIN	63
E.3	Grafické prostředí JAPI	64
	Literatura	65

Seznam tabulek

1.1	Zabudované funkce pro převod číselných typů	7
1.2	Operátor sjednocení řetězců a některé zabudované funkce pro řetězce	9
1.3	Logické operátory	9
1.4	Relační operátory	10
1.5	Položky formátovacích řetězců	18
2.1	Zabudované funkce související s mezemi, tvarem a velikostí pole	24
2.2	Zápis subpolí v jednodimensionálním poli	25
2.3	Zápis subpolí v dvoudimensionálním poli	25
2.4	Řídící znaky, které lze vkládat do formátovacího řetězce	32
4.1	Dotazovací specifikátory pro příkaz INQUIRE	48

Předmluva

Programovací jazyk FORTRAN (Formula Translator) byl vytvořen v polovině padesátých let minulého století u firmy IBM (tým vedeným J.M. Backusem). Již z názvu je zřejmé, že byl určen pro výpočetní práce (byly doby, kdy počítače jen počítaly) a tomuto zaměření zůstal věrný i během dalšího vývoje (1958 Fortran II, 1961 Fortran IV, 1977 Fortran 77, 1990 Fortran 90, 1995 Fortran 95, 2003 Fortran 2003). Až do verze F77 vyžadoval pevný formát zdrojového textu, který měl původ v děrných štítcích do nichž se kdysi vstupy pro sálové počítače děrovaly. Velkým zlomem ve vývoji byl Fortran 90, který dovolil používat volný formát a zavedl řadu nových programových struktur běžných v novějších programovacích jazycích (mnohé mají původ v jazyce Pascal a jeho následovnicích Modula2 a Oberon). Fortran 95 upřesnil některé prvky Fortranu 90 a doplnil konstrukce vhodné pro paralelní programování v souladu s dialektem HPF (High Performance Fortran). Definice pro Fortran 2003 vyšla v roce 2004; některé kompilátory F95 z ní přebírají několik konstrukcí jako rozšíření, úplný kompilátor však zatím není běžný. Protože rozsáhlá komunita uživatelů Fortranu je z pochopitelných důvodů (obrovské množství vypracovaných výpočetních programů) dosti konzervativní, musí být nové kompilátory schopné pracovat i v režimu F77. Jestliže však s programováním začínáte, není zvláštní důvod se detailně učit F77; stačí umět zkompileovat potřebné procedury a zapojit je do nových programů psaných už jen s využitím moderních konstrukcí F95. Právě pro výuku v tomto směru je, podle mého (a nejen mého) názoru, velice vhodný jazyk F.

Programovací jazyk F je pečlivě vybranou podmnožinou F95. Obsahuje jen moderní konstrukce Fortranu 95 a vhodně volenými omezeními nutí uživatele k disciplině programování. Pro začínající programátory, kteří se míní věnovat především výpočetní praxi, může být proto vhodným prvním programovacím jazykem. Práce na programech v jazyce F nebude ztracená při přechodu na úplný Fortran 95. Platí, že program napsaný v jazyce F musí být možné zkompileovat libovolným úplným kompilátorem F95; v následujícím textu budu takový kompilátor označovat F95.

Kompilátor jazyka F byl řadu roků volně přístupným produktem *The Fortran Company* ve verzi pro Windows i Linux. Počátkem března 2006 však došlo u této společnosti k zásadní změně: původní kompilátor jazyka F byl nahrazen balíkem *FortranTools*, který obsahuje jazyk F na bázi volně šiřitelného kompilátoru G95. Přesněji: balík prodávaný touto společností na CD obsahuje G95, zdarma lze stáhnout balík obsahující jen F vytvořený na bázi G95. Balík kromě kompilátoru obsahuje napojení na vývojové prostředí *Photran* verze 3.x (to je třeba stáhnout zvlášť), řadu užitečných knihoven a hlavně velmi cenný soubor dokumentace. Podstatné však je, že syntaxe této verze jazyka F se od předchozí v některých bodech liší; někdy ji mírně rozšiřuje, ale na druhé straně zase, podle mého názoru, zbytečně omezuje. Zcela chybí příkaz GOTO (a s ním i CONTINUE), není možný užitečný a přehledný jednořádkový IF-příkaz, v úvodu nesnese deklaraci IMPLICIT NONE (protože ji automaticky předpokládá, ale může chybět při překladu jiným kompilátorem) atd. Domnívám se, že syntaxe této verze je v souladu s prací kompilátoru G95 s klíčem -std=F. Pro některé uživatele Windows může být také nepříjemné, že tato verze balíku *FortanTools* pracuje v prostředí *cygwin* (emulace unixovského prostředí ve Windows).

V této situaci pokládám za rozumnější přejít přímo na volně dostupný kompilátor G95. Binární verze (velmi často aktualizované) je možné si pro nejrůznější platformy stáhnout ze serveru www.g95.org. Pro uživatele Windows, kteří nemají instalované prostředí *cygwin* je vhodný instalátor *g95-MinGW.exe*, který pracuje v prostředí *MinGW* ("Minimalistic GNU for Windows"). Podrobnosti instalace a základního použití najdete v dodatku A.

V dalším textu budou probírány programové konstrukce a zásady, které v podstatě odpovídají původní definici jazyka F. Doplněno je jen několik drobností zabudovaných v G95 (jde především o rozšíření směrem k Fortranu 2003), které mohou výrazně usnadnit programování (především interakci s operačním systémem); v textu na ně najdete upozornění, protože by je nemusel přeložit (na rozdíl od F) každý kompilátor F95. Znovu opakuji, že používání jazyka F programátora nijak neomezuje. Obsahuje vše co potřebujete k napsání libovolného programu a výsledkem vaší práce bude plnohodnotný program ve Fortranu 95.

1 Začínáme

1.1 Překládáme první program

Předpokládejme, že máme nainstalovaný fungující kompilátor jazyka g95 podle dodatku A. Založíme si pracovní adresář a v něm nějakým textovým editorem (Notepad, PsPad, SciTe a pod.) vytvoříme snad nejjednodušší možný program v g95 :

List. 1.1: Náš první program

```
PROGRAM prvni
PRINT *, "1+1=2"
END PROGRAM prvni
```

a uložíme ho pod jménem `prvni.f95`. V adresáři s tímto souborem otevřeme CMD-okno ¹⁾ a v něm napíšeme příkaz `g95 -c prvni.f95`. Je-li *program v pořádku*, nebude se chvíli (podle rychlosti PC) nic dít a potom se znovu objeví vstupní prompt. Jestliže se ale podíváme do adresáře (např. příkazem `dir`), uvidíme tam nový soubor `prvni.o`. To je již program přeložený do strojového kódu (tzv. objektový modul ²⁾), který ale neobsahuje vše potřebné k tomu aby mohl být spuštěn. Spustitelný soubor (s příponou `exe`) vytvoří až spojovací program, tzv. *linker*, který dokáže pospojovat více objektových modulů (včetně modulů ze standardních i externích knihoven) v jediný spustitelný soubor. To, že jsme provedli pouhou kompilaci do objektového modulu způsobil *klíč (option) -c* v zadaném příkazu. Při této kompilaci se ovšem dělo ještě něco významného – *prováděla se syntaktická kontrola programu*. Zkuste v našem programu udělat úmyslně chybu; umažte např. ve slově `program` ve třetím řádku `M` a program znovu uložte. Jestliže nyní provedete předchozí příkaz, uvidíte zprávu :

```
In file prvni.f95 : 3
end progra prvni
  1
Error: Expecting END PROGRAM statement at (1)
Error: Unexpected end of file in 'prvni.f95'
Exit code: 1
```

a soubor `prvni.o` se nevytvoří. Na základě vypsání informací provedeme opravu a opakujeme překlad s klíčem `-c` tak dlouho, až se vytvoří požadovaný objektový soubor. Je potřeba si uvědomit, že při překladu delších programových modulů s více chybami se při prvním pokusu *zpravidla nevypíše všechny chyby*. Navíc výpis chyb nemusí být vždy tak jednoznačný jako v tomto jednoduchém příkladu; počáteční chyby mohou kontrolní mechanismus kompilátoru „zmást“ a další výpisy nemusí odpovídat realitě. Opravovat chyby budeme proto postupně a po každé jasné opravě provedeme znovu pokus o kompilaci.

Jakmile projde první fáze (s klíčem `-c`) bez připomínek a vytvoří se objektový soubor, můžeme přikročit k *vytvoření spustitelného souboru*. V našem prostínkém případě k tomu stačí předchozí příkaz bez `-c`, tj. `g95 prvni.f95`. Kompilátor si vytvoří objektový soubor a hned přistoupí ke druhé – *linkovací* – fázi. Výsledkem je spustitelný soubor s příponou `exe`. Podíváte-li se však do adresáře, neuvídíte tam soubor `prvni.exe`, ale soubor `a.exe`. Důvod je prostý. Vytváří-li se výsledný soubor z více objektových modulů (jak to budeme brzy dělat) překladač neví které jméno mu přiřadit a proto vždy použije

¹⁾Ve Windows je otevřeme např. z menu *Start/Spustit* a do okna zapíšeme `cmd`, resp. v nižších verzích Windows `command`. Vlastnosti okna nastavíte klepnutím na ikonu v levém horním rohu okna a v menu zvolíte položku *Vlastnosti*. Použitelné příkazy vypíšete zapsáním *help*.

²⁾Přípona `o` je vlastní objektovým modulům v Linuxu; v DOSu a Windows mají zpravidla příponu `obj`.

a. exe. Můžeme mu ovšem přikázat, jaké jméno má výstupnímu souboru dát pomocí *klíče* -o. Po zadání příkazu

```
G:\WORK\g95>g95 prvni.f95 -o prvni
```

se nám v adresáři skutečně objeví soubor `prvni.exe`, příkaz `prvni` ho spustí a uvidíme

```
G:\WORK\G95>prvni
```

```
1+1=2
```

1.2 Jak psát programy

Programy v F95 (a tedy i v F) se mohou psát ve *volném formátu*. Zhruba řečeno to znamená, že úprava zápisu zdrojového textu závisí jen na nás; tam kde definice jazyka vyžaduje mezeru je možné zapsat jeden nebo více tzv. *bílých znaků* (mezera, tabelátor). Úpravu proto budeme volit takovou, aby výsledný *program byl přehledný a tím i snadno kontrolovatelný*. Značně při tom mohou pomáhat editory, které mají o syntaxi jazyka nějaké informace (např. editor SciTe doporučený v dod. B).

V programech zapsaných v *jazyce F se píše jeden jednoduchý příkaz na řádek* (nemůžeme např. na jeden řádek napsat dva příkazy `print`); složitější příkazy (programové konstrukce), s nimiž se seznámíme dále, naopak rozepisujeme přehledně na více řádků. Aby výpisy programů byly přehledné, je vhodné psát krátké řádky (zpravidla je editor omezuje na 80 znaků). Jak ale zapsat do programu jednoduchý příkaz, který se na takto omezenou řádku nevejde? Řešením je *znak pro pokračování řádku* – `&` – po jehož zapsání příkazový řádek pokračujeme na dalším řádku. Tak např. můžeme napsat

```
PRINT *, "1+1=", &      ! ukázka pokračování řádku
1+1
```

a výsledek bude stejný jako v předchozím případě. Dělicí znak je možné dát jen tam, kde nebude mást kompilátor (např. uvnitř textového řetězce by byl chápán jako jeden z jeho znaků, který se má tisknout).

Za příkazem `PRINT` jsem zapsal *komentář*. Komentář začíná znakem `!` a vše co je za ním až do konce řádku kompilátor ignoruje. Komentáře jsou pro Vás. Nepodceňujte je a při psaní programu na nich nešetřete. Často, zvláště na začátku programových modulů, jsou běžné dlouhé informační texty zapsané v *komentářových řádcích* se znakem `!` na začátku řádku.

1.3 Co jsme naprogramovali

Co jsme zatím naprogramovali? První a třetí řádek ohraničují programový modul v F95; mezi nimi může být libovolný počet povolených konstrukcí (příkazů) jazyka F95. Základní *struktura programového modulu* je v List. 1.2.

List. 1.2: Základní struktura programového modulu

```
PROGRAM <jmeno_programu>
  < konstrukce_jazyka_g95 >
END PROGRAM <jmeno_programu>
```

Zavedeme si při této příležitosti následujících *typografické dohody* pro zápis programových konstrukcí jazyka F:

- části které *dosadí programátor* budeme ve výpisech zapisovat takto `<.....>`,

- *klíčová slova jazyka F* budeme ve výpisech psát velkými písmeny (tak to automaticky dělá např. editor SciTe). *Jazyk F nerozlišuje velká a malá písmena* (není *case sensitive* jako např. jazyk C); dodržování této dohody však značně zlepší čitelnost programů,
- povinný výběr jedné položky z více možností zapíšeme {moznost_1|moznost_2|...},
- volitelné části (části, které nemusíme použít jestliže je nepotřebujeme) programových konstrukcí budeme uzavírat do hranatých závorek, tj. [volitelná_cast]; tyto závorky, stejně jako v předchozím případě {...}, nejsou součástí objektů, které uzavírají.

V programu `prvni.f95` (List.1.1) máme tedy jediný příkaz: `PRINT *, "1+1=2"`. Jeho funkci jsme poznali při spuštění programu `prvni.exe`: vypsál na obrazovku text (textový řetězec) uzavřený uvozovkami. Obecná struktura příkazu `PRINT` je:

```
PRINT < vystupni_format > , < seznam_vystupnich_polozek >
```

kde `vystupni_format` je {*|formatovaci_retezec} a

`seznam_vystupnich_polozek` jsou položky výpisu (řetězce, čísla atd) oddělené čárkami.

S formátovacími řetězci se seznámíte v kapitole 4. Varianta s * vypisuje jednotlivé položky seznamu (číselné hodnoty na maximální počet platných cifer), odděluje je mezerami a po vyčerpání seznamu přejde na nový řádek. Náš program `prvni` nic nepočítal. Na obrazovku pouze vypsál zadaný text.

1.4 Začínáme počítat

Z programu `prvni` vytvoříme `druhy.f95` malou úpravou. Výstupní seznam příkazu `PRINT` má nyní dvě položky: řetězec, který vystoupí tak jak je zapsán a *číselný výraz*, který se vypočte a výsledná hodnota se vypíše (oddělená mezerou).

List. 1.3: Náš druhý program sám počítá součet

```
PROGRAM druhy
PRINT *, "1+1=", 1+1      ! do seznamu je možné psát výrazy
END PROGRAM druhy
```

Vypsaný text ovšem zase může tvrdit nesmysl. Lepší by bylo, kdyby se vypisovala jak zadaná čísla tak i výsledek operace. Než to provedeme, všimněme si ještě jednoho rozšíření v programu `druhy`.

1.4.1 Deklarace numerických proměnných

Zatím jsme pracovali s konkrétními čísly (celočíselnými konstantami). Abychom mohli realizovat náš záměr, musíme operandy uložit do paměti počítače a nějak je pojmenovat. Programátorským jazykem: *deklarovat proměnné*. Jejich jména jsou tzv. *identifikátory* a každý jazyk má pravidla pro jejich tvorbu.

V jazyce F platí:

- jména proměnných (identifikátory) mohou obsahovat *alfanumerické znaky*, tj. A–Z, a–z, 0–9 a podtržítko "_",
- prvním znakem musí být *písmeno*, tj. znak z množiny A–Z, a–z,
- maximální délka identifikátoru je 31 znaků.

Identifikátory (jména) se přiřazují nejen proměnným, ale i jiným objektům; příkladem může být např. `<jmeno_programu>` v Tab.1.2. Všechna tato jména musí splňovat uvedené požadavky.

Přejděme k deklaraci proměnných. Existují jazyky (starší verze Fortranu, většina tzv. skriptovacích jazyků), které dovolují nebo přímo vyžadují *implicitní deklaraci*; proměnná se deklaruje když je její jméno poprvé uvedeno při zápisu programu. I když se to může na první pohled zdát výhodné, je to

z programátorského hlediska velice nešťastné. Při psaní programu může snadno dojít k překlepu (zaměnit třeba "n" a "m"), při překladu nebude kompilátor hlásit chybu a problémy se objeví až při běhu programu. A hledejte potom původ problémů v dlouhém zdrojovém textu. Jazyk Fortran 95 kvůli kompatibilitě se staršími verzemi Fortranu implicitní deklaraci povoluje, ale v nových programech je žádoucí ji vynutit příkazem IMPLICIT NONE hned na počátku programu. Jazyk F deklaraci vyžaduje. Protože programy napsané v F musí být možné přeložit libovolným kompilátorem F95, budeme tuto direktivu uvádět i v našich programech.

Proměnné mohou být nejrůznějších typů: celá, reálná a komplexní čísla, textové řetězce a pod. Tyto objekty se v počítači různě zobrazují a ukládají. Také operace, které se s nimi mohou provádět jsou různé. Tato problematika je ve Fortranu 95 velice detailně zpracovaná, protože je klíčová pro *přenositelnost programů* (výsledky výpočtů na různých počítačích musí být v mezích požadované přesnosti stejné). Tato problematika je podrobněji zpracovaná v dod. C. Zde se omezíme pouze na použití základních typů INTEGER a REAL. Další číselný typ je ještě COMPLEX pro komplexní čísla. Jsou reprezentována *uspořádanou dvojicí čísel typu REAL*. Zapisují se takto: (realna_cast, imaginarni_cast).

Doplníme předchozí program o deklaraci celočíselných proměnných (typu INTEGER), uložíme do nich číselné hodnoty a vypíšeme opět jejich součet.

List 1.4: Deklarace proměnných a přiřazovací příkaz

```
PROGRAM treti
IMPLICIT NONE      ! nepovoluje implicitní deklaraci
INTEGER :: m, n    ! deklarace dvou celočíselných proměnných
! vlastní začátek programu
m = 1              ! přiřazovací příkazy
n = 1
PRINT *, m, "+", n, "=", m+n
END PROGRAM treti
```

Výpočet a výstup tohoto programu už musí být v pořádku (pokud ovšem nepopleteme jediné dva řetězce ve výpisu: "+" a "=").

Proveďte nyní drobnou změnu – nahraďte v deklaraci typ INTEGER typem REAL. Výpis bude správný ale nehezky. Formát reprezentovaný "*", totiž vypisuje čísla s „plnou přesností“. Jak se dozvíte v dod. C, typ REAL odpovídá tzv. „jednoduché přesnosti“, která může dát asi 7 platných dekadických cifer. Nahraďte volný formát (tj. znak *) *formátovacím řetězcem* "(f4.1, a2, f4.1, a2, f4.1)" (samozřejmě i s uvozovkami) a pokuste se odvodit význam symbolů v něm; úplnou odpověď najdete v kap. 4.

1.4.2 Přiřazovací příkaz a převod typů

Vraťme se však ke zdánlivě jednoduchému *přiřazovacímu příkazu*, který má tvar

$$\text{proměnná} = \text{výraz}$$

V poslední verzi programu jsme měli deklarované reálné proměnné m, n a přiřazovali jsme jim celočíselné konstanty (neobsahovaly desetinnou tečku). Výstup programu ukázal, že při přiřazení byly převedeny na typ REAL, tj. typ proměnných, kterým se výrazy přiřazují. Že se to skutečně děje, můžete naopak ověřit v původní verzi programu treti; celočíselným proměnným m, n přiřaďte nějaké reálné hodnoty (např. m=1.3 a n=1.5) a podívejte se co program vypíše (mělo by to být 1 + 1 = 2).

Co se tedy při přiřazení děje? Vypočte se výraz a na výsledek se aplikuje jedna ze *zabudovaných konverzních funkcí* tak, aby její výsledek odpovídal typu *proměnné*. Použijí se k tomu funkce (označme vypočtenou hodnotu výrazu a) z tabulky 1.1

Tabulka 1.1: Zabudované funkce pro převod číselných typů

Typ <i>proměnné</i>	Funkce	Působení funkce na <i>a</i>
INTEGER	<code>int(a)</code>	a typu INTEGER ponechá Za a typu REAL dosadí nejbližší celé číslo směrem k 0 Pro a typu COMPLEX převede na typ INTEGER jeho reálnou část
REAL	<code>real(a)</code>	a typu INTEGER převede na typ REAL a typu REAL ponechá Pro a typu COMPLEX převede na typ REAL jeho reálnou část
COMPLEX	<code>cmplx(a)</code>	a typu INTEGER převede na reálnou část typu REAL a typu REAL převede na reálnou část a typu COMPLEX ponechá

Ve skutečnosti je situace ještě trochu složitější. Z dod. C víme, že všechny uvedené typy mohou mít několik možných hodnot *specifikátoru* KIND. Znovu platí, že se při přiřazení rozhoduje *specifikátor proměnné*. Je zřejmé, že *při přiřazení proměnné se může informace ztrácet*; při převodu REAL na INTEGER se „uřízne“ desetinná část, při konverzi REAL(KIND=DP) na REAL(KIND=SP) ubude asi polovina platných cifer mantisy a při převodu COMPLEX na REAL se ztratí imaginární část.

1.4.3 Výpočet numerických výrazů – operandy a operátory

V předcházejícím odstavci jsme jen konstatovali, že se výraz vypočte. Co je to však výraz a jak se vypočte? Výraz je kombinace operandů a operátorů vytvořená v souladu se syntaxí jazyka. Příkladem jednoduchého výrazu jsou dva operandy spojené binárním operátorem

operand *operátor* operand, např. $m+n$

nebo unární operátor a operand

operátor operand, např. $-m$.

Operandem může být konstanta, proměnná, funkce i sám výraz. Při vyhodnocení výrazu se uplatní *priorita operátorů*. Nejvyšší prioritu mají závorky; výrazy v závorkách se vyhodnotí nejdříve. *Výrazy bez závorek* vyhodnocuje Fortran *zleva doprava* a ctí při tom prioritu operátorů. Ve složitějších výrazech je však nevhodnější používat závorky; zpřehlední zápis a zřetelně zaručí postup vyhodnocení. Pro *skalární numerické výrazy* jsou operandy veličiny typu INTEGER, REAL, COMPLEX a operátory seřazené podle priority (klesá od $**$ k $+-$)

$**$ umocnění
 $*$ / násobení a dělení
 $+$ - sečítání a odečítání

POZOR: výsledek *dělení* s operandy typu INTEGER je vždy „uřezán“ směrem k 0, takže

$$9/3 = 3, 11/3 = 3, -11/3 = -3 \quad \text{a} \quad 2 ** (-3) = 0 \quad \text{neboť} \quad 2 * * (-3) = 1 / (2 * * 3).$$

V numerických výrazech je možné míchat veličiny všech tří typů (a s různými specifikátory KIND). Při výpočtu platí, že se před provedením operace konvertuje objekt s „nižší přesností“ (jednodušší, s menší informací) na příslušný typ s „vyšší přesností“ (složitější, nesoucí více informací) a tento typ má i výsledek operace. Vyjímkou z tohoto pravidla je pouze celočíselná mocnina reálného nebo komplexního čísla.

1.4.4 Deklarace s počátečními hodnotami a konstanty

Proměnná deklarovaná předchozím způsobem (např. INTEGER :: m) má v paměti rezervované místo, ale jeho obsah (hodnota proměnné) *není definován*. Přiřadit této proměnné hodnotu zatím umíme přiřazovacím příkazem. Je však také možné přiřadit jí počáteční hodnotu již při deklaraci takto ³⁾

```
INTEGER :: m = 1
```

Hodnota proměnných, které jsme zatím deklarovali, se může v průběhu práce programu měnit. Často však potřebujeme deklarovat veličiny – *konstanty* – jejichž hodnota by se zachovávala (nemohla se měnit ani omylem). Jejich deklarace se provede dodáním specifikátoru PARAMETER takto:

```
REAL, PARAMETER :: PI = 3.141592653, c = 2.997924E8
```

V obou případech (deklarace proměnné s počáteční hodnotou i konstanty) mohou být na pravé straně " =" výrazy, které obsahují již deklarované konstanty (nikoliv ale proměnné s počáteční hodnotou):

```
REAL, PARAMETER :: PI = 3.141592653, a0 = 5.2917706e-11
REAL, PARAMETER :: c = 2.997924E8, c2 = c*c
REAL :: plocha = PI*a0**2
```

1.5 Textové konstanty a proměnné – řetězce a podřetězce

Dosud jsme se zabývali jen proměnnými s numerickou hodnotou. Dalším zabudovaným typem jsou *znakové konstanty a proměnné*. Znakové konstanty jsme již používali v příkazech PRINT jako *posloupnost znaků mezi dvojitými uvozovkami*. Na rozdíl od 127 ANSI znaků (anglická abeceda, číslice, speciální znaky) s nimiž počítá definice Fortranu, je možné ve znakových konstantách a proměnných používat všech 255 ASCII znaků (127 ANSI + 128 znaků národních abeced, kódovaných v závislosti na použité kódové stránce).

Znakové proměnné se deklarují takto

```
CHARACTER(LEN=<len_hodnota>) :: <seznam_promennych>
```

kde *len_hodnota* je kladné celé číslo nebo *. Znak * lze použít jen pro formální parametr procedury nebo při *deklaraci znakové konstanty*

```
CHARACTER(LEN=*), PARAMETER :: pozdrav = "Nazdar"
```

Ve všech ostatních případech musíme použít přirozené číslo, které udává maximální délku znakového řetězce. Jestliže se pokusíme přiřadit proměnné délky *n* (LEN=*n*) delší řetězec, nebude se hlásit chyba a přiřadí se pouze prvních *n* znaků (zbytek řetězce se „uřízne“).

Je možné jednoduše pracovat i s částmi deklarovaného řetězce – *podřetězci* (substrings). Máme-li např. řetězec

```
CHARACTER(LEN=80) :: radek
```

potom *radek(i:j)* (*i, j* jsou přirozená čísla ≤ 80) je *podřetězec* (*substring*) obsahující všechny znaky od *i* do *j* v řetězci *radek*. Vystupuje-li v podřetězci první nebo poslední znak řetězce, můžeme použít zápis

```
radek(:i)   je totéž jako radek(1:i)
radek(i:)   je totéž jako radek(i:80)
radek(:)    je totéž jako radek(1:80)
```

³⁾Již zde je vhodné poznamenat: jde-li o *inicializovanou* lokální proměnnou v proceduře, je této proměnné automaticky přiřazen atribut SAVE. To znamená, že hodnota této proměnné se po opuštění procedury (na rozdíl od ostatních lokálních proměnných) zachovává a je k dispozici při dalším volání procedury. Dá se tedy využít např. jako počítadlo volání procedury.

Tabulka 1.2: Operátor sjednocení řetězců a některé zabudované funkce pro řetězce

//	operátor sjednocení (concatenation) dvou řetězců v jeden (např. "abc" // "de" dá jeden řetězec "abcde")
len(c)	vrací deklarovanou délku řetězce c
trim(c)	vrací řetězec bez mezer na začátku a konci řetězce
len_trim(c)	vrací délku řetězce bez úvodních a koncových mezer
adjustr(c)	vrací řetězec bez koncových mezer (zarovnává vpravo)
adjustl(c)	vrací řetězec bez úvodních mezer (zarovnává vlevo)
char(i)	vrací znak s kódem i v použité kódové stránce ($0 \leq i \leq 255$)
ichar(c)	vrací kód znaku c v použité znakové stránce (tj. <code>ichar(char(i))</code> vrátí i)

Tabulka 1.3: Logické operátory

Operátor	Název	Výraz je pravdivý když
.not.	unární operátor negace	operand je .false.
.and.	logický součin	oba operandy mají hodnotu .TRUE.
.or.	logický součet	alespoň jeden z operandů je .TRUE.
.eqv., .neqv.	ekvivalence / neekvivalence	oba operandy mají/nemají stejnou hodnotu

V tabulce 1.2 je uveden jeden operátor a několik zabudovaných funkcí pro práci s řetězci.

1.6 Logické proměnné, relační operátory, rozhodovací příkaz

1.6.1 Logické konstanty a proměnné, logické operátory

Logické konstanty jsou `.TRUE.` a `.FALSE.` (včetně teček na začátku a konci!). Logické proměnné a konstanty (mohou nabývat právě uvedené dvě hodnoty) se deklarují např. takto:

```
LOGICAL :: test
LOGICAL, PARAMETER :: ano = .TRUE., ne = .FALSE.
```

Logické konstanty, proměnné a funkce (vrací logickou hodnotu) mohou fungovat jako operandy v logických výrazech. *Logické operátory* (uspořádané podle priority) jsou tabulce 1.3 ve směru klesající priority od `.not.` k dvojici `.eqv., .neqv.` Poznamenejme, že operátor `.neqv.` funguje stejně jako operátor, který se zpravidla označuje `.xor.` (exklusivní logický součet). Znovu je vhodné zdůraznit, že závorky mají nejvyšší prioritu a jejich použitím zlepšíte čitelnost výrazu.

1.6.2 Relační operátory

Velice často potřebujeme testovat zda numerické výrazy (a podobně i textové výrazy) splňují určité podmínky. Tyto podmínky se formulují pomocí relačních operátorů, které jsou v Tab. 1.4.

Je-li alespoň jeden z operátorů typu `COMPLEX`, je možné použít pouze operátory `==`, `/=`. Výsledkem porovnání je vždy jedna z předdefinovaných logických konstant `.TRUE.`, `.FALSE.` Předpokládejme deklarace

```
INTEGER :: i, j
REAL :: a, b
CHARACTER(LEN=1) :: znak = "a"
LOGICAL :: L1, L2, L3, L4, L5, L6, L7
```

a všimněme si následujících výrazů:

Tabulka 1.4: Relační operátory

Operátor	Význam
<	je menší
<=	je menší nebo rovno
==	je rovno
/=	není rovno
>	je větší
>=	je větší nebo rovno

```

L1 = i<0
L2 = a<b
L3 = a+b>i-j
L4 = znak=="A"    ! L4 je .FALSE. protože ichar("a")>ichar("A")
L5 = "A"<"B"      ! L5 je .TRUE. protože ichar("A")<ichar("B")
L6 = "abcd">"abce" ! .FALSE. protože ichar("d")<ichar("e")
L7 = "abcd"<"abcd_" ! .FALSE. protože v kratším se doplní zprava mezera

```

Abychom se mohli přehledněji vyjadřovat, očíslovali jsme příkazy přiřazením výsledné logické konstanty logickým proměnným L1--L7.

V prvním a druhém případě nevidíme žádnou nejasnost, neboť se porovnávají proměnné stejných typů. Jak ale proběhne vyhodnocení třetí relace? Pravidlo je takové, že se nejprve vyhodnotí výrazy (*numerické operátory mají vyšší prioritu než relační*) podle pravidel, která již známe z odstavce 1.4.3, oba operandy se konvertují na „vyšší“ typ z obou a potom se porovnají. Pro L3 se vypočtou oba operandy a $i+j$ se převede na typ REAL.

Při porovnávání znakových proměnných se vlastně porovnávají celočíselné kódy jednotlivých znaků (viz. 1.5). Pro alfanumerické znaky s kódy do 127 platí *lexikografické uspořádání*:

```
"0"<"1"< ... <"9"< ... <"A"<"B"< ... <"Z"< ... <"a"<"b" ... <"z".
```

Při porovnávání řetězců se postupně porovnávají jednotlivé znaky zleva doprava a výsledek určí první rozdílná dvojice znaků (viz. L6). Jsou-li řetězce různě dlouhé, doplní se kratší mezerami zprava na délku delšího (viz. L7).

1.6.3 Rozhodovací příkaz a konstrukce IF

Logické výrazy jsou základním prvkem rozhodovacích (podmíněných) příkazů, které umožňují *větvení programu* podle jejich hodnoty. Jednoduchý IF příkaz se zapisuje takto:

```
IF (<logicky_vyraz> <vykonny_prikaz>
```

Je-li hodnota logického výrazu .TRUE., provede se <vykonny_prikaz>. V opačném případě se pokračuje příkazem na následujícím řádku. Uveďme příklad:

```
IF (a-b<0.0) a = 1.0
```

Mnohem bohatší možnosti nabízí IF-konstrukce, která v nejjednodušší podobě má tvar podle List. 1.5 Zde si připomeňme naši dohodu, že *části v hranatých závorkách jsou volitelné* (část začínající ELSE nemusí být použita). Funkce konstrukce je zřejmá: je-li podmínka (logicky_vyraz) splněna, provede se blok_prikazu_1; v opačném případě se provede blok_prikazu_2. Pokud není ELSE-část přítomna chová se IF-konstrukce stejně jako IF-příkaz; na rozdíl od něho však dovoluje místo jediného příkazu

List. 1.5: Struktura konstrukce IF...END IF

```

IF (<logicky_vyraz>) THEN
  <blok_prikazu_1>
[ELSE
  <blok_prikazu_2>]
END IF

```

zapsat libovolně dlouhý a složitý blok příkazů. Příklady obou možností jsou v následujících výpisech

```

IF (a<b) THEN
  pom = a
  a = b
  b = pom
END IF

```

```

IF (a<b) THEN
  a = -a
ELSE
  b = -b
END IF

```

Tyto příklady představují nejjednodušší IF-konstrukce; v kapitole 3 uvidíte, že jejich možnosti jsou mnohem bohatší.

1.7 Vstup z klávesnice a jednoduchý cyklus DO

1.7.1 Vstup z klávesnice

Náš poslední program List.1.4 dokáže sečíst jen dvě čísla, která jsou v něm deklarovaná. Užitečnější by byl ovšem, kdyby dokázal požádat o zadání těchto čísel s klávesnice. To samozřejmě jde a potřebný příkaz je analogický příkazu PRINT:

```

READ <FORMAT> [, <seznam_vstupnich_polozek> ]

```

kde format = { * | formatovaci_retezec } a <seznam_vstupnich_polozek> je seznam proměnných (oddělených čárkami), které se mají číst. Formát * je *volný formát*, který jedině má smysl pro čtení z klávesnice. Formátovací řetězce se mohou výborně uplatnit při čtení ze souborů, jestliže z nich chceme vybírat např. jen určité části (sloupce a pod.). Za povšimnutí stojí hranaté závorky, které naznačují, že *seznam vstupních položek může chybět* a příkaz má pak tvar

```

READ *

```

K čemu to může být dobré pochopíme, když zjistíme *jak čtení probíhá*.

Když program přijde k příkazu READ, zastaví se a čeká na zadání požadovaných vstupních dat. Vy je začnete vytvářet na klávesnici a vstup se současně zobrazuje na displeji. Každou ze vstupních položek ukončíte mezerou nebo novým řádkem (klávesou ENTER); za poslední položkou však musí být ENTER (nový řádek). Toto pravidlo Vám umožňuje přehledně uspořádat na obrazovce vstupní data, což je zvláště účelné např. při vstupu matic a pod. Navíc, pokud nestisknete ENTER, můžete data v řádku editovat. Zadávaná data se totiž čtou z klávesnice do tzv. vyrovnávací paměti (bufferu) a teprve po ukončení vstupu si je odtud převezme program a přiřadí je předepsaným proměnným. A k čemu tedy může být příkaz PRINT * dobrý? Spustíte-li některý z našich dosavadních programů tak, že na vytvořený exe-soubor poklepete myší v souborovém manažeru, otevře se CMD-okno, program vypíše co jste předepsali a okno se zavře. Stejně přitom stačí postřehnout záblesk okna. Jestliže ale jako *poslední příkaz programu* dáte

právě zmíněný PRINT *, program se na něm zastaví a bude čekat na závěrečný stisk klávesy ENTER, kterým se příkaz ukončuje.

Upravme tedy program List. 1.4 pro vstup čísel, která se mají sečíst:

List. 1.6: Doplnujeme vstup z klávesnice

```
PROGRAM ctvrty
IMPLICIT NONE      ! nepovoluje implicitní deklaraci
INTEGER :: m, n    ! deklarace dvou celocíselných promenných
                ! začátek programu
PRINT *, "Zadej_cela_cisla_m,n"
READ *, m,n       ! přečte čísla z klávesnice a přiřadí proměnným
PRINT *, m, "+", n, "=", m+n
PRINT *, "Program_ukonci_stisk_ENTER"
READ *           ! čeka na stisknutí klávesy ENTER
END PROGRAM ctvrty
```

Do programu jsme doplnili nejen potřebné příkazy READ, ale před každý z nich jsme ještě zadali výstup textu, který uživatele informuje na co počítač čeká. Program přeložte a ověřte, že správně pracuje. Potom si zkuste:

1. Zadat na řádek více čísel než dvě požadovaná. Po odklepnutí ENTER získáte poznatek, že program si vzal z buferu první dvě čísla a zbytek obsahu buferu ignoroval.
2. Zadat místo požadovaných celých čísel alespoň jedno reálné (s desetinnou tečkou). Na obrazovce uvidíte něco takového

a program se ukončí. Nestane se tedy to, co jsme mohli pozorovat v předchozím programu. Když jsme reálnou hodnotu zapsali do přiřazovacího příkazu (např. $m = 1.1$), byla podle pravidel v Tab. 1.1 převedena na celocíselnou hodnotu. Nyní se to považuje za fatální chybu a program se zastaví. Důvod je snadno pochopitelný. Jestliže programátor zapíše příkaz do programu, měl by vědět co a proč dělá. Vstupní data však mohou obsahovat nejrůznější chyby a je proto žádoucí trvat na tom, aby odpovídala předepsanému typu.

3. Zadat nenumerická vstupní data, např. dvojici a 3. Asi vás už nepřekvapí, když reakce bude stejná jako v předchozím případě.

1.7.2 Jak zajistit kontrolu vstupních dat v programu

Okamžité ukončení programu doprovázené výše uvedeným hlášením je poněkud nešťastný způsob indikace chyby ve vstupních datech. Inteligentní by bylo, zachytit informaci o chybě již v programu a *programově ji ošetřit*; např. tak, že se informace vypíše na obrazovku a uživateli se nabídne možnost zadat data znovu. Abychom to mohli provést, nevystačíme s použitou jednoduchou formou příkazu READ, ale musíme použít kousek z úplné formy popsané v kap. 4. Pro náš účel stačí toto:

```
READ(UNIT=*, FMT=*, IOSTAT=<celocíselna_promenna>)<seznam_vstupnich_polozek>
```

Od jednoduché formy se úplná forma liší tím, že v kulatých závorkách je možné, podle potřeby, uvést celou řadu pojmenovaných položek. Položka UNIT uvádí výstupní zařízení; * zde znamená displej (tzv. standardní vstup – *stdin*) a FMT uvádí formát, který se vyskytuje i ve zjednodušené formě (UNIT se tam

neuvádí, protože zjednodušená forma pracuje jen s displejem). Nová je zde tedy jen položka IOSTAT, která uloží informaci o průběhu vstupu do námi deklarované celočíselné proměnné. Jestliže příkaz READ proběhl bez chyby, je hodnota této proměnné 0 a pokud došlo k chybě je > 0 . Celočíselné kódy chyb (nejen vstupně-výstupních) jsou dány normou včetně pojmenování (část jich najdete např. v [4, str.6-2]). Víme nyní jak v programu zjistit chybu vstupu. Abychom mohli realizovat náš záměr – nabídnout uživateli opakování – potřebujeme ještě jednu programovou konstrukci.

1.7.3 Opakování bloku příkazů – jednoduchý cyklus DO a příkaz EXIT

Se všemi možnostmi cyklu DO se seznámíme v kap. 3. Zde nám stačí jeho nejjednodušší forma:

List. 1.7: Jednoduchý cyklus DO s příkazem EXIT

```
DO
  <prikazy_1>
  IF (<podminka>) EXIT
  <prikazy_2>
END DO
```

Jestliže by chyběl řádek IF (<podminka>) EXIT, opakovala by se do nekonečna skupina příkazů mezi DO a END DO. Při splnění podmínky v příkazu IF zajistí příkaz EXIT ukončení cyklu a pokračování programu příkazem následujícím za END DO. Nyní již máme vše potřebné pro realizaci našeho záměru. Program List. 1.6 upravíme na program List. ??.

List. 1.8: Vstup dat s možností opravy

```
PROGRAM paty
IMPLICIT NONE          ! nepovoluje implicitní deklaraci
INTEGER :: m, n, ios   ! do ios ulozi READ kod chyby (0 znaci bez chyby)
                  ! zacatek programu
PRINT *, "Zadej_cela_cisla_m,n"
DO
  READ(UNIT=*, FMT=*, IOSTAT=ios) m,n
  IF (ios == 0) EXIT    ! cteni bylo bez chyb, vyskocit z cyklu
  PRINT *, "Chybna_vstupni_data_(kod_chyby:_" , ios, ")._Opakujte_vstup!"
END DO
PRINT *, m, "+", n, "=", m+n
PRINT *, "Program_se_ukonci_stisk_ENTER"
READ *                ! ceka na stisknuti klavesy ENTER
END PROGRAM paty
```

1.8 Procedury a funkce

V posledním programu List. 1.8 jsme potřebovali pro ošetřený vstup z klavesnice několik programových řádků. Jestliže bychom potřebovali takový vstup ve více místech nějakého delšího programu, nebylo by

příliš přehledné a pohodlné stále přepisovat potřebný blok příkazů (i když by se vstupní data přiřazovala jiným proměnným a výzva pro vstup by byla jiná). Elegantně se tento problém dá vyřešit tak, že potřebná *skupina příkazů se deklaruje jako procedura s formálními parametry*. Ve chvíli kdy budeme potřebovat provést tyto příkazy, *zavoláme proceduru s aktuálními parametry*. Struktura deklarace procedury je v List. 1.9

List. 1.9: Deklarace procedury

```
SUBROUTINE <jmeno_proc>(<seznam_formalnich_parametru>)
  <specifikacni_prikazy>
  <vykonne_prikazy>
END SUBROUTINE <jmeno_proc>
```

kde jednotlivé položky mají tento význam:

jmeno_proc

je jméno (identifikátor), kterým budeme deklarovanou proceduru volat.

seznam_formalnich_parametru

je čárkami oddělený *seznam identifikátorů* objektů, pomocí nichž procedura komunikuje s vnějším okolím.

specifikacni_prikazy

deklarují *typ a přístupový charakter* formálních parametrů. Některé standardní typy už umíme deklarovat (INTEGER, REAL apod.). Způsob přístupu k jednotlivým formálním parametrům se stanoví pomocí *přístupových specifikátorů*: INTENT(IN) pro vstupní parametry, INTENT(OUT) pro výstupní a INTENT(INOUT) pro parametry, které mají obojí funkci.

vykonne_prikazy

jsou příkazy, které vykonávají vlastní činnost procedury a tvoří tzv. *tělo procedury*. Na jeho počátku mohou být deklarované *lokální proměnné*, které se při volání procedury nadeklarují, používají se a po ukončení práce procedury se zruší. Jejich identifikátory jsou platné uvnitř procedury a mají přednost před proměnnými téhož jména deklarovanými vně procedury. Pravda je, že uvnitř procedury je možné použít i proměnné deklarované vně procedury, tzv. *globální proměnné*. Procedura, která by tyto tzv. *vedlejší efekty* používala je však téměř k ničemu. Když ji budete chtít použít v jiném programu, musíte zajistit deklaraci příslušné globální proměnné (jejíž identifikátor už mohl být použit k něčemu jinému) a ladění (hledání a odstraňování chyb) programu se silně znepráhlední. Stručné: dobrá procedura komunikuje se svým okolím **jen** přes formální parametry v hlavičce procedury.

Kam však máme deklaraci procedury zapsat? Odpověď dává List. 1.10, který vznikl z List. 1.2 doplněním CONTAINS následovaným blokem deklarace procedur a funkcí použitých (volaných) v úvodní programové části.

Program List. 1.8 přepsaný tak, aby vstup obstarávala procedura je v List. 1.11. Všimněte si v něm několika nových věcí:

- Volání procedury se děje příkazem
CALL <jmeno_procedury>(<seznam_skutecnych_parametru>)
- Místo dosud používaného příkazu PRINT jsme použili jeho úplnou formu WRITE (podrobnosti opět v kap. 4). To nám umožňuje zadání řídicích specifikátorů ovlivňujících výstup. Zde jsme konkrétně použili specifikátor ADVANCE={"yes"|"no"}; s řetězcem "no" po vypsání seznamu výstupních hodnot *nepřejde na nový řádek*. Jak se dozvíte v [?, R912], tento specifikátor lze použít jenom

List. 1.10: Struktura programového modulu s deklarací procedur a funkcí

```
PROGRAM <jmeno_programu>
  < programova_cast >
CONTAINS
  <deklarace_procedur>      ! deklarace procedur(SUBROUTINE) a funkci(FUNCTION)
END PROGRAM <jmeno_programu>
```

u formátovaných výstupů. Proto jsme zadali formátovací řetězec pro výstup znakového řetězce: FMT="(a)".

- Dvojici příkazů na konci programu jsme nahradili procedurou CekejEnter. Z její deklarace je vidět, že WRITE bez ADVANCE skutečně snese FMT=* (příkaz je ekvivalentní PRINT *, "..."). Důležitější poučení je však toto: i když je seznam formálních parametrů prázdný, *musí být v deklaraci i při volání procedury uvedeny závorky* ().

List. 1.11: Program 1.8 s deklarací procedur

```
PROGRAM sestý
IMPLICIT NONE
INTEGER :: i

CALL CtiCisloI("Zadej_cele_cislo",i)      ! Volani procedury
PRINT *, "Zadane_cislo_", i
CALL CekejEnter()

CONTAINS

SUBROUTINE CtiCisloI(vyzva,i)
  CHARACTER(LEN=*),INTENT(IN) :: vyzva      ! specifikacni prikazy
  INTEGER,INTENT(OUT) :: i
  INTEGER :: ios                             ! vykonna cast (telo procedury)
  WRITE(UNIT=*,FMT="(a)",ADVANCE="no") vyzva
  DO
    READ(UNIT=*,FMT=*,IOSTAT=ios) i
    IF (ios==0) EXIT
    PRINT *, "Chyba_v_zadani,_zadejte_znovu"
  END DO
END SUBROUTINE CtiCisloI

SUBROUTINE CekejEnter()
  WRITE(UNIT=*,FMT=*)"Cekam_na_ENTER"
  READ *
END SUBROUTINE CekejEnter

END PROGRAM sestý
```

Za podrobnější zmínku ještě stojí *vstupní a výstupní parametry*. Parametry s INTENT(IN) se nemohou v těle procedury měnit (podrobněji [?, R512]). Při volání procedury se za ně mohou dosadit jak konstanty tak proměnné. Za parametry s INTENT(OUT) a INTENT(INOUT) je naproti tomu nutné dosazovat iden-

tifikátory proměnných, které procedura může změnit. Procedura v tomto případě totiž pracuje skutečně s dosazenými proměnnými, zatímco u vstupních (IN) parametrů pracuje s kopiemi, které si vytvoří a při výstupu z procedury je zruší.

Vedle právě uvedených procedur je jistě vhodné mít možnost *deklarovat funkce*, které bude možné psát do programu stejně jako standardní (zabudované) funkce. Deklarace se provede podle List. 1.12.

List. 1.12: Deklarace funkce

```
FUNCTION <jmeno_funkce>(<formalni_parametry>) RESULT(jmeno_vysledku)
  <specifikacni_prikazy>
  <vykonne_prikazy>
END FUNCTION <jmeno_funkce>
```

Vidíme, že jde o jistou modifikaci deklarace List. 1.9. Základní rozdíly jsou:

- seznam *formalni_parametry* může obsahovat *jen* parametry s přístupovým specifikátorem *INTENT(IN)*; parametrem procedur i funkcí však může být i procedura nebo funkce, která přístupový specifikátor nemá (o tom až v 5).,
- pro *jmeno_vysledku* se přístupový specifikátor neuvádí (je už dán slovem *RESULT*),
- *jmeno_vysledku* se musí objevit alespoň jednou na levé straně přiřazovacího příkazu ve výkonných příkazech.

Jednoduchý příklad deklarace funkce je v List. 1.13. Tento zdánlivě zbytečný příklad má praktický význam. Parametrem procedury nebo funkce sice může být funkce, *nesmí to však být žádná ze zabudovaných (intrinsic) funkcí*. Budu-li potřebovat aby skutečným parametrem procedury nebo funkce byla funkce $\sin(x)$, mohu to dosáhnout jedině pomocí takto deklarované funkce.

V testovacím programu si kromě deklarace funkce *Sinus* všimněte ještě dvou věcí :

- Běžný trik pro získání čísla π s požadovanou přesností je v řádce 5. Zkuste, jestli se ve výpisu programu něco změní, když do programu vložíte místo toho „přesnější“ konstantu `REAL, PARAMETER :: pi=3.1415926535897931`.
- V řádce 10 jsme použili novou položku formátovacího řetězce. Abychom mohli slušně formátovat výstupy již od počátku, uveďme v tabulce 1.5 možné položky formátovacích řetězců

1.9 Moduly

1.9.1 Vytvoření modulu, generická jména procedur

Deklarace procedur⁴⁾ v programovém bloku je jistě výhodná. Jestliže ale vytvoříme proceduru, která má širší použití, jistě by nebylo praktické ji pokaždé kopírovat do programu, který ji má použít. Jazyk F (Fortran 90 a vyšší) nabízí elegantní řešení: procedury se vloží do speciálního programového bloku, ten se přeloží do objektového tvaru (soubor s příponou *o*, viz. 1.1) a při výsledné kompilaci se nabídne linkovacímu programu, který zabuduje žádanou proceduru do výsledného programu. Struktura tohoto speciálního programového modulu je v List. 1.16.

Přístup k procedurám a případně i objektům deklarovaným v oblasti *specifikacni_prikazy* získá program tak, že hned za hlavičkou programu se uvede příkaz

⁴⁾Zde i jinde v textu používám termín *procedura* jak pro skutečné procedury (SUBROUTINE) tak i pro funkce (FUNCTION).

List. 1.13: Testovací program s deklarací funkce Sinus

```

1 PROGRAM T_Sinus
2 IMPLICIT NONE
3 REAL :: pi,x,dx
4
5 pi = 4*atan(1.0)      ! vypocte PI pomoci zabudovane funkce arctan
6 PRINT *,pi           ! vypise na "plnou presnost" pro typ REAL
7 dx = pi/10          ! krok vypisu
8 x = 0
9 DO
10  PRINT "(f10.7,2es15.7e1)",x,Sinus(x),sin(x)
11  IF (x>pi) EXIT
12  x = x+dx
13 END DO
14 READ *              ! ceka na stisk ENTER
15
16 CONTAINS
17
18 FUNCTION Sinus(x) RESULT(vysledek)
19  REAL,INTENT(IN) :: x
20  REAL :: vysledek
21  Sinus = sin(x)
22
23 END FUNCTION Sinus
24 END PROGRAM T_Sinus

```

USE <jmeno_modulu>.

Protože i modul může využívat objekty deklarované v jiných modulech, najdeme takovéto příkazy hned za hlavičkou většiny modulů. V modulech, které nabízím pro výuku numerických metod to bude vždy modul `std_type` v němž jsem soustředil označení základních numerických typů, které budeme používat (viz. ??).

Ze struktury v List. 1.14 je vidět, že kromě hlavičky a závěrečného řádku může vše chybět (připomínám dohodnutý význam [...]); takový modul by byl asi stejně užitečný jako náš první program 1.1 bez příkazu `PRINT`. Z druhé strany, že chybí deklarace `procedur`, není neobvyklé. Příkladem může být zmíněný modul `std_type`. Blok `specifikacni_prikazy` může totiž zavádět řadu užitečných obecně použitelných objektů (viz. 5). Pro ilustraci si vytvoříme modul, který bude obsahovat `procedur` deklarovaných v List. 1.11; zobrazen je v List. 1.15

List. 1.14: Struktura programového bloku MODULE

```

MODULE <jmeno_modulu>
  [<specifikacni_prikazy>]
  [CONTAINS
    <deklarace_procedur>]
END MODULE <jmeno_modulu>

```

Tabulka 1.5: Položky formátovacích řetězců

<i>Položka</i>	<i>Význam</i>
Iw[.m]	Celé číslo na celkem w pozic, m je celkový počet vypsanych cifer (doplní se úvodní nuly)
Fw.d	Reálné číslo: formát s pevnou desetinou tečkou, w je celkový počet míst, d počet míst za des. tečkou.
ESw.d[Ee] ENw.d[Ee]	Reálné číslo v exponenciálním tvaru: w je celkový počet míst, d míst za des. tečkou, e počet míst pro exponent i se znaménkem. S ES je výstup ve vědeckém tvaru s mantisou v intervalu [0,10) a EN je tzv. inženýrský tvar, který má exponent rovný násobku 3.
Lw	Logická hodnota na w pozic; vypíše se T nebo F zarovnané vpravo.
A[w]	Textový řetězec. Bez w je počet pozic určen délkou řetězce. Je-li w přítomno a je menší než délka řetězce, vypíše se prvních w znaků, v opačném případě se výpis zarovná vpravo.
<i>Poznámky:</i>	Před každým specifikátorem může být přirozené číslo n, které udává počet opakování; např. 3fw.d je ekvivalentní fw.d, fw.d, fw.d. Do celkového počtu míst w se započítává i znaménko (i když se + nevypisuje); totéž platí i pro místa pro výpis exponentu e.

List. 1.15: Modul Cti s procedurami z List. 1.11

```

MODULE Cti
  IMPLICIT NONE
  PUBLIC :: CtiCisloI, CekejEnter
  CONTAINS
    ! sem se zkopiruje deklarace obou procedur z List1.11
  END MODULE Cti

```

V modulu se nám v bloku `specifikacni_prikazy` objevilo nové slovo `PUBLIC`. U objektů zavedených v modulu musí být totiž vždy jasné, zda jsou použité jen uvnitř modulu (tzv. *privátní*) a zvenčí (tedy pro uživatele modulu) nepřístupné a nebo jsou naopak určeny k volání zvenčí. Proto musí být u každého takového objektu uveden jeden ze *specifikátorů dosažitelnosti*: `PRIVATE` nebo `PUBLIC`. Ve výpisech většiny modulů si jistě všimnete další možnosti; specifikátor `PRIVATE` je uveden hned v záhlaví modulu za příkazem `USE`. Tím se dosáhne toho, že *vše v modulu bude privátní* a veřejné přístupné bude pouze to u čeho bude *explicitně* uveden specifikátor `PUBLIC` (F dovoluje tento postup jen v modulech, které mají uvedeno alespoň jedno `USE`, viz. [?, R1104]).

Než se pustíme do překladu a použití tohoto modulu, zvažme ještě jeho *další rozšíření*. Procedura `CtiCisloI` dovolí číst pouze celá čísla. Aby bylo možné *číst i čísla reálná* (typu `REAL`), doplníme deklarační část modulu o deklaraci procedury `CtiCisloR`, kterou získáme snadno několika drobnými úpravami `CtiCisloI`. V programech používajících tento modul pak budeme volat jednu nebo druhou proceduru, podle toho jaké číslo budeme číst. Jazyk F (Fortran95) nabízí elegantnější řešení: zvolíme si nějaké *genericke jméno*, např. `CtiCislo`, a necháme na kompilátoru, aby sám podle typu čteného čísla rozhodl, kterou ze dvou deklarovaných procedur použít. Musíme mu ovšem dát informaci, mezi kterými procedurami má vybírat, když dostane např. příkaz

```
CALL CtiCislo("Zadej_cislo", x).
```

Uděláme to tak, že ve specifikační části uvedeme řádky podle 1.16. Výsledný modul je v List 1.17.

List 1.16: Zavedení generického jména pro více příbuzných procedur

```

...
PUBLIC :: CtiCislo           ! pouze toto jmeno bude verejne pristupne
PRIVATE :: CtiCisloI, CtiCisloR   ! puvodni jmena mohou zustat skryta
INTERFACE CtiCislo
    MODULE procedure CtiCisloI, CtiCisloR
END INTERFACE
...

```

1.9.2 Překlad modulu a programu s voláním modulu

Samotný modul přeložíme standardním příkazem `F -c <jmeno_souboru>`, kde `jmeno_souboru` je soubor s deklarací modulu; toto jméno se nemusí shodovat se jménem modulu. Předpokládejme pro určitost, že deklaraci modulu `Cti` podle List 1.17 napíšeme do souboru `m_cti.f95`. Po bezchybném provedení příkazu `F -c m_cti` se v pracovním adresáři objeví dva soubory:

```
m_cti.o ,   cti.mod.
```

Objektový soubor má jméno souboru v němž je deklarace modulu `Cti` a soubor s příponou `mod` má jméno modulu. Důvod pro vytvoření těchto dvou souborů je prostý. V souboru `m_cti.f95` nemusí být jen deklarace modulu `Cti`. Soubor může klidně obsahovat deklaraci několika modulů (případně i testovací program). V objektovém modulu (zde `m_cti.o`) bude překlad všech součástí zdrojového textu (zde `m_cti.f95`) a v souborech `*.mod` budou uloženy informace o jednotlivých modulech (jsou to textové soubory, podívejte se do nich).

Pro otestování modulu `Cti` napíšeme krátký testovací program `T_cti.f95` (List 1.18).

Příkazem `F -c T_cti` získáme objektový soubor `T_cti.o`. Pokusíme-li se ale získat `T_cti.exe` příkazem `F T_cti`, skončí pokus hlášením tohoto typu

Důvod je prostý: spojovací program (linker) nemohl najít přeloženou (binární) podobu modulu `Cti`, aby požadovaný kód zapojil do výsledného programu. Že je tento kód v souboru `m_cti.o` mu sdělíme následujícím zápisem příkazu

```
F m_cti.o T_cti -o T_cti.
```

Obecně bude příkaz vypadat takto:

```
F <seznam_obj_souboru> <zdrojovy_soubor> -o <vystupni_soubor> ,
```

kde `<seznam_obj_souboru>` je čárkami oddělený seznam všech potřebných objektových souborů. Příkaz v této podobě vyžaduje, aby všechny potřebné soubory byly v témže (pracovním) adresáři. V odst. ?? se dozvíte, jak z objektových modulů vytvořit knihovnu a při kompilaci ji volat.

1.10 Uživatelem definované typy

Zatím jsme pracovali se skalárními proměnnými předdefinovaných numerických typů (např. `INTEGER`, `REAL`) a textovými řetězci. V následující kapitole se ještě seznámíme s poli (arrays) – pravoúhlými jedno a více dimenzionálními tabulkami prvků téhož typu. Jazyk `F` (Fortran95) však umožňuje, aby si

programátor definoval objekty vlastního typu. Obecně to je pojmenovaná struktura⁵⁾ objektů různých typů. Uved' me oblíbený příklad. Náš program má pracovat se seznamem osob. Jistě by bylo výhodné mít potřebné údaje o každé osobě soustředěné tak, aby je bylo možné označit jediným identifikátorem nebo jako jednu položku pole osob. Fortran95 nám to umožní definicí typu, který nazveme např. OSOBA, podle List. 1.19.

Proměnné typu OSOBA budeme deklarovat takto:

```
TYPE(OSOBA) :: Josef,Karel,Petr
```

K jednotlivým položkám typu OSOBA se dostaneme takto:

```
Josef%prijmeni  
Petr%vek
```

To jsou již proměnné příslušné typu, takže můžeme např. součet stáří všech tří osob zapsat

```
celkem = Petr%vek + Josef%vek + Karel%vek
```

Přiřazovací příkaz pro typ OSOBA (typ už musí být definovaný) vypadá takto

```
Vaclav = OSOBA("Vaclav", "Sova", 22.4, 123654)
```

Položkou uživatelsky definovaného typu může být i dříve definovaný uživatelský typ. Mnoho příkladů uživatelem definovaných typů najdete např. v modulu Jap i Graf, který vám nabízím (viz. dod. E).

⁵⁾V Pascalu tomu odpovídá RECORD a v C pak STRUCT.

List. 1.17: Modul Cti s deklarací generického jména CtiCislo

```
MODULE Cti
  IMPLICIT NONE

  PUBLIC :: CtiCislo,CekejEnter
  PRIVATE :: CtiCisloI, CtiCisloR

  INTERFACE CtiCislo
    MODULE procedure CtiCisloI, CtiCisloR
  END INTERFACE

  ! interni globalni promenna, nemusi byt deklarovana v procedurach
  INTEGER,PRIVATE :: ios

  CONTAINS

  SUBROUTINE CtiCisloI(vyzva,i)
    CHARACTER(LEN=*),INTENT(IN) :: vyzva
    INTEGER,INTENT(OUT) :: i
    WRITE(UNIT=*,FMT="(a)",ADVANCE="no") vyzva
    DO
      READ(UNIT=*,FMT=*,IOSTAT=ios) i
      IF (ios==0) EXIT
      PRINT *,"Chyba_v_zadani,_zadejte_znovu"
    END DO
  END SUBROUTINE CtiCisloI

  SUBROUTINE CtiCisloR(vyzva,r)
    CHARACTER(LEN=*),INTENT(IN) :: vyzva
    REAL,INTENT(OUT) :: r

    WRITE(UNIT=*,FMT="(a)",ADVANCE="no") vyzva
    DO
      READ(UNIT=*,FMT=*,IOSTAT=ios) r
      IF (ios==0) EXIT
      PRINT *,"Chyba_v_zadani,_zadejte_znovu"
    END DO
  END SUBROUTINE CtiCisloR

  SUBROUTINE CekejEnter()
    WRITE(UNIT=*,FMT=*)"Cekam_na_ENTER"
    READ *
  END SUBROUTINE CekejEnter

END MODULE Cti
```

List. 1.18: Testovací program pro modul Cti

```
PROGRAM T_Cti
USE Cti
INTEGER :: i
REAL :: r

CALL CtiCislo("Zadej_cele_cislo_",i)
PRINT *,"Zadane_cislo_=",i
CALL CtiCislo("Zadej_realne_cislo",r)
PRINT *,"Zadane_cislo_=",r

CALL CekejEnter()

END PROGRAM T_cti
```

List. 1.19: Deklarace typu OSOBA

```
TYPE OSOBA
  CHARACTER(LEN=15) :: jmeno
  CHARACTER(LEN=15) :: prijmeni
  REAL :: vek
  INTEGER :: oc      ! osobni cislo
END TYPE OSOBA
```

2 Pole – vektory, matice a jim podobné objekty

2.1 Deklarace polí

V kap. 1 jsme pracovali pouze se skalárními proměnnými. Ze zkušenosti však víme jak široké použití ve výpočetní praxi mají vektory, matice, tj. *jedno- a dvou- dimenzionální pravoúhlá pole prvků téhož typu*. Fortran95 je pro práci s číselnými (ale nejenom číselnými) poli neobyčejně dobře vybaven. Deklaraci pole je možné provést nejrůznějšími způsoby. Začněme od nejjednoduššího.

Jednorozměrné pole čísel typu REAL s předepsanou délkou deklarujeme podle tohoto vzoru

```
REAL, DIMENSION(15) :: a .
```

Tímto příkazem jsme deklarovali pole s patnácti prvky

```
a(1), a(2), ... , a(14), a(15) .
```

Prvky pole jsou indexované celými čísly. Všimněte si, že na rozdíl od jiných jazyků, *indexy se píšou do kulatých závorek*. Není ovšem nutné začínat indexem 1.

Jestliže deklaraci zapíšeme takto

```
REAL, DIMENSION(-5:10) :: a
```

bude mít pole a celkem 16 prvků

```
a(-5), a(-4), ... , a(0), ... , a(10) .
```

Horní mez indexů musí být vždy uvedena. Není-li uvedena *spodní mez*, použije se *implicitní hodnota 1*.

Dvojměrné pole se deklaruje obdobně; např. matici 3×3 zavedeme příkazem

```
REAL, DIMENSION(3,3) :: A .
```

Rozsahy indexů pro jednotlivé dimenze se oddělují čárkou a jazyk dovoluje deklarovat maximálně 7 dimenzí. Pro určité operace je dobré vědět, že Fortran používá definované *uspořádání prvků pole* (často tomu odpovídá i pořadí uložení prvků pole v paměti) tak, že nejrychleji se mění první index, potom druhý atd. Pro naši matici A je to uspořádání po sloupcích

```
A(1,1), A(2,1), A(3,1), A(1,2), A(2,2), A(3,2), A(1,3), A(2,3), A(3,3) .
```

Indexem prvku může být libovolný *celočíselný výraz*. Jestliže jeho hodnota dá index ležící mimo deklarovaný rozsah, vznikne pochopitelně chyba. Ta bude ohlášena již při překladu (pokud je to možné v této fázi zjistit – lepší případ) nebo až během výpočtu.

2.2 Konstantní pole, počáteční hodnoty a funkce reshape

Podobně jako u skalárních proměnných, je možné i u polí deklarovat konstantní pole nebo přiřadit polí počáteční hodnoty. Jednodimensionální *konstantní pole* je seznam hodnot uzavřený mezi (/ a /), např.

```
REAL, DIMENSION(4), PARAMETER :: m=(/1.1, 8.3, 5.2, 3.0/)
```

Jde-li o pravidelnou posloupnost, např. (/1, 3, 5, 7, 9/) , dá se zapsat takto

```
(/(i,i=1,9,2)/) ! musi ovsem predchazet deklarace INTEGER :: i
```

Reálné pole (/1.1, 1.2, 1.3, 1.4/) je možné získat např. takto:

```
(/(i*0.1, i=11,14)/) ! je-li tretí položka rovna 1, nemusí se psat .
```

Vícerozměrná konstantní pole je možné získat z jednorozměrných polí pomocí zabudované funkce **reshape**, jejíž hlavička vypadá takto:

Tabulka 2.1: Zabudované funkce související s mezemi, tvarem a velikostí pole

<i>Funkce</i>	<i>Význam</i>
<code>lbound(pole[,dim])</code>	Bez <code>dim</code> vrací vektor spodních mezí pro všechny dimenze; je-li <code>dim</code> (celé číslo) uvedeno, vrací skalár – spodní mez pro zadanou dimenzi.
<code>shape(source)</code>	Vrací celočíselný vektor jehož prvky udávají velikost v příslušném směru (dimenzi); např. vektor $(/4, 3/)$ vrátí pro matici 4×3
<code>size(pole[,dim])</code>	Bez položky <code>dim</code> vrací celkový počet prvků pole. Se zadaným celočíselným <code>dim</code> vrací počet prvků ve směru <code>dim</code> .
<code>ubound(pole[,dim])</code>	Alternativa <code>lbound(pole[,dim])</code> pro horní meze polí.

`reshape(source, shape[,pad][,order])` .

Funkce transformuje pole `source` na pole s prvky stejného typu, jehož tvar (*shape*) je předepsán jednorozměrným celočíselným polem `shape`. Volitelná položka `pad` je jednorozměrné pole s prvky stejného typu jako `source`; prvky `pad` se doplní nové pole, jestliže počet prvků `source` byl menší (jestliže se prvky pole `pad` „vyčerpají“, začne se opět od prvního prvku). Nepovinná položka `order` je jednorozměrné pole stejného tvaru jako `shape`; určuje pořadí v němž se plní jednotlivé dimenze výstupního pole prvky pole `source` (bez jeho zadání se plní v pořadí $1, 2, \dots, n$, kde n je velikost pole `shape`). Nejlépe vše objasní příklad: mějme pole

`source=(/ 1,2,3,4,5,6 /)`, `shape=(/2,5/)`, `pad=(/0,1,2/)`.

Potom funkce `reshape` vrátí následující dvojrozměrná pole (matice) tvaru ¹⁾ 2×5 :

$$\text{reshape}(\text{source}, \text{shape}, \text{pad}) = \begin{pmatrix} 1 & 3 & 5 & 0 & 2 \\ 2 & 4 & 6 & 1 & 0 \end{pmatrix}$$

$$\text{reshape}(\text{source}, \text{shape}, \text{pad}, (/2,1/)) = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 0 & 1 & 2 & 0 \end{pmatrix} .$$

Pokud jde o fungování položky `order`, projevilo se zde výše zmíněné definované uspořádání prvků pole. V prvním případě se projevilo tím, že výsledné pole se plnilo tak, aby se nejrychleji měnil první index a potom druhý; pole se tedy plnilo fortransky definovaným způsobem, tj. „*po sloupcích*“ takto

$$r_{11} = 1, r_{21} = 2, r_{12} = 3, r_{22} = 4, r_{13} = 5, r_{23} = 6, r_{14} = 0, \dots, r_{25} = 0 .$$

Ve druhém případě se nejrychleji měnil druhý index a potom první, prvky nového pole se plnily „*po řádcích*“:

$$r_{11} = 1, r_{12} = 2, r_{13} = 3, r_{14} = 4, r_{15} = 5, r_{21} = 6, r_{22} = 0, \dots, r_{25} = 0 .$$

Z mnoha zabudovaných (intrinsicých) funkcí pro pole pokládám za vhodné uvést v Tab. ještě ty, které souvisí s dimenzí a tvarem pole.

DOHODA: pro stručnost vyjadřování bude v dalším textu nazývat jednorozměrná pole *vektory* a dvojrozměrná pole *matice*.

2.3 Subpole – výřezy z polí

Subpolem (*array section*) budeme rozumět ve všech směrech (dimenzích) souvislou část zadaného pole. Speciálním případem subpole je celé pole. Subpole je pole téhož typu (má prvky téhož typu) a dimenze jako mateřské pole.

¹⁾V algebře se říká *typu* $(2, 5)$ nebo 2×5 . Zde používám termín *tvar*, protože slovo *typ* spojujeme s typem prvků pole (říkáme, že pole je typu REAL jsou-li jeho prvky typu REAL a pod.).

Tabulka 2.2: Zápis subpolí v jednodimensionálním poli

Zápis subpole	Význam zápisu
vektor(:)	celé pole, totéž jako vektor
vektor(1:50)	celé pole, totéž jako vektor
vektor(10:19)	vektor tvořený deseti prvky $v(10), v(11), \dots, v(19)$
vektor(19:10:-1)	předchozí vektor s prvky v opačném pořadí, tj. $v(19), \dots, v(10)$
vektor((/3,6,40/))	vektor se třemi prvky $v(3), v(6), v(40)$
vektor(indexy)	vektor s pěti prvky, indexy jsou v celočíselném poli indexy

V tabulce se předpokládá deklarace

```
REAL, DIMENSION(50) :: vektor a
INTEGER, DIMENSION(5), PARAMETER :: indexy = (/ 2, 8, 32, 40, 49/)
```

Tabulka 2.3: Zápis subpolí v dvoudimensionálním poli

Zápis subpole	Význam zápisu
matice(:, :)	celé pole, totéž jako matice
matice(1:50, 1:50)	celé pole, totéž jako matice
matice(5, :)	pátý řádek matice (vektor), totéž jako matice(5, 1:50)
matice(:, 7)	sedmý sloupec matice (vektor), totéž jako matice(1:50, 7)
matice(5:10, 15:25)	submatice vyříznutá z řádků 5...10 a sloupců 15...25, tvar (shape) je 6×11
matice(2:50:2, 2:50:2)	matice 25×25 vytvořená z prvků na průsečících sudých sloupců a řádků
matice(indexy1, indexy2)	matice tvořená prvky na průsečících řádků z pole indexy1 a sloupců z indexy2

V tabulce se předpokládá deklarace

```
REAL, DIMENSION(50:50) :: matice
INTEGER, DIMENSION(m) :: indexy1 ! 1 <= m <= 50
INTEGER, DIMENSION(n) :: indexy2 ! 1 <= n <= 50
```

Uveďme příklady:

- v poli (vektoru) v deklarovaném s DIMENSION(6)

```
v(1) v(2) v(3) v(4) v(5) v(6)
```

je ohraničeno subpole (subvektor) $v(4:5)$.

- v matici A deklarované s DIMENSION(3, 4)

```
A(1,1) A(1,2) A(1,3) A(1,4)
```

```
A(2,1) A(2,2) A(2,3) A(2,4)
```

```
A(3,1) A(3,2) A(3,3) A(3,4)
```

je vyznačeno subpole (submatice) $A(1:2, 3)$.

V příkladech je již použit zápis subpolí; další možnosti jsou v tabulkách Tab. 2.2, Tab. 2.3.

2.4 Konformní pole, výrazy obsahující pole, konstrukce where

Dvě pole jsou konformní jestliže mají *stejný tvar*. Fortran95 dovoluje prakticky všechny operace mezi konformními poli a rovněž aplikaci zabudovaných funkcí pro skalární argumenty. Přitom

- unární operátory se aplikují na všechny prvky pole,
- binární operátory se aplikují mezi odpovídajícími si prvky obou polí,
- funkce se aplikují na všechny prvky pole.

Znovu zdůrazněme: *při všech operacích záleží na tvaru polí*, indexy prvků jsou nepodstatné.

Pro ilustraci předpokládejme, že máme deklarovaná pole

```
REAL, DIMENSION(4:5) :: A, B
LOGICAL, DIMENSION(4,5) :: L
REAL, DIMENSION(12) :: v
```

Potom můžeme psát např. příkazy

```
* B = -A           ! bude B(i,j) = -A(i,j)  ∀ i,j ,
* B = 0.2*sqrt(A)+1.0 ! B(i,j) = 0.2 * √A(i,j) + 1.0  ∀ i,j ,
* B(1:3, (/2,5/)) = A(/1,2,4/),1:2)+2*sin(B(2:4,1:2))
* L = B>=A        ! L(i,j) = T pro B(i,j) >= A(i,j) a F pro ostatní
* v(6:9) = B(1,:) + 0.1*B(2,:)
```

Zatím se příkazy prováděly se všemi prvky konformních polí. To je možné změnit pomocí příkazu **where**; příklad použití je v List. 2.1 (vzpomeňte na analogický příkaz **if**).

List. 2.1: Příklad použití příkazu where

```
WHERE (B>=A) B = 1.0 ! pro .true. nahradí prvek B 1.0, zbytek B se nemění
```

Konstrukce where je podobná konstrukci **if...else...end if**. Její jednoduché použití je zřejmé z příkladu v List. 2.2, který vytvoří matici B analogickou logické matici C: $B(i,j)=1.0$ tam kde $C(i,j)=T$ a $B(i,j)=0.0$ když $C(i,j)=F$. Přesnou specifikaci viz. [?, R739].

List. 2.2: Příklad konstrukce where .

```
WHERE (B>=A)
  B = 1.0 ! dosadí 1.0 do prvků B splňujících podmínku (viz. List. 2.1)
ELSEWHERE
  B = 0.0 ! dosadí 0.0 do prvků B nespňujících podmínku, tj. zbývajících
END WHERE
```

2.5 Alokovatelná pole

Pole která jsme zatím deklarovali jsou *statická*; vytvoří se při startu programu v paměti a existují dokud program neskončí. Navíc musíme při jejich deklaraci zadat požadovanou velikost, kterou nemusíme při startu programu znát (např. záleží na objemu dat načítaných ze souboru apod.). Ve Fortranu95 však existuje ještě jiná – dynamická – možnost deklarace pole: v deklarační části programu nebo modulu deklarujeme *alokovatelné pole* požadovaného typu a tvaru bez zadání jeho velikosti např. takto:

```
REAL, DIMENSION(:), ALLOCATABLE :: v           ! alokovatelné reálné 1D pole
COMPLEX, DIMENSION(:, :), ALLOCATABLE :: A    ! alokovatelné komplexní 2D pole .
```

Tím jsme dali kompilátoru na vědomí, že budeme s takovým polem pracovat, zavedli jsme pro něj identifikátor, ale zatím jsme ho nevytvořili a v paměti nezabírá žádné místo.

Pole vytvoříme v potřebné velikosti až ho bude program potřebovat. Provedeme to příkazem **ALLOCATE** podle těchto vzorů

```

ALLOCATE(v(n))          ! n je celé číslo
ALLOCATE(A(0:m,n))     ! m,n jsou celá čísla.

```

V argumentu ALLOCATE může být vše co již známe od deklarace statických polí (viz. 2.1).²⁾ S vytvořeným (alokovaným) polem pracujeme naprosto stejně jako s polem statickým.

Když program alokované pole již nepotřebuje, zrušíme ho (a uvolníme tak paměť) příkazem

```

DEALLOCATE(v)         ! nebo
DEALLOCATE(A)         !.

```

Alokace pole se ovšem nemusí povést (např. když v paměti – na haldě – již není požadované místo); totéž platí pro dealokaci pole (důvod je zde méně názorný a souvisí např. s dealokací *ukazatele*). Z to důvodu mají oba příkazy ještě druhý (volitelný) parametr STAT, který vrací celé číslo $i \geq 0$ (připomeňme analogii s IOSTAT v 1.7.2). Je-li $i = 0$, proběhla akce v pořádku, pro $i > 0$ došlo k chybě a akce se neprovede (příkazy bez parametru STAT by v tomto případě vedly k okamžitému ukončení programu). Je-li alokovatelné pole v danou chvíli alokované či nikoliv se zjistí pomocí *zabudované logické funkce* ALLOCATED. Příklad použití všech uvedených příkazů je krátkém programu v List. . Navíc je zde ještě ukázka použití alokovatelného pole v proceduře a funkci.

List. 2.3: Příklad použití alokovatelného pole.

```

1 PROGRAM T_alokpole
2 IMPLICIT NONE
3 REAL,DIMENSION(:),ALLOCATABLE :: v
4 INTEGER :: stav
5 CHARACTER(LEN=*),PARAMETER :: LF=char(10) ! konec řádku (pro PRINT)
6
7 PRINT *,"Je_v_alokovane_",allocated(v) ! zabudovaná funkce, zde vrací F
8 CALL alokuj_a_napl(v,4,stav)
9 IF (stav>0) THEN
10    PRINT *,"Alokace_v_se_nepovedla,_STAT=",stav,"Konec:_stiskni_ENTER"
11    READ * ! čeká na ENTER
12    STOP ! okamžitě ukončí program
13 END IF
14 IF (allocated(v)) PRINT *,"size(v)",SIZE(v),LF,"v=",LF,v,LF
15 DEALLOCATE(v) ! zruší pole
16 READ *
17
18 CONTAINS
19 SUBROUTINE alokuj_a_napl(pole1d,prvku,staterr)
20    REAL,DIMENSION(:),ALLOCATABLE,INTENT(INOUT) :: pole1d
21    INTEGER,INTENT(IN) :: prvku ! počet prvků pole
22    INTEGER,INTENT(OUT) :: staterr
23
24    ALLOCATE(pole1d(prvku),STAT=staterr)
25    IF (staterr==0) CALL random_number(pole1d)
26 END SUBROUTINE alokuj_a_napl
27
28 END PROGRAM T_alokpole

```

K programu v List. 2.3 připojme pro poučení ještě několik doplňujících poznámek:

²⁾Jen pro informaci uved' me, že dynamická pole se vytváří v určité části paměti nazývané *halda* (*heap*). Mechanismus práce s touto částí paměti dovoluje dynamicky na haldu přidávat a také z ní odebírat objekty s nimiž program pracuje.

- v řádku 5 jsme si zavedli znakovou konstantu LF pro řádkování v PRINT; připomeňme, že v deklaraci znakové konstanty musí být LEN=*,
- v řádku 12 jsme použili **příkaz STOP**, který způsobí *okamžité ukončení programu*. Použít se dá v hlavním programu i v procedurách (kromě funkcí a tzv. čistých – PURE – procedur). Je-li STOP použit ve více místech je žádoucí vědět, který příkaz program ukončil. To se dá snadno zařídit, neboť obecný tvar příkazu je

```
STOP [<vystupni_textovy_retezec>] .
```

Řádky 9-13 jsme mohli nahradit příkazem

```
IF (stav>0) STOP "Alokace_pole_v_se_nepodarila" .
```

Nevypsala by se však hodnota promenné stav a nebylo by možné pozastavit ukončení příkazem READ * (což nevádí, když program spouštíme v otevřeném CMD-okně, které zůstane otevřené i po ukončení programu).

- V řádku 25 jsme zavoláním zabudované procedury **random_number** naplnili alokované pole (pseudo)náhodnými čísly rovnoměrně rozdělenými na intervalu $[0, 1)$. Argumentem této procedury musí být skalár nebo pole typu REAL (i s KIND). K ní logicky patří ještě zabudovaná procedura random_seed uvedená v

2.6 Textové řetězce a znaková pole

V odst. 1.5 jsme se seznámili s textovými řetězci. Zde jen chci zdůraznit: *textový řetězec není textové (znakové) pole*. Znaky v textovém řetězci jsou subřetězce, nikoliv prvky znakového pole. Jestliže např. máme *textový řetězec* deklarovaný a naplněný takto

```
CHARACTER(LEN=10) :: Tretez
Tretez = "abcdef"
```

a chci vytisknout znak "c", musím zadat příkaz PRINT *, Tretez(3:3). Když naopak budu deklarovat *znakové pole* Tpole a naplním ho jako „znakovým vektorem“

```
CHARACTER(LEN=1), DIMENSION(6) :: Tpole
Tpole = (/ "a", "b", "c", "d", "e", "f" /) ,
```

mohu požadovat PRINT *, Tpole(3) a vytiskne se "c".

Další rozdíl: řetězec Tretez byl deklarován s LEN=10 a mohl jsem mu přiřadit řetěz délky 6; při přiřazení se totiž zbytek doplní znaky mezera a dojde k tomu i při přiřazení „prázdného řetězce“ příkazem Tretez="" . Snadno se o tom přesvědčíte např. pomocí nám již známé funkce ichar, která vrací kód znaku v argumentu (ichar(" ")=32). Abych mohl použít obdobný přiřazovací příkaz pro statické Tpole, musel jsem ho deklarovat s DIMENSION(6); pole na obou stranách přiřazovacího příkazu musí být konformní. Jiná situace však je u alokovatelných polí. Ověřte si, že můžete do programu zapsat

```
CHARACTER(LEN=1), DIMENSION(:), ALLOCATABLE :: Tpole
ALLOCATE(Tpole(10))
Tpole = (/ "a", "b", "c", "d", "e", "f" /)
```

a zjistěte, jaké znaky jsou např. v Tpole(9). Připomínám, že alokovatelná pole se při alokaci inicializují (všimli jste si toho u numerických polí?). Samozřejmě, že nemusíme znaková pole deklarovat jen s LEN=1 a jako jednodimenzionální. Řekli jsme hned v úvodu této kapitoly, že prvky polí mohou být proměnné libovolného typu. Mohou to tedy být např. textové řetězce délky n uspořádané do matice. S prvky takové matice (nebo jejími submaticemi) pak mohou provádět operace povolené pro textové řetězce (napište např. v předchozím příkladu příkaz PRINT *, ichar(Tpole)).

2.7 Další zabudované funkce pro pole

Funkce v Tab. 2.1 doplníme dalšími dostupnými funkcemi. Protože v jejich popisu se opakovaně vyskytují některé položky, zavedme si pro ně označení a objasněme si již zde jejich význam.

- i) *Dimenze pole (rank)* – budeme ji značit r – je dána počtem indexů, které určují prvek pole. Pro skalár je $r = 0$.
- ii) Volitelný celočíselný argument DIM určuje *směr* (index) v poli; $1 \leq \text{DIM} \leq r$ (např. pro matici je $r = 2$ a $\text{DIM} = \{1 | 2\}$).
- iii) Volitelný argument MASK je *logické pole* stejné dimenze a tvaru jako POLE (jestliže je mezi argumenty).

K volitelným parametrům je třeba doplnit další informaci (více viz. 5). Při volání procedury je význam parametrů zadán jejich pořadím. Jestliže ale existuje několik volitelných parametrů a některý chci vynechat, nemuselo by být jasné o který parametr jde. To se řeší tak, že se parametr uvede se jménem, které dostal při deklaraci. Např. když pro dále uvedenou funkci EOSHIFT vynechám parametr BOUNDARY, ale chci zadat DIM, bude zápis vypadat takto

EOSHIFT(pole, posun, DIM=2) ! pole a posun jsou povinné parametry

Z tohoto důvodu jsou u následujících funkcí uváděny volitelné parametry s deklarovanými jmény (povinné parametry obecně nikoliv).

ALL(MASK[, DIM])

vrací .true. jsou-li všechny prvky MASK (ve směru DIM, je-li přítomen) rovny .true..

ANY(MASK[, DIM])

vrací .true. je-li libovolný prvek MASK (ve směru DIM, je-li přítomen) roven .true..

COUNT(MASK[, DIM])

vrací počet prvků s hodnotou .true. v MASK (ve směru DIM, je-li přítomen).

CSHIFT(POLE, POSUN[, DIM]) – cirkulární posun prvků POLE

vrací pole stejného typu, dimenze a tvaru jako je POLE. Celočíselný argument POSUN musí být skalárem, je-li POLE jednodimensionální. Jestliže argument DIM není uveden, je to totéž jako DIM=1. *Pro skalární POSUN se všechna 1D subpole POLE ve směru DIM posunou o POSUN pozic. Pro POSUN>0 probíhá posun vlevo (směrem k menším indexům) a pro POSUN<0 vpravo (směrem k větším indexům). Uvedme příklady:*

$$\text{cshift}([1, 2, 3, 4, 5], 2) \underset{\leftarrow{2 \times}}{=} [3, 4, 5, 1, 2], \quad \text{cshift}([1, 2, 3, 4, 5], -2) \underset{\rightarrow{2 \times}}{=} [4, 5, 1, 2, 3]$$

$$\text{cshift}\left(\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}, 2\right) \underset{\uparrow{2 \times}}{=} \begin{bmatrix} 9 & 10 & 11 & 12 \\ 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix},$$

$$\text{cshift}\left(\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}, -2\right) \underset{\downarrow{2 \times}}{=} \begin{bmatrix} 9 & 10 & 11 & 12 \\ 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

$$\text{cshift}\left(\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}, 1, 2\right) \underset{\leftarrow{1 \times}}{=} \begin{bmatrix} 2 & 3 & 4 & 1 \\ 6 & 7 & 8 & 5 \\ 10 & 11 & 12 & 9 \end{bmatrix}$$

$$\text{cshift}\left(\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}, -1, 2\right) \underset{\rightarrow{1 \times}}{=} \begin{bmatrix} 4 & 1 & 2 & 3 \\ 8 & 5 & 6 & 7 \\ 12 & 9 & 10 & 11 \end{bmatrix}$$

Je-li POSUN *pole*, musí mít stejný tvar jako příslušné subpole argumentu POLE (pro odpovídající DIM) a prvky pole značí přímo posuny. Nejlépe funkce vysvitne z příkladu:

$$\text{cshift}\left(\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}, (/0,1,-1,-2/)) = \begin{bmatrix} 1 & 6 & 11 & 8 \\ 5 & 10 & 3 & 12 \\ 9 & 2 & 7 & 4 \end{bmatrix},$$

$$\text{cshift}\left(\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}, (/ -1,0,1/), 2) = \begin{bmatrix} 4 & 1 & 2 & 3 \\ 5 & 6 & 7 & 8 \\ 10 & 11 & 12 & 9 \end{bmatrix}.$$

EOSHIFT(POLE, POSUN[, BOUNDARY][, DIM])

pracuje stejně jako cirkulární CSHIFT, pokud jde o argumenty POLE, POSUN, DIM. Nedochozí však k cirkulárnímu přenosu a uvolněné pozice se vyplňují podle argumentu HRANICE takto:

- argument BOUNDARY není uveden, prvky se vyplní 0 (totéž jako BOUNDARY= 0),
- argument BOUNDARY je skalár téhož typu jako POLE, prvky se vyplní tímto skalárem,
- BOUNDARY je vektor s prvky téhož typu jako POLE a jeho dimenze a tvar splňuje tytéž podmínky jako pole POSUN.

Rozdíl proti CSHIFT jasně uvidíte, když zopakujete předchozí příklady s EOSHIFT a doplněným argumentem BOUNDARY.

MAXLOC(POLE[, MASK])

vrací vektor indexů určujících polohu maximálního prvku POLE; je-li uvedeno pole MASK, hledá se jen mezi prvky odpovídající .true..

MAXLOC(POLE, DIM[, MASK])

vrací celočíselné pole indexů maximalních prvků ve směru DIM.

Příklad (s deklarací LOGICAL, PARAMETER :: T=.true. , F=.false.):

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}, \quad \text{MASK} = \begin{bmatrix} T & T & T & F \\ F & T & F & T \\ F & T & T & F \end{bmatrix}$$

PRINT *, <prikaz>	Vypíše	Komentář
MAXLOC(A)	3 4	maximální prvek je A ₃₄
MAXLOC(A, MASK)	3 3	maximální prvek je A ₃₃ (v polohách T v MASK)
MAXLOC(A, 1)	3 3 3 3	indexy řádků s max. prvkem ve sloupcích; A ₃₁ , A ₃₂ , A ₃₃ , A ₃₄
MAXLOC(A, 1, MASK)	1 3 3 2	předchozí případ s MASK: A ₁₁ , A ₃₂ , A ₃₃ , A ₂₄
MAXLOC(A, 2)	4 4 4	indexy sloupců s max. prvkem v řádcích 1-3: A ₁₄ , A ₂₄ , A ₃₄
MAXLOC(A, 2, MASK)	3 4 3	předchozí případ s MASK: A ₁₃ , A ₂₄ , A ₃₃

MINLOC(POLE[, MASK])

MINLOC(POLE, DIM[, MASK])

jsou identické funkcím MAXLOC, jen s tím rozdílem, že se týkají minimálních prvků.

MAXVAL(POLE[, DIM])

MAXVAL(POLE, DIM[, MASK])

MINVAL(POLE[, DIM])

MINVAL(POLE, DIM[, MASK])

vybírají prvky stejně jako MAXLOC a MINLOC, vrací však hodnoty v nich uložené. Jsou proto použitelné jen pro celočíselná a reálná pole. Pokud je velikost pole rovna 0 (např. A(2:1, 1), což se může stát třeba

v cyklech a *nebude hlášena chyba*), vrací MAXVAL zápornou a MINVAL kladnou hodnotu největšího čísla pro typ prvků pole (viz. funkci HUGE v dod. ??).

MERGE(Tpole, Fpole, MASK)

Tpole je pole libovolného typu a Fpole je konformní pole téhož typu. Funkce vrací pole, které je konformní s Tpole a je stejného typu. Vytvoří se tak, že se jeho prvky berou z Tpole tam kde odpovídající prvky MASK jsou .true. a z Fpole tam kde MASK má .false.. MASK může být též *skalár*; funkce potom vrátí Tpole pro .true. a Fpole pro .false..

PRODUCT(POLE[, MASK])

bez MASK vrací součin všech prvků pole, s konformním MASK pouze součin těch prvků pro něž jsou odpovídající prvky .true.. Použitelná je pouze pro celočíselná, reálná a komplexní POLE. Je-li velikost POLE rovna 0 nebo všechny prvky MASK jsou .false., vrací 1.

PRODUCT(POLE, DIM[, MASK])

chová se stejně jako předchozí, ale pracuje ve směru DIM. To znamená, že např. pro matici vrací vektor součinů prvků ve sloupcích (DIM=1) nebo v řádcích (DIM=2). S výše uvedenými maticemi A a MASK

PRINT *, PRODUCT(A, 1, MASK) vypíše 1.0000000 1.2000000E02 33.0000000 8.0000000

PRINT *, PRODUCT(A, 2, MASK) vypíše 6.0000000 48.0000000 1.1000000E02.

SUM(POLE[, MASK])

SUM(POLE, DIM[, MASK])

pracují stejně jako PRODUCT s tím, že součiny jsou nahrazeny součty příslušných prvků a pro SIZE(POLE)=0 vrací 0.

SPREAD(POLE, DIM, NCOPIES)

POLE je skalár nebo pole libovolného typu s dimenzí $0 \leq r < 7$ ($r = 0$ pro skalár). Funkce vytvoří pole s dimenzí $r + 1$ tak, že kopíruje POLE ve směru DIM; počet kopií je NCOPIES a pro DIM musí platit $1 \leq \text{DIM} \leq r + 1$. Pro NCOPIES ≤ 0 se vytvoří pole velikosti 0. Je-li POLE skalár, vytvoří se vektor (1Dpole) s NCOPIES tohoto skaláru. Je-li POLE skutečně pole ($r > 0$), potom prvek výsledného pole s indexy $(s_1, s_2, \dots, s_{n+1})$ má hodnotu prvku $\text{POLE}_{s_1 \dots s_{DIM-1} s_{DIM+1} \dots s_{n+1}}$.

Příklady:

SPREAD(2, DIM=1, NCOPIES=3) vrací (/2, 2, 2/) (2 je skalár s $r = 0$, vrací pole s $r = 1$)
Necht' $v = (/1, 2, 3/)$ ($r = 1$, možné hodnoty DIM jsou 1, 2). Vytvořená pole mají $r = 2$ a jsou

$$\text{spread}(v, 1, 2) = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}, \quad \text{spread}(v, 2, 2) = \begin{bmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}$$

Závěrem uved' me ještě tři funkce pro běžné operace s vektory ($r = 1$) a maticemi ($r = 2$).

TRANSPPOSE(A)

provádí transpozici matice A libovolného typu. Má-li A tvar $(m \times n)$, potom transponová A^T má tvar $(n \times m)$ a prvky $A_{ij}^T = A_{ji}$.

DOT_PRODUCT(x, y) – skalární součin vektorů

kde x, y jsou logické, celočíselné, reálné nebo komplexní *vektory* ($r = 1$) stejné velikosti. Možnosti:

- Pro x celočíselný nebo reálný vrací sum(x*y).
- Pro x komplexní vrací sum(conjg(x)*y) (conjg vrací komplexně sdružený vektor).
- Pro logické vektory x, y vrací any(x .and. y).

MATMUL(A, B) – maticové násobení

MATMUL(v, B)

MATMUL(A, v)

Tabulka 2.4: Řídící znaky, které lze vkládat do formátovacího řetězce

Znak	Funkce
[n]/	Přechod na nový řádek, volitelné $n \geq 1$ znamená počet opakování.
Tn, TRn, TLn	n je přirozené číslo. Provede následující výstup na pozici n (Tn), posunutý o n pozic vpravo (TRn) nebo vlevo (TLn). Pro levý okraj je $n=1$.
SP, SS, S	SP nastaví výstup (<i>sign print</i>) znaménka +, SS potlačí (<i>sign suppress</i>) výstup +, S nastaví standardní (<i>default</i>) stav. Nastavení SP nebo SS funguje (pokud není změněno) do ukončení výpisu.
:	Ukončí formátovací řetězec je-li v daném místě již vyčerpán seznam vypisovaných položek.

kde A, B jsou matice ($r = 2$) a v je vektor ($r = 1$). Je-li A je matice tvaru ($k \times m$) a B má tvar ($m \times n$), pak vektor v musí mít tvar (m) neboť se *chápe* jako matice tvaru ($1 \times m$) pro $\text{MATMUL}(v, B)$ a matice ($m \times 1$) pro $\text{MATMUL}(A, v)$. Pozor: v posledních dvou případech je výsledkem *vektor*. Jestliže však místo v dosadíme skutečnou matici ($1 \times m$), resp. ($m \times 1$), bude výsledkem *matice*.

Pro celočíselné, reálné nebo komplexní matice (vektory) je

- prvek (i, j) $\text{MATMUL}(A, B)$ ekvivalentní $\text{sum}(A(i, :)*B(:, j))$,
- prvek (j) $\text{MATMUL}(v, B)$ ekvivalentní $\text{sum}(v*B(:, j))$,
- prvek (i) $\text{MATMUL}(A, v)$ ekvivalentní $\text{sum}(A(i, :)*v)$.

Pro logické matice (vektory) je tvar výsledků stejný, pouze se nahradí sum a $*$ v uvedených výrazech any a and .

2.8 Možnosti příkazů PRINT, WRITE (nejen) pro výstup polí

Pro výpisy při experimentování s vektory a maticemi je vhodné již zde prozradit některé další možnosti příkazu PRINT, resp. WRITE (podrobnosti jsou v kap. 4). Pro další výklad předpokládejme deklarace

```
{INTEGER|REAL|COMPLEX}, DIMENSION(m) :: v      ! numerický vektor tvaru m
{INTEGER|REAL|COMPLEX}, DIMENSION(m,n) :: A    ! numerická matice tvaru m x n
```

Snadno zjistíte, že je možné napsat legální příkaz $\text{PRINT } *, v$ nebo $\text{PRINT } *, A$. Výstup vás však asi příliš neuspokojí. Oba vypíše všechny prvky pole ve volném formátu (*) v „plné přesnosti“, což je většinou zbytečně dlouhý a nepřehledný tvar. Prvky *matice* se navíc vypisují v definovaném pořadí, tj. *po sloupcích*, takže výsledek vůbec nepřipomíná běžný zápis matice. Předepsat formát výstupu pomocí formátovacího řetězce však umíme z odst. 1.8, 1.9.

Protože stručná informace o položkách formátovacích řetězců v Tab. ?? není vyčerpávající, uved' me zde nejprve další možnosti:

- i) Přirozené číslo n vyznačující počet opakování položky (např. $nfw.d$) je možné předřadit i skupině formátovacích položek uzavřené v závorkách. Tak např. $2(3f6.1, 2(2i3, a))$ je ekvivalentní $3f6.1, 2i3, a, 2i3, a, 3f6.1, 2i3, a,$.
- ii) Je-li počet položek ve formátovacím řetězci menší než počet položek v seznamu pro výpis, začne se *formát používat znovu* od začátku s tím, že *napřed přejde na nový řádek*.
- iii) Přímo do formátovacího řetězce lze ještě vkládat několik řídicích znaků podle Tab. 2.4, které ovlivňují pozici výstupu.

S těmito vědomostmi se již dá výrazně zlepšit kvalita výstupu. Např. s využitím poznatku pod bodem ii) snadno vypíšeme:

- *vektor jako sloupec* příkazem $\text{PRINT } "(f6.1)", v$,

– matici po řádcích příkazem `PRINT "(nf6.1)", transpose(A) ! A^T má tvar $n \times m$.`

Zatím jsme využili vlastnosti formátu. Je však možné také modifikovat seznam vystupujících položek. Víme, že i -tý řádek matice se dá zapsat `A(i, :)`. S našimi dosavadními znalostmi bychom mohli naši matici vypsát tímto úsekem programu:

```
INTEGER :: i,m    ! A je již deklarované
m = SIZE(A,1)    ! m je počet řádků A
i = 0
DO
  i = i+1
  IF (i>m) EXIT
  PRINT "(nf6.1)",A(i,:) ! n je konstanta, počet sloupců A
END DO
```

V následující kapitole se naučíme variantu DO-konstrukce (ekvivalentní příkazu FOR v jiných jazycích), která dovolí následující zápis

```
INTEGER :: i    ! A je již deklarované
DO i = 1,SIZE(A,1)
  PRINT "(nf6.1)",A(i,:) ! n je konstanta, počet sloupců A
END DO
```

Fortran95 dovoluje ještě úspornější zápis – uvést cyklus jako položku výstupního seznamu

```
INTEGER :: i    ! A je již deklarované
PRINT "(nf6.1)", (A(i,:),i=1,SIZE(A,1)) ! n je konstanta, počet sloupců A
```

Obecný tvar seznamu výstupních položek má tvar

List. 2.4: Seznam výstupních položek s cyklem

```
( seznam_DO_polozek, DO_prom = expr1, expr2 [,expr3] )
```

kde seznam_DO_polozek je seznam výrazů závislých na DO_prom; může to být opět seznam s cyklem, takže je např. možný příkaz `PRINT "(2f6.1)", ((A(i, j), j=1,4,2), i=1,3) .`

expr1, expr2, expr3 jsou celočíselné výrazy, volitelný expr3 udává krok cyklu. V našem příkladu by

- `PRINT "(nf6.1)", (A(i, :), i=1, SIZE(A,1), 2)` vypsál jen liché řádky matice,
- `PRINT "(nf6.1)", (A(i, :), i=SIZE(A,1), 1, -1)` vypsál řádky v opačném pořadí.

V tuto chvíli se přirozeně nabízí myšlenka *napsat proceduru*, které provede výpis když jí zadám matici a požadovaný formát. Téměř vše potřebné pro splnění zadání už umíme, zbývá vyřešit jen jeden problém – jak programově dostat konstantu $n=size(A,2)$ do formátovacího řetězce. Fortran95 k tomu nabízí elegantní možnost. V odst. 1.8 jsme se krátce seznámili s příkazem `write` a použili jsme ho v proceduře `CtiCislo` deklarované v List. 1.11. Víme, že položka `UNIT` určuje výstupní zařízení. Pro nás je teď důležité, že **výstupním zařízením může být textový řetězec** (někdy se mu říká *vnitřní soubor*). Vytoužený formátovací řetězec mohou tudíž vytvořit úsekem programu podle List. 2.5

List. 2.5: Zápis do textového řetězce příkazem write

```
CHARACTER(LEN=20) :: buf    ! matice A je již deklarovaná
CHARACTER(LEN=10) :: forma ! formát čísel, bude parametr procedury
FORMAT="f7.2"              ! zde jen jako příklad

WRITE(UNIT=buf, FMT="(a, i3, a)") (" ", SIZE(A,2), FORMAT//)"
```

S takto vytvořeným formátovacím řetězcem bych již mohl psát

```
PRINT buf, (A(i, :), i=1, SIZE(A,1))
```

Uvědomte si, že `buf` je deklarovaný jako textový řetězec a *není* proto uzavřen do uvozovek! Nyní už můžeme bez problémů napsat požadovanou proceduru `PisMatici_R`; najdete ji v List. 2.6. Pojmenování není náhodné, protože bude asi vhodné vytvořit ještě `PisMatici_I` (případně pro další typy), shrnout je do modulu a zavést generické jméno `PisMatici` tak jak jsme to udělali v List. 1.16. Procedura dobře poslouží pro výpis matic, jejichž řádek se vejde na řádek obrazovky. Jako cvičení zkuste další vylepšení, např. místo formátu zadat počet pozic pro výpis prvku pole a podle velikosti absolutní hodnoty čísla, použít formát s pevnou desetinnou tečkou nebo exponenciální tvar. *Komplexní čísla* se vypisují jako dvojice reálných čísel a pro jejich výpis je potřeba zadat zvlášť *formát pro reálnou a imaginární část* (nemusí být stejný). Ve volném formátu (*) se vypisují jako dvojice uzavřená v závorkách; naprogramujte výstup v obvyklém tvaru $\mathcal{R} + i\mathcal{I}$.

List. 2.6: Procedura pro výpis reálné matice na displej

```

SUBROUTINE CtiMatici_R(hlavicka,matice,forma)
  IMPLICIT NONE
  CHARACTER(LEN=*),INTENT(IN) :: hlavicka,forma
      ! text na displeji pred vystupem matice a format pro vystup cisla
  REAL,DIMENSION(:,:),INTENT(IN) :: matice ! vypisovana matice

  INTEGER :: i
  CHARACTER(LEN=20) :: buf
  WRITE(UNIT=buf,FMT="(a,i3,a)") "(",SIZE(matice,2),forma//")"
  PRINT *,hlavicka
  PRINT buf,(matice(i,:),i=1,SIZE(matice,1))
END SUBROUTINE CtiMatici_R

```

3 Řídící konstrukce

3.1 Rozhodovací konstrukce IF

Konstrukce kterou známe z odstavce 1.6.3 může mít ještě obecnější tvar podle List. 3.1.

List. 3.1: Obecný tvar IF konstrukce

```
IF (<skalarni_logicky_vyraz>) THEN
    <blok_prikazu>
[ELSE IF (<skalarni_logicky_vyraz>) THEN ! může se
    <blok_prikazu> ! opakovat několikrát
[ELSE
    <blok_prikazu>]
END IF
```

Nově se zde objevuje vložená volitelná část ELSE IF (lze psát i ELSEIF), která se může i několikrát opakovat. Podmínky (logické výrazy) v posloupnosti příkazů ELSEIF se mohou překrývat. Při běhu programu se vyhodnocují ve vypsáném pořadí a jakmile je některá podmínka (logický_vyraz) splněna, provede se následující blok_prikazu a řízení přejde na příkaz následující za IF-konstrukcí (za END IF); následující podmínky se již nevyhodnocují. Libovolný počet IF-konstrukcí je možné do sebe vkládat, jako např. v List. 3.2. I když tento program nevykonává příliš inteligentní činnost, všimněte si, kromě vnořování IF-konstrukcí, ještě několika detailů.

- Pro přehlednost a snadnou kontrolu je žádoucí udržovat určitou úpravu zápisu programu, zejména:
 - (a) odsazovat vnořené konstrukce,
 - (b) psát prvky konstrukcí (např. IF . . . ELSE IF . . . ELSE . . . END IF , DO . . . END DO) pod sebe. Zvykněte si na určitý styl a dodržujte ho. Některé editory, např. nabízený SciTe, tomu velice napomáhají.
- Podmínka ($r==0.0$) v řádku 9 bude jistě splněna při zadání 0 z klávesnice (nula se zobrazuje přesně). Snadno však zjistíte, že bude splněna i tehdy, když zadáte dostatečně malé číslo; dokážete to vysvětlit? Pomoci může dod. ???. Abychom tyto dvě možnosti rozlišili, bylo by možné psát za sebou dvě (překrývající se) podmínky

```
ELSEIF (r==0) THEN
    . . .
ELSEIF (r<rmin) THEN
    . . .
```
- V řádcích 14, 16 je pro připomenutí a ilustraci trochu obohacený formátovací řetězec (Tab. 2.4). Zkuste ještě zaměnit v řádku 16 specifikátor es za en a měnit počet pozic pro exponent.

3.2 Konstrukce CASE

Konstrukce CASE poskytuje další cestu k výběru z více možností. Její obecná struktura je v List. 3.3. Hodnota položky <vyraz> musí být skalár typu INTEGER nebo CHARACTER (libovolné délky LEN). Hodnota položek <selektor> musí být stejného typu jako <vyraz>, délka textových řetězců však může

List. 3.2: Příklad vložených IF-konstrukcí

```

1 PROGRAM T_if
2 REAL :: r
3
4 DO
5   PRINT *, "Zadej_realne_cislo_r,_pro_r=0_KONEC_programu"
6   READ *, r
7   IF (r<0.0) THEN
8     PRINT *, r, "je_zaporne"
9   ELSE IF (r==0.0) THEN
10    PRINT *, "r=0.0"
11    EXIT
12  ELSE ! r>0
13    IF (r<1000.0 .and. r>0.001) THEN
14      PRINT "(a,sp,f6.3)", "r=", r
15    ELSE
16      PRINT "(a,sp,es12.3e2)", "r=", rJaké jsou
17    END IF
18  END IF
19 END DO
20 READ * ! ceka na Enter
21 END PROGRAM T_if

```

být různá. CASE-konstrukce pracuje tak, že se vyhodnotí výraz a provede se <blok_prikazu> za selektorem, kterému hodnota výrazu vyhovuje. Selektor je obecně *seznam nepřekrývajících se konstant a intervalů*. Interval je *spodni_mez:horni_mez*, přičemž jedna z mezí může chybět; chybějící (spodní/horní) mez znamená všechny možné hodnoty až (do/od) (horní/spodní) meze.

List. 3.3: Obecný tvar CASE-konstrukce

```

SELECT CASE (<vyraz>)
  CASE (<selektor>) ! může se
    <blok_prikazu> ! opakovat vícekrát
  :
  [CASE DEFAULT
    <blok_prikazu>]
END SELECT

```

Příklady selektorů:

```

(1,2,4,12) ! výraz musí být některá z uvedených hodnot
("a","n","A","N") ! výraz musí být některá z uvedených hodnot
(:-1,5,10:15) ! hodnota výrazu musí být: {≤ 0|5|10|11|12|13|14|15}
("j":"m", "r":"z") ! hodnota výrazu musí být: {"j"|"k"|"l"|"m"|"r"|. . .|"z"}

```

Pokud jde o *znakové selektory* porovnávají se vlastně celočíselné kódy znaků. Ty jsou jednoznačně dané pro ANSI tabulku s kódy 0 – 127 (anglická abeceda, 7 bitový kód). Tisknutelné znaky začínají

dekadickým kódem 32(mezera); kódy alfanumerických znaků jsou potom 48("0")–57("9"), 65("A")–90("Z"), 97("a")–122("z"). Zbývající znaky ASCII tabulky s kódy 128 – 255 jsou sice v řetězcích použitelné, ale jak samotné znaky tak i uspořádání závisí na nastavené kódové stránce (příkaz `chcp` v CMD-okně). V tuto chvíli si jistě už každý dokáže vypsát kódy a přiřazené znaky příkazem `PRINT "(i4,a4)",(i,char(i),i=32,255)`. Znakové výrazy i selektory mohou být samozřejmě víceznakové, tj. *textové řetězce*. Při rozhodování potom nastupuje známé *lexikografické uspořádání*. Jestliže nemají výraz a selektor stejnou délku, doplní se kratší mezerami.

Závěrem ještě důležité upozornění. Zatímco v posloupnosti příkazů `ELSE IF` se mohly podmínky překrývat, *selektory v konstrukci CASE musí být jedinečné*. Jestliže výraz nevyhovuje žádnému selektoru a existuje volitelná část `CASE DEFAULT`, provede se blok příkazů které za ní následují; v opačném případě se neprovede nic a `CASE`-konstrukce končí. Malý ilustrační příklad je v List. 3.4.

List. 3.4: Příklad na užití konstrukce CASE

```
PROGRAM Mesice

CHARACTER(LEN=*),DIMENSION(12),PARAMETER :: mesic= &
  ("Leden_","Unor_","Brezen_","Duben_","Kveten_","Cerven_"," &
  "Cervenec","Srpen_","Zari_","Rijen_","Listopad","Prosinec"/)
INTEGER :: i

DO
  WRITE(UNIT=*,FMT="(a)",ADVANCE="no")"Zadej_cislo_mesice_"
  READ *,i
  IF (i>=1 .and. i<=12) EXIT
  PRINT *, "Cislo_musi_byt_z_[1,12],_Zadej_znovu!"
END DO
SELECT CASE (i)
  CASE (1,3,5,7,8,10,12)
    PRINT *,trim(Mesic(i))//"_ma_31_dnu"
  CASE (2)
    PRINT *,trim(Mesic(i))//"_ma_28,_prestupny_29_dnu"
  CASE (4,6,9,11)
    PRINT *,trim(Mesic(i))//"_ma_30_dnu"
  END SELECT
READ *      ! ceka na Enter
END PROGRAM Mesice
```

V programu List. 3.4 si kromě zopakování několika již známých věcí všimněte:

- V deklaraci konstanty `mesic` (pole textových řetězců) je nutné aby všechny prvky přiřazovaného pole měly stejnou délku. Proto jsou některé z nich doplněny mezerami na maximální délku 8. Připomeňme: kdybych deklaroval

```
CHARACTER(LEN=8),DIMENSION(12) :: mesic ,
```

potom mohu psát přiřazovací příkazy jako např.

```
mesic(1) = "Leden"
mesic(12) = "Prosinec"
```

a potřebné mezery se doplní automaticky.

- Protože jsme ve vstupním cyklu `DO` zkontrolovali zadávané číslo, mohli jsme místo `CASE (4,6,9,11)` použít `CASE DEFAULT`.

– Kontrolu zadávaného čísla jsme mohli zabudovat jako jednu z položek konstrukce CASE (proved'te).

3.3 Cykly – konstrukce DO

Konstrukci DO zatím známe jen v jednoduché podobě podle List. 1.7 (a částečně z List. ??). Její obecný tvar je dán List. 3.5. Nové jsou zde dvě věci:

- i) Konstrukce DO může mít <jmeno>, což je nějaký identifikátor. Je-li uvedeno před DO musí být uvedeno i za END DO. Jaké jsou možnosti využití pojmenování konstrukce DO uvidíme dále.
- ii) Volitelná položka <řízení_cyklu> má tvar

DO_prom = vyraz1,vyraz2[,vyraz3] ! význam: DO_prom = od,do[,krok] ,

kde DO_prom je celočíselná proměnná deklarovaná v programu nebo v proceduře která konstrukci DO používá. Tato proměnná nesmí být: prvkem pole, položkou uživatelem definovaného typu (viz. 1.10) a formálním parametrem procedury. Její hodnota se nesmí v konstrukci DO měnit (nesmí být na levé straně přiřazovacího příkazu). Konstrukce DO s <řízení_cyklu> je obdobou cyklů FOR v jiných jazycích.

Podívejme se teď na některé možnosti DO konstrukce podrobněji.

List. 3.5: Obecný tvar konstrukce DO

```
[<jmeno>:] DO [<řízení_cyklu>
      <blok_prikazu>
      END DO [<jmeno>]
```

3.3.1 Řízení cyklu, příkazy exit a cycle

Základní informace o struktuře <řízení_cyklu> jsme uvedli výše. Položky vyraz1, vyraz2 a vyraz3 jsou libovolné celočíselné výrazy. Volitelná položka vyraz3, pokud je uvedena, musí být nenulová. Význam všech tří výrazů je zřejmý: cyklus se má provádět s proměnnou DO_prom měnící se od hodnoty vyraz1 do vyraz2 s krokem vyraz3. Je-li vyraz3 = 1, nemusí se uvádět. Celkový počet provedených cyklů bude

$$\max\left(\frac{\text{vyraz2} - \text{vyraz1} + \text{vyraz3}}{\text{vyraz3}}, 0\right),$$

kde funkce max, vrací maximální hodnotu ze seznamu argumentů. Z toho je zřejmé, že *cyklus nemusí být proveden ani jednou*. To se stane např. když v příkazu

DO i = 1,m

bude $m \leq 0$. Další příklady řídicích konstrukcí:

DO i = 1,9,2 ! i postupně bude 1,3,5,7,9

DO i = 9,0,-1 ! i postupně bude 9,8,...,1,0

DO i = 2*j,m,n(j)-2 ! i podle hodnot j,m,n(j) na počátku cyklu

Z List. 1.7 již víme, že z cyklu můžeme vyskočit příkazem **exit**; nyní navíc víme, že lze k němu přidat jméno cyklu, takže může vypadat např. takto:

prvni: DO j = 1,n

...

IF (j==k) EXIT prvni

```

...
END DO první
m = j

```

Pokud jde o jméno (první) má zde pouze ilustrativní funkci. Jaká však bude hodnota m po provedení prvního příkazu za cyklem? Z následujících tří možností se více poučíme i o realizaci cyklů s řídicí konstrukcí:

- Necht' při zahájení cyklu je $n \leq 0$; proměnné j se přiřadí hodnota 1, cyklus vůbec neproběhne a provede se první příkaz za cyklem (m bude rovno 1).
- Pro $n \geq k$ dojde ke splnění podmínky ($j==k$), cyklus bude ukončen příkazem `exit` a m bude rovno k .
- Pro $n < k$ proběhne všech n cyklů, na konci n -tého se j zvětší na hodnotu $n+1$ s níž už cyklus neproběhne a m se přiřadí $n+1$.

Další příkaz použitelný v těle cyklu je

```
CYCLE [<jmeno>] ,
```

kteřý předá řízení na na konec odpovídající DO konstrukce (se jménem `<jmeno>` je-li pojmenovaná). Jinými slovy: neprovedou se příkazy následující za `cycle`, program skočí na `END DO [<jmeno>]` a začne další iteraci cyklu.

3.3.2 Příkaz `go to`, vložené řídicí konstrukce

Příkaz `go to <navesti>` (lze psát i `goto`¹⁾) patří dlouhá léta k nejvíce diskutovaným příkazům v programovacích jazycích. Pravda je, že jeho bohaté používání, konkrétně v rozsáhlejších programech při skocích zpět, vedlo k tomu, že programy byly velice těžko čitelné a tím i kontrolovatelné. Pravdou také je, že moderní jazyky poskytují dostatek řídicích konstrukcí aby se bez skoků `goto` mohly obejít. Ve vyjimečných případech však může rozumné použití `goto` naopak program zpřehlednit. Jazyk F proto jeho použití ve velmi omezené míře připouští.

Činnost příkazu `goto` je snadno pochopitelná: předá řízení na příkaz který je označený návěštím (*label*) `<navesti>`. *Návěští* je posloupnost maximálně pěti cifer

```
<navesti> = cifra[cifra[cifra[cifra[cifra]]] ,
```

z nichž alespoň jedna musí být různá od nuly. V jazyce F platí tato omezení:

- i) Jediný příkaz na který lze pomocí `goto` přejít (skočit) je příkaz

```
<navesti> CONTINUE ,
```

Tento příkaz „nedělá nic“, pouze je možné k němu připsat návěští a umožnit tak skok.

- ii) Nejsou povoleny skoky zpět, tj. na řádky předcházející `goto`.

Snadno pochopitelné také je, že nejen ve Fortranu95 (a tedy i v F), *nemůže být povoleno* skákat z vnějšku konstrukcí (IF, CASE, DO, WHERE, FORALL) na příkaz CONTINUE v jejich těle (uvnitř konstrukce).

Příklad rozumného použití příkazu GOTO je v úseku programu podle List. 3.6. Jistě ho už dokážete napsat bez použití GOTO, ale asi stěží bude výsledek přehlednější.

V každé z probíraných konstrukcí byl `<blok_prikazu>` jehož součástí může samozřejmě být zase kterákoliv konstrukce. Základní pravidlo však je, že vložená *konstrukce musí být celá uvnitř bloku*. Není proto možné vytvořit např. něco takového jako v List 3.7.

Víme již, že konstrukce DO může mít jméno (viz. List.3.5) a toto jméno může také následovat za příkazy CYCLE a EXIT. Použitá jména plní dvojí funkci:

¹⁾V F je možné všechny příkazy sestávající ze dvou slov oddělených mezerou (END IF, END DO, SELECT CASE apod.) psát bez mezery (viz. [?, R304]).

List. 3.6: Příklad rozumného použití GOTO

```

    ! hledani zadaneho textoveho retezce KLIC v poli retezcu SEZNAM
DO i=1,SIZE(SEZNAM)
  IF (SEZNAM(i)==KLIC) GOTO 10
ENDDO
PRINT *, "KLIC v seznamu neni, stisk ENTER=KONEC"
READ *
STOP          ! ukonci program
10 CONTINUE
PRINT *, "Hledany retezec byl nalezen v polozce ",i
...          ! pokracovani programu

```

List. 3.7: Příklad nepovoleného vnoření konstrukcí

```

IF (<logicky_vyraz>) THEN    !!! NEPOVOLENÉ (a nesmyslné)
  DO i = 1,m                !!! překrývání
    ...                    !!! konstrukcí
  END IF
...
END DO

```

- Zpřehledňují zápis a kontrolu programu, zvláště když obsahují delší a vložené cykly.
- Pro vložené cykly umožňují, spolu s příkazy CYCLE a EXIT, předání řízení z vnitřního cyklu do vnějšího. Schematické znázornění takové možnosti je v List. 3.8. Pro používání této možnosti však platí totéž co pro GOTO; při nevhodném použití může program velice znepřehlednit.

List. 3.8: Příklad možného využití pojmenování vnořených DO konstrukcí

```

prvni: DO i=1,m
  ...
  druhy: DO j=1,n
    ...
    IF (<logicky_vyraz>) THEN
      ...
      CYCLE druhy
    ELSE
      ...
      CYCLE prvni
    END IF
  END DO druhy
  ...
END DO prvni

```

Na závěr ještě upozornění. Mnoho cyklů pracujících s prvky polí se dá elegantně zapsat pomocí operací a funkcí, které známe z kap. 2. Je však potřeba přemýšlet a vyvarovat se zkratkovitých závěrů. Jistě snadno poznáte, že cyklus

```
DO i = 2,n
```

```

      v(i) = v(i-1)+u(i)
    END DO

```

nedá stejný výsledek jako

```

      v(2:n) = v(1:n-1)+u(2:n) .

```

3.4 Příkaz a konstrukce FORALL

Na závěr kapitoly uved' me ještě elegantní konstrukci (resp. příkaz) FORALL, zavedenou až ve Fortranu 95. Její používání vyžaduje obezřetnost, nemá-li dojít k chybám. Důvod je v tom, že na první pohled může začátečníkovi připomínat normalní cykly DO, ve skutečnosti však postupuje při běhu programu jinak.

Úplná syntaxe konstrukce FORALL je v List. 3.9, kde index je skalární celočíselná proměnná, která se mění od *odi* do *doi*. Je-li krok jiný než 1, uvede se jako třetí položka. Maximální počet indexů je samozřejmě sedm (max. počet indexů pole ve Fortranu 95). Poslední položka hlavičky – volitelný <skalarni_logicky_vyraz> – plní funkci obdobnou příkazu *where*. Konstrukce FORALL může obsahovat (v <blok_prikazu>) *pouze* přiřazovací příkazy, konstrukce *where* a vložené konstrukce FORALL.

List. 3.9: Syntaxe konstrukce FORALL

```

[<jmeno>:] forall (index=odi:doi[:krok][, index=odi:doi[:krok]... &
                                     [, <skalarni_logicky_vyraz>])
      <blok_prikazu>
END forall [<jmeno>]

```

Nejlépe snad vše objasní několik příkladů. Základní rozdíl mezi DO a FORALL: v cyklu DO je pořadí provádění příkazů jasně zadané, ve FORALL nikoliv. Rozdíl plyne z toho, že DO garantuje *sekvenční* zpracování a FORALL počítá s možností *paralelní* (víceprocesorové) realizace příkazu; v přiřazovacím příkazu proto nemohu bez rizika použít na pravé straně prvek pole s nižším indexem než na levé straně (např. $a(i)=2*a(i-1)$), protože nemusí mít ještě předpokládanou hodnotu.

Zásadní rozdíl v realizaci DO a FORALL je zřejmý z výsledků programu podle List. 3.10. Výstup z tohoto programu je

```

Cyklus DO
u =  1  2  3  4  5
v =  5  5  5  5  0
FORALL
u =  1  2  3  4  5
v =  2  3  4  5  0

```

Proč jsou výsledky různé? Při deklaraci je pole *u* inicializované na hodnotu 5 a pole *v* na 0. V cyklu DO se provede $u(1)=1$, $v(1)=u(2)=5$, ..., $u(4)=4$, $v(4)=u(5)=5$, zůstává $u(5)=5$ a $v(5)=0$. V cyklu FORALL *nejprve* proběhne pro všechna *i* první příkaz, tj. sekvence $u(1)=1$, ..., $u(4)=4$ a *potom* pro všechna *i* druhý příkaz, tj. $v(1)=u(2)$, ..., $v(4)=u(5)$.

Jestliže je <blok_prikazu> tvořen jediným příkazem, můžeme použít příkaz FORALL, který je tvořený hlavičkou z List. 3.9 následovanou výkonným příkazem. Příklad – vytvoření spodní trojúhelníkové matice – je v List 3.11.

List. 3.10: Porovnání práce DO a FORALL

```

PROGRAM T_forall1
INTEGER :: i
INTEGER,DIMENSION(5) :: u=5, v=0

DO i=1,4
    u(i) = i
    v(i) = u(i+1)
END DO
PRINT *, "Cyklus_DO"
PRINT "(a,5i4)", "u=",u,"v=",v

FORALL (i=1:4)
    u(i) = i
    v(i) = u(i+1)
END FORALL
PRINT *, "FORALL"
PRINT "(a,5i4)", "u=",u,"v=",v
END PROGRAM T_forall1

```

List. 3.11: Příklad příkazu FORALL (trojúhelníková matice)

```

PROGRAM T_forall2
INTEGER :: i, j, m
INTEGER,DIMENSION(:,:),ALLOCATABLE :: A
CHARACTER(LEN=10) :: forma
m = 5
ALLOCATE(A(m,m))
FORALL (i=1:m, j=1:m,i<=j) A(i,j) = i+j-1
forma = "("//char(48+m)//"i4)"      ! 48 je dekadický kód znaku 0
PRINT *, A
END PROGRAM T_forall2

```

4 Vstupy a výstupy

Základní možnosti vstupu a výstupu, především z klávesnice a na displej, jsme již zvládli. Problematika vstupů a výstupů je však velice rozsáhlá a Fortran 95 poskytuje velmi silné prostředky pro její zvládnutí. Tyto jeho rozsáhlé možnosti nejsou samoučelné. Nejde totiž jen o výpočetní část projektů. Ve fyzikální a technické praxi je potřebné zvládat spolupráci počítače s nejrůznějšími měřicími přístroji, které přijímají a vydávají data v nejrůznějších formátech. Na druhé straně běžný uživatel jazyka obvykle potřebuje jen několik základních instrukcí pro čtení vstupních dat z klávesnice nebo disku, zápis dat na disky a výstup výsledků výpočtů na displej nebo tiskárnu. V této kapitole se proto omezíme jen na doplnění dosavadních poznatků v tomto směru.

4.1 Několik terminologických doplňků

Vstupní a výstupní (dále budeme psát v/v) operace pracují se *soubory (files)* dat, které jsou tvořeny posloupností *záznamů (records)*. V běžném textovém souboru představují záznamy např. jednotlivé řádky textu. Kromě vlastních dat musí být v souboru *znaky pro ukončení záznamu* – EOR (např. v textových souborech je to LF(Unix), CR(Mac), CR+LF(DOS)) a *znak pro ukončení souboru* – EOF.

Záznamy v souborech mohou obsahovat *formátovaná* nebo *neformátovaná data*. První případ již dobře známe. Formátovaný výstup jsme získávali pomocí formátovacích řetězců a výsledek jsme viděli např. na displeji; o formátovaném vstupu se ještě zmíníme. Počítač však pracuje s daty, která jsou v paměti zpravidla uložena v binárním tvaru, tj. jako posloupnost nul a jedniček (viz. Dod. ??). Jestliže nepotřebujeme čitelný výstup, můžeme ukládat a následně i číst data v tomto *neformátovaném* tvaru. Neformátovaný zápis a čtení zpravidla zabírá méně místa na médiích, je rychlejší a neztrácíme nic z přesnosti dat. Velice vhodný bude např. pro ukládání a následné čtení mezivýsledků při rozsáhlejších výpočtech. Nevýhodou neformátovaných dat je, že nejsou lidsky čitelná a mohou vznikat potíže při jejich přenosu na jiný počítač (jejich tvar závisí na vnitřní reprezentaci dat v daném počítači). Záznam může být tvořen jen formátovanými daty (*formátovaný záznam*) nebo jen neformátovanými daty (*neformátovaný záznam*).

Soubor je tvořený posloupností záznamů ukončených znakem EOF. V souboru mohou být jen formátované záznamy nebo jen neformátované záznamy (nelze je v jednom souboru míchat). Budeme rozlišovat dva druhy souborů:

- *externí soubory* jsou uloženy na periferních zařízeních, jako jsou disky, pásky, CD; obecně za soubor považujeme i klávesnici, tiskárnu a pod. ,
- *interní soubory* jsou uloženy v paměti jako textové řetězce; seznámili jsme se s nimi, včetně možného využití, již v odst. 2.8.

Další dva typy souborů, které Fortran 95 umožňuje:

- *soubory se sekvenčním přístupem (sequential files)* mají začátek a konec, záznamy jsou v nich uloženy v přirozené posloupnosti jeden za druhým,
- *soubory s přímým (nebo náhodným) přístupem (direct (random) access or indexed files)*; všechny záznamy mají stejnou délku, záznam je určen svým indexem a je možné číst, zapsat nebo přepsat libovolný zadaný záznam.

Všechny soubory ve Fortranu 95 jsou sekvenční, pokud nejsou deklarované jako indexované (zadáním specifikátorů ACCESS="direct" a RECL=<delka_zaznamu> v příkazu OPEN). Na rozdíl od indexovaných souborů je možné v sekvenčních souborech přepsat (bez ztráty ostatních záznamů) jen poslední

záznam. Soubory s náhodným přístupem jsou užitečné např. v aplikacích, které vyžadují vyhledávání záznamů (skoky v souboru) a případně i jejich změnu. Záznamy v sekvenčních souborech se musí číst postupně a v případě potřeby je nutné se vracet nebo nastavit soubor opět na začátek.

4.2 Formátovaný vstup dat

Formátování výstupů pomocí formátovacích řetězců již umíme; naprosto stejné formátovací řetězce je však možné použít i pro vstup dat příkazem READ. Při vstupu z klávesnice formátovaný vstup příliš užítku nepřinese; spíše by ho komplikoval a proto jsme ho zatím nepoužívali. Neobyčejně užitečný však může být při čtení dat ze souborů v nichž jsou data v záznamech známým způsobem uspořádána (např. do sloupců známé šířky). Potom je velice snadné přečíst z každého záznamu buď všechny položky nebo některé vynechat a podobně. Jednoduchý příklad je v List. 4.1. V programu lze změnou hodnoty keyboard nastavit čtení z klávesnice nebo z interního souboru vstup. Dokážete zjistit proč při čtení z interního souboru vstup bylo nutné místo jednoduchého seznamu položek ip použít načtení jen určitého počtu položek?

List. 4.1: Příklad formátovaného vstupu

```
PROGRAM T_vv1
INTEGER :: i,j
INTEGER,DIMENSION(20) :: ip = 10 ! indikuje položky neovlivněné čtením
CHARACTER(LEN=8),DIMENSION(2) :: forma=("/(20i1)","(10i2)")
CHARACTER(LEN=20) :: vstup = "12345678901234567890"
LOGICAL :: keyboard = .true.

DO i=1,2
  IF (keyboard) THEN
    PRINT *, "Zadej_20_cifer:_",vstup
    READ forma(i),ip
  ELSE
    READ(UNIT=vstup,FMT=forma(i))(ip(j),j=1,20/i)
  END IF
  PRINT *,ip
END DO
READ *
END PROGRAM T_vv1
```

4.3 Příkazy pro zápis a čtení

Všechny příkazy vlastně již v nějaké podobě známe. Zde jen v List. 4.2 shrnujeme jejich obecnou strukturu a dále osvětlíme některé dosud neznámé položky. Položku <fmt_retez> již celkem dobře známe (viz. Tab. 1.5, 2.4 a odst. 2.8). Rovněž jsme již použili různé tvary volitelných seznamů v/v položek (viz. opět např. odst. 2.8). Všimněme si proto podrobněji některých specifikátorů, které mohou (musí) být v položce <ridici_specifikace>.

List. 4.2: Příkazy pro čtení a zápis dat

```

READ (<ridici_specifikace>) [<seznam_vstupnich_polozek>]
READ <fmt_retez> [, <seznam_vstupnich_polozek>]
WRITE (<ridici_specifikace>) [<seznam_vystupnich_polozek>]
PRINT <fmt_retez> [, <seznam_vystupnich_polozek>]

```

4.3.1 Propojení se souborem, specifikátor UNIT

Každá v/v operace pracuje s nějakým konkrétním souborem. V příkazech READ a WRITE se neodkazuje přímo na tyto konkrétní soubory, ale na celé číslo ze specifikátoru

UNIT = <cele_cislo>

. Přiřazení určitého čísla souboru se děje dvojnásobem:

- v prováděném programu příkazem OPEN (viz. odst. 4.4.1),
- pro některé standardní soubory provede přiřazení operační systém (kompilátor).

Platí následující pravidla a omezení:

- UNIT = * specifikuje v/v zařízení (soubor) přidělené operačním systémem. Ve zkrácených příkazech – druhá varianta READ a PRINT v List. 4.2 – se předpokládá a neuvádí se. V úplných formách READ a WRITE **musí být vždy uvedeno** a musí být vždy doplněno formátovacím specifikátorem FMT=<fmt_retezec>. Použitelné je pouze pro sekvenční přístup.
- Číslo UNIT identifikuje jeden a pouze jeden soubor v F programu. Je globální pro celý program; může být přiděleno např. v jedné proceduře a použito v jiné.
- Napojení je možné jen na externí soubory.
- Pro interní soubory se užije UNIT=<jmeno_textoveho_retezce>.

V závislosti na systému jsou použitelná jen některá celá čísla, zpravidla v rozsahu 1 – 99. Určitá čísla bývají vyhrazena pro systém, např. UNIT=5 pro vstup a UNIT=6 pro výstup, pro všechny programy. Pro zjištění zda soubor existuje se používá příkaz INQUIRE (odst. 4.5).

4.3.2 Chyby v/v operací, konec souboru a záznamu – IOSTAT

Se specifikátorem IOSTAT jsme se setkali již v odst. 1.7.2,2.8 takže k jeho použití stačí dodat již jen málo. Kromě skutečných chyb (např. pokus číst posloupnost písmen jako číslo a pod.) je použitelný též ke *zjištění konce souboru* – znaku EOF a *konce záznamu* – EOR. Tyto stavy indikuje *vrácením záporného čísla* (pro chyby vracel kladná čísla). Nastanou-li oba případy najednou, tj. EOF a současně chyba, vrací se kladné číslo. Příklad v němž se zjišťuje počet řádků textového souboru je v List. 4.3. Za opětovné připomenutí stojí, že význam kódů chyb najdete v [4, odst.6.2.2].

4.3.3 Specifikátory ADVANCE, SIZE

S jednoduchým použitím volitelného specifikátoru ADVANCE jsme se již seznámili v odst. 1.8 a použili ho např. v List. 1.8, 1.17. Víme, že může být ADVANCE = {"yes" | "no"}, standardně nastavená hodnota je "yes". Při nastavení ADVANCE="no" budeme mluvit o *postupném čtení nebo zápisu dat (non-advancing data transfer)*. Použití lze pouze s explicitně formátovanými externími sekvenčními soubory.

Příkazy READ a WRITE s ADVANCE="yes" (*advancing data transfer*) čtou nebo zapisují vždy celé záznamy (např. řádky v textovém souboru), tj. nastavují po každé v/v operaci interní *ukazatel polohy*

List. 4.3: Příklad: počet řádků v souboru

```

PROGRAM PocitejRadky
CHARACTER(LEN=120) :: radek
INTEGER :: citac, stav
citac = 0
DO
  READ(UNIT=11,FMT="(a)",IOSTAT=stav) radek  ! soubor je uz prirazen
  IF (stav<0) EXIT
  citac = citac+1
END DO
PRINT *, "Soubor_obsahuje", citac, "radku"
END PROGRAM PocitejRadky

```

v souboru na začátek dalšího záznamu. Při postupném čtení zůstává tento ukazatel často *uvnitř* záznamu. Při pokusu číst za koncem aktuálního záznamu, je splněna podmínka pro konec záznamu (IOSTAT=-2 v F) a ukazatel přejde na začátek následujícího záznamu. Jestliže je nastaven volitelný specifikátor SIZE=<int_prom>, je v celočíselné proměnné <int_prom> počet právě přečtených znaků. Vše by měl ujasnit příklad v programu List. 4.4, který počítá znaky v souboru. Zkuste také experimentovat se specifikátorem SIZE např. tak, že „odpoznámkuje“ řádek 24 a změníte hodnotu LEN v deklaraci proměnné znaky. Aby program pracoval, musíme mu nabídnout nějaký externí soubor. Příkazem OPEN trochu předbíháme a spojujeme s UNIT=11 s nějakým textovým souborem; vhodný je např. zdrojový text prováděného programu (předpokládáme, že je uložen v pracovním adresáři v souboru Tadvance.f95).

Oba typy v/v operací (ADVANCE="yes" a ADVANCE="no") je možné provádět na témže záznamu nebo souboru. Můžeme např. přečíst několik prvních znaků záznamu s ADVANCE="no" a zbytek záznamu standardním způsobem s ADVANCE="yes". Běžně se to používá v kombinaci, kterou jsme již použili dříve a je také v řádcích 9, 10 programu List. 4.4: postupným WRITE napíšeme na obrazovku sdělení (*prompt*) a následující READ přečte zadaný vstup až do konce záznamu (EOR vytvoříme stiskem Enter). Závěrem znovu připomeňme, že postupné v/v operace nelze provádět na interních souborech a na souborech u nichž jsou vstupní či výstupní operace řízené seznamem v/v položek (např. (ip(j), j=1, 20/i) v List. 4.1).

4.4 Otevření a zavření souboru

4.4.1 Příkaz OPEN

Příkaz OPEN realizuje propojení mezi externím souborem a číslem UNIT s nímž potom pracují příkazy jazyka. Toto propojení (nastavení UNIT) je automaticky nastavené pro některé standardní soubory (klávesnice, displej). Příkaz OPEN se může použít kdekoliv v programu. Jakmile se jednou provede, platí UNIT globálně – v hlavním programu i procedurách – po celou dobu běhu programu nebo do uzavření souboru (zrušení propojení UNIT↔soubor) příkazem CLOSE. Syntaxe příkazu OPEN je

```
OPEN (UNIT=<int_vyraz>, <seznam_specifikatoru_propojeni>)
```

kde kromě specifikátoru UNIT může v (<seznam_specifikatoru_propojeni>) být

Specifikátor	Typ proměnné, možné hodnoty
IOSTAT=	proměnná typu INTEGER, nastaví se na 0 při úspěšném provedení příkazu, při chybě je to kladné číslo, které specifikuje chybu
FILE=	textový řetězec, jméno souboru propojovaného s UNIT
STATUS=	textový řetězec, {"old" "new" "replace" "scratch"}
ACCESS=	textový řetězec, {"sequential" "direct"}
FORM=	textový řetězec, {"formatted" "unformatted"}
RECL=	kladné celé číslo, délka záznamu pro ACCESS="direct" a maximální délka záznamu pro ACCESS="sequential"
POSITION=	textová řetězec, {"rewind" "append"}
ACTION=	textový řetězec, {"read" "write" "readwrite"}

K uvedeným specifikátorům je třeba dodat ještě několik podrobnějších informací:

- ▷ V příkazu OPEN *musí být povinně uveden* specifikátor STATUS; pro "old" musí soubor existovat a pro "new" nikoliv. Jestliže s "replace" soubor neexistuje, bude vytvořen a status se změní na "old". Status "scratch" znamená soubor, který existuje jenom během práce programu nebo do uzavření příslušné UNIT pomocí CLOSE a nesmí mít jméno (vyklučuje FILE=). Použití "replace" je vhodné při nejisté existenci souboru a pokud existuje, je žádoucí ho přepsat.
- ▷ V příkazu OPEN *musí být také uveden* specifikátor ACTION. Pro ACTION="read" nesmí být STATUS={"new"|"replace"}. Pro status "scratch" lze užít *pouze* ACTION="readwrite".
- ▷ Příkaz OPEN s ACCESS="sequential" a STATUS="old" vyžaduje přítomnost POSITION.
- ▷ Při otvírání nového souboru (STATUS="new") musí být uveden ACCESS.
- ▷ Pro existující soubor musí ACCESS odpovídat metodě nastavené při vytvoření souboru; není-li uveden, předpokládá se sekvenční přístup.
- ▷ Je pochopitelné, že každý ze specifikátorů může být v argumentech OPEN (a týká se to i následujících CLOSE a INQUIRE) použit *jen jednou*.

Kromě OPEN uvedeného již v List. 4.4 uveďme ještě jako příklad

```
OPEN(UNIT=11,IOSTAT=stav,STATUS="scratch",ACTION="readwrite")
OPEN(UNIT=8,ACCESS="direct",FILE="graf.dat",STATUS="old",ACTION="read")
```

4.4.2 Příkaz CLOSE

Příkaz CLOSE ukončí propojení UNIT ↔ soubor. Ještě v témže programu může být novým OPEN obnoveno (není možný nějaký jiný způsob znovuotevření uzavřeného souboru) nebo číslo UNIT uzavřeného programu může být použito ke zcela odlišnému propojení. Všechny otevřené soubory se automaticky uzavřou při ukončení programu. Dobrým zvykem však je uzavírat soubory jakmile nejsou potřeba. Jednak to zpřehlední program a také zabráníte nekorektnímu uzavření souboru při nečekaném ukončení programu; důležité je to především pro soubory do nichž se zapisuje. Bez správného uzavření jsou soubory nepoužitelné (nečitelné).

Syntaxe příkazu CLOSE je

```
CLOSE (UNIT=<int_vyraz>,<seznam_specifikatoru_uzavreni>)
```

kde <seznam_specifikatoru_uzavreni> může obsahovat specifikátory

```
IOSTAT = <promenna_typu_INTEGER>
STATUS = { "keep" | "delete" }
```

K uvedeným specifikátorům opět dodejme ještě několik podrobnějších informací:

Tabulka 4.1: Dotazovací specifikátory pro příkaz INQUIRE

access=	character	name=	character	readwrite=	character
action=	character	named=	logical	recl=	integer
direct=	character	nextrec=	integer	sequential=	character
exist=	logical	number=	integer	unformatted=	character
form=	character	opened=	logical	write=	character
formatted=	character	position=	character		
iostat=	integer	read=	character		

- ▷ Uzavírat lze jen externí soubory.
- ▷ Při aplikaci CLOSE na UNIT nepropojenou se souborem se neprovede nic a nehlásí ani chyba.
- ▷ Význam specifikátoru STATUS je zřejmý z přiřazovaných hodnot; soubory s "keep" zůstanou po provedení CLOSE zachovány a s "delete" se vymažou. Není-li STATUS uveden, je pro "scratch" soubory přednastaven na "delete" a pro ostatní na "keep".
- ▷ Specifikátor IOSTAT má stejný význam jako v příkazu OPEN.

4.5 Vše o v/v prozradí příkaz INQUIRE

Příkaz INQUIRE dokáže během práce programu zjistit všechny potřebné informace o existenci souboru, propojení s UNIT, přístupové metodě atd. Jeho syntaxe je

```
INQUIRE (<seznam_inquire_specifikatoru>)
```

Příkaz má *dvě podoby* podle toho zda <seznam_inquire_specifikatoru> obsahuje

```
UNIT = <int_vyraz>   nebo   FILE = <jmeno_souboru> ,
```

nikoliv však UNIT a FILE současně. V prvním případě se budou získané informace vztahovat k zadané UNIT, ve druhém pak k souboru zadaného jména. Samozřejmě, že požadované informace na sobě závisí. Jestliže např. neexistuje propojení UNIT se souborem, nemá smysl se v UNIT variantě dotazovat třeba na FORM. Pokud to přesto provedeme, vrátí se nám hodnota UNDEFINED. Na druhé straně ve FILE variantě není nutné aby soubor byl propojen s UNIT nebo vůbec existoval; tyto informace naopak můžeme dotazem získat.

Dotazovací specifikátory, které mohou být v <seznam_inquire_specifikatoru> jsou spolu s typy skalárních proměnných pro vrácené hodnoty v Tab. 4.1. Význam specifikátorů v této tabulce je snad zřejmý.

Příkaz INQUIRE je ještě možné použít ve tvaru

```
INQUIRE (IOLENGTH=<int_prom>)<seznam_vv_polozek>
```

ke zjištění délky seznamu v/v položek. Hodnota vrácená v INTEGER proměnné <int_prom> může být použita pro specifikátor RECL= v příkazu OPEN.

V programu List. 4.5 je soustředěno několik příkladů použití příkazu INQUIRE. Nejprve se zjistí potřebná délka záznamu pro číslo typu INTEGER, REAL a textový řetězec s LEN=12. S touto hodnotou RECL se potom otevře soubor test.dat s přímým přístupem a neformátovaným zápisem. V následujícím cyklu se pomocí INQUIRE získávají informace o souborech přiřazených UNIT=1 až 12. Navíc se do otevřeného souboru test.dat zapisují data v opačném pořadí. Z výpisu na obrazovce uvidíte, že otevřeny jsou tři soubory: UNIT=5 pro čtení, UNIT=6 pro zápis a UNIT=10 pro zápis i čtení. První dva otevřel automaticky systém. Pomocí příkazů READ(UNIT=5, ...) a WRITE(UNIT=6, ...) ověřte, že

UNIT=5 je skutečně spojen se vstupem z *klávesnice* a UNIT=6 s výstupem na *displej*. V závěru programu je ještě ukázka varianty INQUIRE(FILE="jmeno",...) a vytvořený soubor je uzavřen s požadavkem STATUS="keep" (zůstane zachován na disku).

4.6 Neformátované v/v operace

Soubor s neformátovaným zápisem jsme vytvořili v předcházejícím programu List. 4.5 zápisem v řádce 22. Skutečnost, že šlo o přímý (indexovaný) zápis není podstatná. Neformátovaný přenos dat je možný i pro sekvenční soubory. Jestliže se podíváte na vytvořený soubor test.dat v nějakém textovém editoru, uvidíte jen zapsané texty ("ZaznamX") a zbývající znaky budou většinou nečitelné. Jestliže zobrazíte obsah souboru v nějakém editoru, který umí hexadecimální zobrazení (zobrazí obsah každého bytu souboru jako hexadecimální číslo), dostanete výpis který začíná podle podle obr. 4.1. Víme, že délka záznamu pro jedno číslo typu INTEGER, jedno číslo typu REAL a textový řetězec délky 12 znaků byla stanovena na 20 bytů.

Obrázek 4.1: Začátek hexadecimálního výpisu neformátovaného souboru

V dod. ?? se dozvíme, že se číslo INTEGER ukládá do 4 bytů a číslo typu REAL rovněž do 4 bytů. První 4 byty jsou

0c 00 00 00 hexadecimálně,
tj. 00001100 00000000 00000000 00000000 binárně a 12 0 0 0 dekadicky.

V programu jsme prováděli zápis v cyklu DO j=1,12; v řádce 22 jsme jako první položku sice zapsali j, ale do záznamu 12 (REC=13-j). V prvním záznamu (REC=13-12) je proto jako první položka skutečně zapsané j=12. Reálné číslo 13.0-j (převodu na reálné číslo jsme dosáhli právě tímto zápisem; 13-j by zapsalo typ INTEGER, viz. odst. 1.4.2) je zapsané způsobem, který tak zřetelně nedekodujeme. Objeví se ale, až ho následujícím programem přečteme. Od bytu 9 je zapsán čitelný textový řetězec "Zaznam1", který je na předepsaných 12 znaků (LEN=12) doplněn prázdnými znaky (char(0)). Další záznam začíná v 21 bytu.

Náhodné čtení tohoto souboru je v programu List. 4.6. V této podobě program bude číst záznamy tak jak byly zapsány. Je však nutné si uvědomit, že v souboru není nikde informace, jak se má 20 bytů záznamu interpretovat. Snadno si ověříte, že program bude uspokojivě fungovat, když v řádce 14 budete požadovat přečtení dvou dvoubajtových celých celých čísel, tj.

```
READ(UNIT=11,REC=j,IOSTAT=stav)m,n,x,text ! upravit format v radku 16
```

nebo dvou dvoubajtových a jednoho čtyřbajtového celého čísla příkazem

```
READ(UNIT=11,REC=j,IOSTAT=stav)m,n,j,text ! upravit format v radku 16,
```

případně různě interpretovat byty textu. Informaci o struktuře záznamu tedy musí mít programátor.

4.7 Nastavení souborového ukazatele: BACKSPACE, REWIND, ENDFILE

Při v/v operacích se zpravidla mění *poloha souborového ukazatele*. Při indexovaných operacích se zapisuje/čte vždy celý záznam a ukazatel se nastavuje specifikátorem REC; při zápisu se však nemusí zaplnit

všechny byty záznamu a při následném čtení se nemusí číst všechny položky záznamu (zkuste např. v řádku 14 programu List. 4.6 dát do seznamu čtených položek jen *i* nebo *i,x* a pod. Systém musí samozřejmě vždy vědět v kterém místě souboru nebo záznamu se nachází, kde se bude číst nebo zapisovat. K tomu účelu si musí zřídit objekt, který jsme nazvali *souborový ukazatel*. Možné polohy takového ukazatele jsou schematicky znázorněny v obr. 4.2.

Obrázek 4.2: Znázornění poloh souborového ukazatele

Můžeme si představit, že ukazatel je „v klidu“ nastaven mezi záznamy (znaky) a teprve příkaz READ nebo WRITE ho posune do aktuální polohy. Po skončení příkazu je nastaven v následující mezipoloze záznamů (znaků). Při sekvenčním čtení vyvolá v/v operace posun na následující záznam (znak při znakové operaci podle obr. 4.2). Při indexovém čtení nebo psaní se specifikátorem REC= nastaví příslušný aktuální záznam, provede se operace a ukazatel se přesune za tento záznam. Viděli jsme, že poloha ukazatele se např. v příkazu OPEN nastavuje specifikátorem POSITION= (viz. např. List. 4.4, řádek 13). Existuje však ještě trojice příkazů, kterými můžeme polohu ukazatele v *otevřeném externím sekvenčním souboru* ovlivňovat.

▷ Příkaz **BACKSPACE**

posune ukazatel před aktuální záznam (pokud existuje, tj. je nastaven) nebo před předcházející záznam (když aktuální záznam neexistuje). To umožní např. přepsání právě zapsaného záznamu nebo nové čtení právě přečteného záznamu. Jeho syntaxe je

```
BACKSPACE(UNIT=<cele_cislo>[ , IOSTAT=<celociselna_promenna>]).
```

Funkci volitelného specifikátoru IOSTAT již dobře známe. Jestliže není nastaven aktuální záznam a předchozí záznam neexistuje, neprovede se nic. Je-li předchozím záznamem EOF, ukazatel se přesune před něj. Příkaz *nelze použít* v soubor se záznamy zapisovanými řízeným seznamem položek (cyklus jako položka seznamu, viz. odst. 2.8).

▷ Příkaz **REWIND**

nastaví ukazatel na začátek souboru *před první záznam*. Je bez účinku, jestliže ukazatel již v této poloze je, např. vlivem specifikátoru POSITION. Syntaxe:

```
REWIND(UNIT=<cele_cislo>).
```

▷ Příkaz **ENDFILE**

zapiše záznam EOF a ukazatel nastaví za něj. V této poloze vyvolá pokus o zápis záznamu chybu. Proto je potřeba po provedení tohoto příkazu provést REWIND nebo BACKSPACE. Syntaxe příkazu je

```
ENDFILE(UNIT=<cele_cislo>[ , IOSTAT=<celociselna_promenna>]).
```

List. 4.4: Příklad: počítání znaků v souboru

```

1 PROGRAM Tadvance
2 INTEGER,PARAMETER :: EOF = -1, EOR = -2 ! konec souboru a zaznamu
3 INTEGER :: citacA=0, citacC=0, citacZ=0 ! vynulovane citace
4 INTEGER :: stav, vel ! pro IOSTAT a SIZE
5 CHARACTER(LEN=1) :: znaky ! pro nacistane znaky
6 CHARACTER(LEN=20) :: soubor ! pro jmeno souboru
7 ! cteni jmena a otevreni souboru
8 OtevriSoubor: DO
9 WRITE(UNIT=*,FMT="(a)",ADVANCE="no")"Zadej jmeno souboru "
10 READ *,soubor
11 ! otevreni souboru pro cteni
12 OPEN(UNIT=11,IOSTAT=stav,FILE=soubor,STATUS="old",&
13 ACTION="read",POSITION="rewind")
14 IF (stav == 0) EXIT OtevriSoubor ! soubor je otevren
15 IF (stav==175) THEN
16 PRINT *, "Pozadovany soubor neexistuje, zadej znovu"
17 CYCLE OtevriSoubor ! novy pokus
18 ELSE
19 STOP "Jina chyba" ! ukonceni programu
20 END IF
21 END DO OtevriSoubor
22 cteni: DO ! zacatek cteni souboru a pocitani znaku
23 READ(UNIT=11,FMT="(a)",ADVANCE="no",IOSTAT=stav,SIZE=vel) znaky
24 ! print "(2(i5,a),a5)",vel," ",ichar(znaky(1:1))," ",znaky
25 IF (stav==EOR) THEN
26 CYCLE cteni ! byl konec zaznamu, dalsi READ
27 ELSE IF (stav==EOF) THEN ! konec souboru, ukoncit cteni
28 EXIT cteni
29 ELSE ! byl precten znak
30 SELECT CASE (znaky) ! trideni prectenych znaku
31 CASE ("A":"Z","a":"z") ! pismena anglicke abecedy
32 citacA = citacA+1
33 CASE ("0":"9") ! cifry
34 citacC = citacC+1
35 CASE DEFAULT
36 citacZ = citacZ+1 ! zbyvajici znaky
37 END SELECT
38 END IF
39 END DO cteni
40 PRINT *, "Pismen",citacA,"cifer",citacC,"a ostatnich",citacZ
41 CLOSE(UNIT=11) ! pro poradek uzavreni otevreného souboru
42 END PROGRAM Tadvance

```

List. 4.5: Příklady použití příkazu INQUIRE

```

1 PROGRAM Tinquire
2 LOGICAL :: ex,op
3 INTEGER :: j,reclen,delka
4 CHARACTER(LEN=12) :: acc,act,forma,fmtd,jmeno
5
6 jmeno="test.dat"                ! jmeno otevireneho souboru
7 INQUIRE(IOLENGTH=delka) j,1.0e30,jmeno ! zjisteni delky zaznamu
8 PRINT *, "Delka INTEGER+REAL+CHARACTER(LEN=12)= ",delka," bytu",char(10)
9     ! otevreme soubor s primym pristupem a delkou zaznamu "delka"
10 OPEN(UNIT=10,STATUS="replace",ACCESS="direct",ACTION="readwrite",&
11     FORM="unformatted",RECL=delka,FILE=jmeno)
12     ! dotaz na vlastnosti UNIT=1 az 12
13 PRINT *, " j op FORM          FORMATTED  ACCESS      ACTION    RECL"
14 PRINT *, "-----"
15 DO j=1,12
16     INQUIRE(UNIT=j,EXIST=ex,OPENED=op)
17     IF (ex) THEN                ! zkuste variantu IF (op) THEN
18         reclen=0                ! vynulovat pred dotazem (zkuste neprovest)
19         INQUIRE(UNIT=j,FORM=forma,FORMATTED=fmtd,ACCESS=acc,&
20             ACTION=act,RECL=reclen)
21         PRINT "(i3,l3,a14,2a12,a7,i6)",j,op,acc,act,forma,fmtd,reclen
22         WRITE(UNIT=10,REC=13-j) j, 13.0-j,"Zaznam"//char(48+13-j)
23     END IF
24 END DO
25     ! dotaz na otevreny soubor FILE=
26 INQUIRE(FILE="test.dat",EXIST=ex,OPENED=op,ACCESS=acc,RECL=reclen)
27 PRINT *,char(10),"Soubor "///jmeno,char(10),&
28     "EXIST=",ex,"OPEN=",op,"ACCESS=",acc,"RECL=",reclen
29 CLOSE(UNIT=10,STATUS="keep")    ! soubor uzavrit a zachovat
30 END PROGRAM Tinquire

```

List. 4.6: Indexované čtení z neformátovaného souboru

```
1 PROGRAM Tunfdir
2 USE std_type           ! umožni nasledující deklaraci (viz dod.C)
3 INTEGER(KIND=i2b) :: m,n   ! celé číslo uloženo do 2 bytu
4 INTEGER :: i,j,stav       ! odpovídá (KIND=i4b), je default
5 REAL :: x                ! jednoduchá přesnost, je default, (KIND=SP)
6 CHARACTER(LEN=12) :: text
7
8   OPEN(UNIT=11, FILE="test.dat", FORM="unformatted", RECL=20, &
9     ACCESS="direct", STATUS="old", ACTION="read")
10  ctirec:DO
11    WRITE(UNIT=6, FMT="(a)", ADVANCE="no") "Zadej číslo záznamu "
12    READ(UNIT=5, FMT=*, IOSTAT=stav) j   ! bude se číst j-ty záznam
13    IF (stav/=0) GOTO 10   ! vstup není celé číslo
14    READ(UNIT=11, REC=j, IOSTAT=stav) i,x,text
15    IF (stav/=0) GOTO 10   ! pro j<1 nebo j>12
16    WRITE(UNIT=6, FMT="(i4, f8.2, a14)") i,x,text
17  END DO ctirec
18 10 CONTINUE   ! program se da ukončit jen vytvořením chyby čtení
19  PRINT *, "Chyba čtení, stav=", stav
20  CLOSE(UNIT=11)
21 END PROGRAM Tunfdir
```

5 Procedury, funkce a moduly

Dodatky

A Instalace kompilátoru G95 a jeho základní použití

A.1 Instalace kompilátoru G95

Pro instalaci kompilátoru jazyka G95 potřebujete soubor:

g95-MinGW.exe	pro Windows v prostředí MinGW,
g95-x86-cygwin.tgz	pro prostředí Cygwin ve Windows
g95-x86-linux.tgz	pro Linux.

Najdete je na stránce www.g95.org (položka *Download prebuild binaries*). V dalším si podrobněji zmíním jen o instalaci ve Windows v prostředí MinGW (podrobnosti najdete na www.mingw.org; pro práci s kompilátorem G95 však nejsou nutné). Tato verze G95 je v prostředí Windows perspektivní a vzhledem ke snadné instalaci vhodná především pro běžné uživatele Windows. Pro uživatele prostředí Cygwin ve Windows a operačního systému Linux nebude jistě problém provést instalaci podle pokynů na stránce www.g95.org/docs.html.

Jestliže nemáte prostředí MinGW nainstalované (případně spolu s MSYS), pak prostě spusťte soubor (instalátor) g95-MinGW.exe. Instalace všeho potřebného se provede do adresáře, který zadáte; během instalace je vhodné odsouhlasit všechny nabídky instalačního programu. Stáhnete-li si později novější verzi, není třeba starou verzi odinstalovat. Nová instalace do původního adresáře starou verzi přepíše. Předpokládejme, že jste provedli instalaci do adresáře D:\G95.

Adresář D:\G95 bude po dokončení instalace obsahovat podadresáře:

bin
doc
lib

V adresáři doc najdete soubor Readme a v něm i další podrobnosti o instalaci. Pokud máte prostředí MinGW nainstalované, doplníte do něj kompilátor G95 tak, že provedete výše popsanou instalaci do kořenového adresáře MinGW, např. D:\MinGW.

A.2 Instalace FortranTools

V dalším kroku je vhodné provést instalaci balíku FortranTools. Pohodlně ho stáhnete pomocí ftp z adresy <ftp://ftp.swcp.com/pub/walt/F> (nebo přes stránku www.fortran.com/fortran/). Najdete zde instalaci pro Windows (soubor FortranTools_windows_F.zip) i odpovídající soubor pro Linux (FortranTools_linux32_F.tgz). Z tohoto volně dostupného balíku by bylo možné nainstalovat v prostředí Cygwin jazyk F založený na G95 (podrobnější zmínka je v předmluvě). Jeho syntaxe (viz. [2]) je proti syntaxi původního jazyka F [1] ochuzena o několik výhodných konstrukcí. Doporučuji proto v této fázi neprovádět úplnou instalaci tohoto balíku, ale pouze ho „rozbalit“ do nějakého adresáře mimo instalační adresář G95 (např. D:\FortranTools). Získáme tak mnoho neobyčejně užitečného materiálu, především v podadresáři doc (najdete tam např. publikace [5],[6] a [7]). Především pro začátečníky ve Fortranu 95 pak může být velice užitečné studium krátkých modulů v adresáři Examples. Na jiném místě si ukážeme jak využít obsah adresářů lib, src.

A.3 Základy práce s kompilátorem G95

A.3.1 Kompilace

Příkaz `g95 -help` vypíše delší seznam "options", jejichž zadáním lze řídit činnost kompilátoru. Podrobně je jejich funkce rozebrána v manuálu [3], který jako `G95manual.pdf`, kterou najdete v podadresáři `DOC`. Zde jen shrneme základní informace o nejčastěji používaných činnostech. I když to není nutné, přijmeme dohodu, že zdrojové soubory budou mít *příponu* `f95` (je na ni nastaven editor SciTe instalovaný podle následujícího odst. ??).

▷ **Kompilace bez vytvoření spustitelného souboru**

```
g95 -c <jmeno_souboru>
```

Pokud zdrojový soubor obsahuje syntaktické chyby, jsme na ně upozorněni a nevytvoří se žádné soubory. Chyby opravujeme dokud kompilace neproběhne bez ohlášení chyb (*error*). Upozornění (*warning*) nebrání překladu, pouze upozorňují např. na deklarovanou a nepoužitou proměnnou a podobně. Po bezchybném překladu najdeme v pracovním adresáři dva soubory ¹⁾: `<jmeno_souboru>.mod` a `<jmeno_souboru>.o`.

▷ **Vytvoření spustitelného souboru bez doplňujících (vlastních) knihoven**

```
g95 [<objektove_soubory>] <jmeno_souboru> [-o <vystupni_soubor>]
```

Bez volitelných částí (uvedených v [...]) můžeme vytvořit pouze nejjednodušší programy, které nepotřebují nic jiného než standardní knihovny. Bez volby `-o` vznikne spustitelný soubor `a.exe`. Jméno podle našeho přání mu přiřadíme právě touto volbou (obvykle `-o <jmeno_souboru>`). Jestliže program využívá moduly, které se v objektovém tvaru (jako `*.o`) nachází v téže adresáři jako program, uvedeme je na místě volitelné položky `<objektove_soubory>` (seznam jmen souborů včetně přípony o oddělený mezerami); příklad je v odst. 1.9.2.

Obecný případ kompilace s doplňujícími knihovnami a moduly uloženými mimo pracovní adresář probereme v následujícím odstavci.

Uvedené příkazy budete používat nejčastěji. Úplné informace o možnostech a používání jazyka G95 najdete v manuálu [4], který je po instalaci uložen v adresáři `doc`.

A.3.2 Vytvoření vlastních knihoven a jejich použití

S kompilátorem se dodává řada modulů a knihoven, které jsou uloženy v předdefinovaných adresářích (viz. např. adresář `lib` ve stromové struktuře kompilátoru G95, odst. ??). Při jejich použití stačí uvést jméno příslušného modulu v klauzuli `USE`; o těchto modulech se dozvíte více v [4] a příklady použití najdete v programech, které jsou v adresáři `examples`.

Jakmile si začnete vytvořit větší počet vlastních modulů, je vhodné je uložit do zvláštního adresáře, aby byly zřetelně oddělené od systémových adresářů. Víme, že při kompilaci modulu (s volbou `-c`) se vytvoří dva soubory: `<jmeno_souboru>.mod` a `<jmeno_souboru>.o`. Kompilátor je při vytváření spustitelného programu potřebuje oba. Protože při kompilaci musí kompilátor (přesněji *linker*, který spojuje vše ve výsledný spustitelný soubor) vědět, kde potřebné soubory najde, je výhodné si vytvořit zvláštní adresář (používám `ModLib`) a do něj takto vytvořené soubory přesouvat. Kompilátor (*linker*) ovšem musíme o tomto adresáři informovat. Provedeme to tak, že do poslední uvedené příkazové řádky přidáme položky uvedené `-I` a `-L` takto (předpokládám, že je v obou případech použitý zmíněný adresář `ModLib`):

¹⁾Překládáme-li modul (zahájený klíčovým slovem `MODULE`); při překladu programu (programová jednotka zahájená `PROGRAM`) se vytvoří pouze `<jmeno_souboru>.o`.

```
g95 -I<cesta_k_ModLib> -L<cesta_k_ModLib> <jmeno_souboru> ...].
```

Adresář uvedený za `-I` obsahuje soubory `*.mod` a adresář za `-L` knihovny `*.a` (v uvedeném příkladu je obojí ve společném adresáři `ModLib`). Zdůrazněme, že je potřeba uvést úplnou cestu k použitému adresáři.

Pokud jde o soubory `*.mod` je přesun do adresáře dostačující. Má-li však kompilátor v adresáři `ModLib` najít objektové soubory, musíme je shrnout do nějaké knihovny, uvést její jméno a teprve z ní si kompilátor (resp. *linker*) potřebný kód vezme. Vytvořit z několika objektových souborů knihovnu je snadné. Jestliže jste provedli instalaci G95 podle odst. A.1, je v adresáři `bin` program `ar.exe`, který to umí. Protože jsme při instalaci nastavili do tohoto adresáře cestu (`path`), stačí když v CMD okně napíšete `ar` a vypíše se základní help pro použití tohoto „knihovníka“ (jestliže se to nestane, zkontrolujte instalaci G95). Nejjednodušší příkaz pro vytvoření knihovny je

```
ar -r <jmeno_knihovny> <seznam_obj_souboru> ,
```

kde `<jmeno_knihovny>` má strukturu `libx.a`; za `x` dosadíte zvolené jméno, takže knihovna se např. bude jmenovat `libmoje.a`. Položka `<seznam_obj_souboru>` je mezerami oddělený seznam objektových souborů, takže celý příkaz bude např.

```
ar -r libmoje.a modul1.o modul2.o ,
```

Objektové soubory zahrnuté do knihovny vypíšete příkazem

```
ar -t libmoje.a ,
```

Takto vytvořenou uložíme v adresáři uvedeném za klíčem `-L`. Jestliže v programu potřebujeme něco z této knihovny, uvedeme ji na příkazovém řádku za klíčem `-l` **na konci** příkazového řádku, např. takto:

```
g95 -ID:\ModLib -LD:\ModLib prog.f95 -o prog -lmoje1 -lmoje2.
```

Zde jsme předpokládali, že program potřebuje objektové moduly ze dvou knihoven – `libmoje1.a`, `libmoje2.a` – uložených v adresáři určeném klíčem `-L`. Všimněte si, že počáteční `lib` se za klíčem `-l` neuvádí. Uvádíme-li více knihoven, musíme vědět, že **záleží na pořadí** v němž jsou uvedeny. Pokud např. knihovna `moje1` používá procedury z knihovny `moje2`, musí být `-lmoje2` uvedeno až za `-lmoje1` (tak jak je to v uvedeném příkladu).

B Editor SciTe

B.1 Instalace

Zdrojové texty (programy) můžete psát v libovolném textovém editoru, který nevnáší do textu žádné formátovací příkazy. Nejjednodušší je např. *Notepad*, který je standardní součástí Windows, nikoliv však *WordPad* nebo dokonce *Word*. Výhodné jsou editory, které jsou schopné zvýraznit např. syntaktické prvky programovacího jazyka, udržovat úpravu textu (odsazení) a pod. To umí pro většinu běžných jazyků tzv. *programátorské editory*. Z českých je to např. *PsPad* (www.pspad.com).

Velmi dobře uživatelsky konfigurovatelný je editor SciTe (www.scintilla.org/SciTE.html), který existuje ve verzi pro Windows i pro Linux; zde se podrobněji zmíním o instalaci pro Windows. Z uvedené stránky můžete editor stáhnout ve třech podobách: (a) „full download“, (b) „single file executable called Sc1“ a (c) „windows installer that includes extensions“; pro naše použití doporučuji variantu (b). Konfigurace editoru se děje pomocí textových souborů *.properties; všechny je najdete ve variantě (a), resp. (c). Ve variantě (b) je vše integrované do jediného spustitelného souboru. Jestliže však tato integrovaná varianta najde v předepsaných místech nějaké soubory *.properties, použije je a modifikuje podle nich zabudované nastavení editoru.

Pro naše účely jsem na bázi Sc1 sestavil balíček Sc1_G95.zip, který najdete spolu s ostatními materiály na mém Webu www.physics.muni.cz/~jancely. Stačí ho rozbalit do zvoleného adresáře a ten zapsat do položky SciTe_HOME v souboru Sc.bat, kterým se bude editor spouštět. Najdete-li později na výše uvedené adrese novější verzi Sc1, stačí ji zkopírovat místo Sc171.exe a opravit číslo verze v druhém řádku spouštěcího souboru Sc.bat.

V rozbaleném balíčku Sc1_G95.zip najdete vedle vlastního editoru následující soubory:

- `fortran.api` – obsahuje většinu příkazů (funkcí) Fortranu90(95), které se vám budou při psaní zdrojového v editoru objevovat jako nápověda.
- `fortran.properties` – upravuje zabudovaný soubor `fortran.properties` pro naše potřeby. Aby nastavení odpovídalo vašemu rozložení adresářů a jménu vaší knihovny, musíte v něm *opravit hodnoty (pravé strany) úvodních proměnných* ModDir, LibDir, LibMoje.
- `locale.properties` – zavádí do editoru česká menu.
- `SciTeUser.properties` – mění některá globální nastavení editoru podle vašich požadavků. Zde si např. nastavíte šířku a výšku okna, odsazení od kraje obrazovky atd. Tento soubor přepisuje globální nastavení ze zabudovaného `SciTeGlobal.properties`. Protože tento soubor není při instalaci integrované verze editoru ScXXX viditelný, najdete v balíčku jeho kopii pod jménem `SciTeGlobal.properties_` (pozor, jestliže umažete podtržítka, začne ho editor používat!). Tento soubor se **nikdy neupravuje**. Požadované změny provedete tak, že příslušné položky (které ještě nejsou v nabízeném souboru) do `SciTeUser.properties` a tam se upraví. Podrobnější informace k úpravám najdete formou poznámek přímo v nabízeném souboru. Význam všech položek najdete v manuálu `SciTeDoc.html`, který je také součástí balíčku.

B.2 Spolupráce editoru SciTe s G95

Předpokládejme, že máte provedenu instalaci editoru podle předchozího odstavce. Jestliže v něm napíšeme (nebo do něj přečteme) programový modul (soubor obsahující MODULE, PROGRAM nebo obojí),

můžeme jeho kompilaci, vytvoření spustitelného souboru a případné spuštění provádět přímo z editoru. Použije se k tomu :

Kombinace kláves **Ctrl+F7**

Jestliže je zdrojový text bez syntaktických chyb, proběhne pouze kompilace (s klíčem `-c`) a v pracovním adresáři se vytvoří objektový soubor `*.o`. Je-li v souboru deklarovaný modul (ohraňovaný dvojicí `MODULE <jmeno_modulu> ... END MODULE <jmeno_modulu>`) vytvoří se ještě odpovídající soubor `*.mod`. Ve výstupním okně editoru se objeví vypsaný příkaz a po skončení bezchybné kompilace ještě řádek `Exit:0`. Jestliže zdrojový soubor obsahuje syntaktické chyby, uvedené soubory se nevytvoří a kompilátor vypíše ve výstupním okně editoru informace o chybě následované řádkem `Exit:n` ($n \neq 0$). Přitom je třeba připomenout, že chyba nemusí být vždy ve vypsaném řádku. U složitějších programových konstrukcí je nutné chybu hledat i v oblasti před vypsaným řádkem. Chybu opravíme a opakujeme tento proces tak dlouho, až se objeví očekávané `Exit:0`.

Klávesa **F7**

Zahájí vytvoření spustitelného souboru a má tedy smysl pouze u zdrojových souborů, které obsahují program (úsek `PROGRAM <jmeno_programu>...END PROGRAM <jmeno_programu>`). Připomeňme, že zdrojový text (soubor) může obsahovat pouze jediný program (`PROGRAM`), ale i více modulů (`MODULE`). Pro vytvoření spustitelného souboru však kompilátor (přesněji *linker*) musí mít všechny potřebné objektové soubory a od modulů i soubory `*.mod`. Najít je může v systémových knihovnách kompilátoru (nainstalovaly se s kompilátorem, jejich umístění neudáváme), v knihovně (knihovněch), kterou jste vytvořili z vašich objektových souborů (viz. A.3.2). Běžné ovšem je, že při ladění nového modulu nebo programu jsou nově vytvořené objektové soubory `*.o` v pracovním adresáři. Informaci o nich musíme uvést také v příkazovém řádku (viz. A.3.1). Editor SciTe dovoluje doplňovat příkazový řádek o položky zapsané do okna, které se otevře stiskem kombinace `Shift+F8`. V okně uvidíme čtyři řádky, jejichž obsah se dosadí za proměnné `$(1)`, ... `$(4)`, které najdete v souboru `fortran.properties` v řádcích začínajících `command.compile` (provádí se při stisku `Ctrl+F7`) a `command.build` (při `F7`). Všimněte si, že `$(1)`, `$(2)` jsou před zdrojovým souborem `*.f95` (`*.f90`) a `$(3)`, `$(4)` za ním. Jak víme z A.3.1, poloha některých položek příkazového řádku je důležitá.

Klávesa **F5**

spustí vytvořený spustitelný soubor v novém CMD-okně. Aby se vám toto okno po ukončení programu hned neuzavřelo, nezapomeňte jako poslední příkaz před `END PROGRAM` vložit `READ *` (případně předcházené např. příkazem `PRINT *`, "Konec, stiskni ENTER"), který bude čekat na stisknutí `ENTER`.

C Zobrazení čísel v počítači a jejich typy v F95

D Dodatek D

E Grafika pro G95

V současné době existuje několik relativně schůdných cest pro grafické zobrazení výsledků výpočtů provedených pomocí g95. Dále se podrobněji zmíním o třech z nich.

E.1 Gnuplot

Asi nejjednodušší cestou k získání kvalitních grafů je použití známého programu Gnuplot (<http://www.gnuplot.info/>). Je k tomu jen třeba se seznámit s jeho koncepcí a několika základními příkazy; k tomu vedle podrobného manuálu, který získáte při instalaci programu, najdete na výše uvedené domovské stránce celou řadu odkazů na dostupné texty. Podrobnější seznámení s tímto programem považují za velice vhodné, neboť jeho použití je univerzální.

Na rozdíl od grafických knihoven, které bývají přímou součástí nebo doplňkem některých kompilátorů, nebudete při použití Gnuplotu programovat grafický výstup standardním voláním fortranských procedur. Programem jen vygenerujete potřebná data, uložíte je do souboru (třeba jen dočasně) a ten použijete jako vstup pro nakreslení požadovaných grafů pomocí příkazů Gnuplotu. K výsledku je možné se dostat v podstatě dvojím způsobem:

- spustit gnuplot (pod windows je to wgnuplot) a ručním zadáváním příkazů vytvořit z vygenerovaného datového souboru požadovaný graf. Tato cesta může být vhodná především v době, kdy se učíte používat Gnuplot.
- přímo ve fortranském programu vygenerovat *řídící soubor* (např. graf.gp), který bude obsahovat všechny potřebné příkazy ke kresbě grafu. Vlastní kresbu grafu pak spustíte rovněž z fortranského programu příkazem

```
CALL system("wgnuplot_"graf.gp").
```

Procedura system (rozšíření g95) umožňuje spustit nějaký program přímo ze spuštěného programu, zdvojení uvozovek je nutné má-li příkaz mít požadovaný tvar

```
wgnuplot "graf.gp".
```

Podrobněji je tento postup popsán v [4].

E.2 Grafická knihovna DISLIN

Tuto dlouho vyvíjenou knihovnu najdete na <http://www.mps.mpg.de/dislin/news.html>. Z distribuční stránky ¹⁾ si stáhnete soubor odpovídající vašemu kompilátoru (pro g95 pracující pod windows v prostředí MinGW to bude dl_91_mg.zip) a podrobný manuál ve formátu pdf. Instalaci provedete podle návodu, který je součástí distribuce (soubor readme.gnu). Protože postup použití knihoven doporučený autorem je poněkud odlišný od naší standardní práce s knihovnami, použil jsem následující postup.

- Pro jednoduchou přesnost zkopírujeme autorův zdrojový kód dislin.f90 v adresáři \dislin\mg95 do souboru dislinSP.f95 a v textu tohoto souboru opravíme v prvním a posledním řádku dislin na dislinSP. Pro dvojnásobnou přesnost postupujeme obdobně v adresáři \dislin\mg95\real64 (vytvoříme a opravíme dislinDP.f95). Z obou souborů vytvoříme standardním postupem

¹⁾Rychlé bývá stažení přes ftp na adrese <ftp.gwdg.de/pub/grafik/dislin>.

```
g95 -c dislinSP.f95, resp. g95 -c dislinDP.f95
```

soubory dislinSP.mod, dislinDP.mod (objektové soubory *.o nepotřebujeme) a zkopírujeme je do adresáře, který používáme pro ukládání *.mod souborů (při kompilaci je uváděn za klíčem -I, viz. dod. A).

- Autorovy knihovny dismg7.a, dismg7_d.a zkopírujeme do adresáře v němž ukládáme vlastní knihovny (při kompilaci se uvádí za klíčem -L, viz. dod.A) a přejmenujeme je na libdismgSP.a, libdismgDP.a.

V programu pak stačí uvést

```
USE dislinSP    !pro jednoduchou přesnost
nebo
USE dislinDP    !pro dvojnásobnou přesnost
```

a při překlada (linkování) uvést knihovny

```
-ldismgSP    nebo    -ldismgDP
a
-luser32    -lgdi32 .
```

Posledně jmenované knihovny (libuser32.a, libgdi32.a) jsou součástí standardní instalace g95.

Základní informace o knihovně DISLIN se dozvíte z několika úvodních kapitol manuálu a jednoduché základní použití najdete v testovacích souborech T_dislinX.f95, které jsou v balíku programů pro výuku. Zde bych jen explicitně upozornil na skutečnost, že procedury z knihovny pracují vždy jen v určitých úrovních; je proto třeba si všimnout údaje *level* = 0,1,2,3 u každé procedury. Typický případ úvodní posloupnosti příkazů je např. následující

```
CALL metafl('xwin')           ! level=0, vystup do okna na obrazovce
CALL csrmod('reverse')       ! level=0, cerna kresba na bilem pozadi
CALL x11mod('nostore')       ! level=0, vystup primo na obrazovku
CALL units('cm')             ! level=0, nastavi 100 bodu na 1cm
                              ! nasledujici prikaz provede inicializaci
CALL disini()                 ! level=0, po provedeni je level=1
```

E.3 Grafické prostředí JAPI

Literatura

- [1] *BNF Syntax of the F Programming Language*, The Fortran Company 2003
Původní definice jazyka F; text můžete najít na mé webové stránce
- [2] *F Syntax Rules*, The Fortran Company 2003
Definice jazyka F na bázi G95; text je v dokumentaci po instalaci FORTRANTOOLS
- [3] *G95manual.pdf*.
Najdete ho po instalaci g95 v adresáři doc
- [4] *Fortran Tools*, The Fortran Company, 2005
www.fortran.com/fortran/imagine1/ftools.pdf
- [5] W.S. Brainerd, Ch.H. Goldberg, and J.C. Adams: *Fortran 95 Using F*,
The Fortran Company, 2005
- [6] J.C.Adams, W.S.Brainerd, J.T.Martin and B.T.Smith: *The Key Features of Fortran 95*,
The Fortran Company, 2006
- [7] L.P. Meissner: *Fortran 90 & 95 Array and Pointer Techniques*,
Computer Science Department, University of San Francisco, 1998
- [8] M.Metcalf and J.Reid: *Fortran 90/95 explained*, 2nd ed., Oxford University Press, 2002,
ISBN 0-19-850558-2
- [9] Z.Dodson: *A Fortran 90 Tutorial*, 1993
www.scd.ucar.edu/tcg/consweb/Fortran90/F90Tutorial/tutorial.html