Programovací jazyk ADA

Karel Ježek, Stanislav Racek

Obsah

Kapitola 1

Základní rysy

ADA je jedním z mála vyšších programovacích jazyků umožňujících *multithreading* (multitasking). Pro tento účel je vybavena konstrukcemi *task* a *task type*. Vlákna (tasky) spolupracují mechanismem označeným jako *rendez-vous* (dostaveníčko), pomocí kterého lze zařídit nejrůznější možnosti interakce (synchronizace, sdílení dat, komuniakace pomocí zpráv).

ADA může být provozována jak na jednoprocesorovém, tak i na na víceprocesorovém počítači. Zda výpočet bude probíhat pseudoparalelně či paralelně, závisí na systému a programátor má pouze k dispozici prostředky, jejichž pomocí vyjadřuje potenciálně možný paralelní běh programu.

Základní verze jazyka (dále popisovaná) verze jazyka pochází z roku 1983 **ADA 83**. Tato verze byla v roce 1995 obohacena o prostředky pro objektové programování, čímž vznikla **ADA 95**.

Jazyk je určen k programování rozsáhlých úloh, zejména z oblasti vestavěných RT systémů (pracujících v reálném čase). Je preferovaným jazykem DOD (ministerstvo obrany) USA. Základními hledisky, která **ADA** respektuje jsou:

- Bezpečnost vytvářených programů (zajišťovaná zejména silným typovým systémem),
- možnosti pro týmovou práci programátorů (modulární struktura programu, dovolující separátní kompilaci modulů),
- přenositelnost programů (překladače jsou testovány na soulad se standardem jazyka, tzv. validovány).

Další významné schopnosti zahrnují:

- Vyjadřování paralelně proveditelných akcí,

- možnost programového ošetření výjimečných situací (chybových stavů),
- prostředky pro vytváření datových abstrakcí (tzv. generické programové jednotky – šablony pro vygenerování programů).

Nevýhodou ADY je zejména její rozsáhlost. V roce 1995 došlo k rozšíření jazyka o prostředky OOP (třídy typů a dědičnost) na podobném principu jaký používají i ostatní imperativní jazyky. Poněvadž nemáme k dispozici kompletní podklady ani překladač jazyka ADA95, poznamenejme pouze, že pomocí slova "tagged" je možné označit takový typ, jehož vlastnosti mohou dědit z něj odvozované typy.

Způsob zápisu programu zjevně odráží vliv **Pascal**u. Identifikátory jsou tvořeny posloupností písmen, číslic a znaku podtržení, velikost písmen se nerozlišuje. Klíčová slova jsou vyhražena. V programech je zvykem je psát malými a identifikátory velkými písmeny. Předdefinované identifikátory lze předeklarovat.

K zápisu čísla je možné použít i jiné než desítkové soustavy uzávorkováním čísla znaky #, s předchozím uvedením základu. Pro zlepšení čitelnosti mohou také čísla obsahovat znak podtržení. Např. 2#1001_0110#

Řetězce jsou uzavírány mezi uvozovky, znakové konstanty mezi apostrofy. Poznámky začínají dvěma pomlčkami a pokračují až do konce řádku.

I/O operace nejsou součástí jazyka, ale jsou exportovány z knihovních modulů, případně tyto moduly exportují prostředky pro jejich vygenerování ze šablon (důsledek typového systému).

Program sestává z jedné, příp. z několika tzv. kompilačních jednotek. Kompilační jednotkou bývá modul (nazývaný package) nebo podprogram. Kompilační jednotku předchází tzv. kontextová specifikace. Ta uvádí jména jednotek, jejichž služby kompilační jednotka používá. Hlavní program má podobu bezparametrové procedury. Jeho implicitním vnějším prostředím je modul `STANDARD`, zavádějící předdefinované typy a jejich operace. V kontextové části jednotek se `STANDARD` necituje. Příkladem triviálního programu je následující text, kterým je vypsán obrázek totemu se zadaným počtem tváří.

Příklad 3.4.

```
with TEXT_IO; -- specifikace prostředí (modulu pro I/O textů)
use  TEXT_IO; -- zviditelnění prostředků modulu TEXT_IO
procedure OBRAZEK is
               --deklarační část
   TOTEM:INTEGER:=1;
   package INT_IO is new INTEGER_IO(INTEGER);
                                         --generování modulu
   --pro I/O operace nad typem INTEGER ze šablony INTEGER_IO
   use INT_IO; --zviditelnění prostředků z modulu INT_IO
begin
               --příkazová část
   PUT_LINE("kolik tváří má totem?");
   GET(TOTEM);
   NEW_LINE;
   for F in 1..TOTEM loop
      PUT_LINE("(((||)))");
      PUT_LINE("(*)++(*)");
      PUT_LINE("   ..   ");
      PUT_LINE(" ------");
   end loop;
end OBRAZEK;
```

Program `OBRAZEK` je tvořen jedinou kompilační jednotkou. Používá typ `INTEGER`, který je deklarován v modulu `STANDARD`.

Klauzule **`with`** zpřístupňuje knihovní modul `TEXT_IO` s prostředky pro práci s textovými soubory. Klauzule **`use`** dovoluje v dalším textu používat jmen z uvedeného modulu bez kvalifikace jménem modulu, tj. psát např. `PUT_LINE` namísto `TEXT_IO.PUT_LINE`.

`INTEGER_IO` je generický modul definovaný v modulu `TEXT_IO`, obsahující prostředky pro I/O operace s celými čísly. Uživatel má možnost vytvořit si svůj vlastní celočíselný typ požadovaného rozsahu a vygenerovat si pro něj množinu periferních operací.

Každý příkaz končí středníkem, a proto nelze vynechat středník před **end** (jak je tomu v **Pascal**u).

Kapitola 2

Klasické výrazové prostředky

V této části textu stručně probereme ty jazykové konstrukce, které **ADA** s určitými úpravami převzala od svých předchůdců. Protože tyto konstrukce nepodporují výrazně vytváření rozsáhlých programových celků, bývají označovány souhrnně jako prostředky pro programování v malém (angl. *programming in small*).

ADA používá silný typový systém s tzv. jmennou ekvivalencí. Rozumí se tím, že každá definice typu zavádí typ nový, odlišný od ostatních i v tom případě, kdy mají stejnou strukturu. Nepřipouští se mixovat ve výrazech objekty různých typů bez explicitně vyjádřené smysluplné konverze. Deklarace typu sdružuje jméno typu spolu s jeho definicí.

Proměnným je dovoleno přiřadit při deklaraci inicializační hodnotu, která nemusí být statická, musí být však vyhodnotitelná v okamžiku zpracování deklarace při výpočtu.

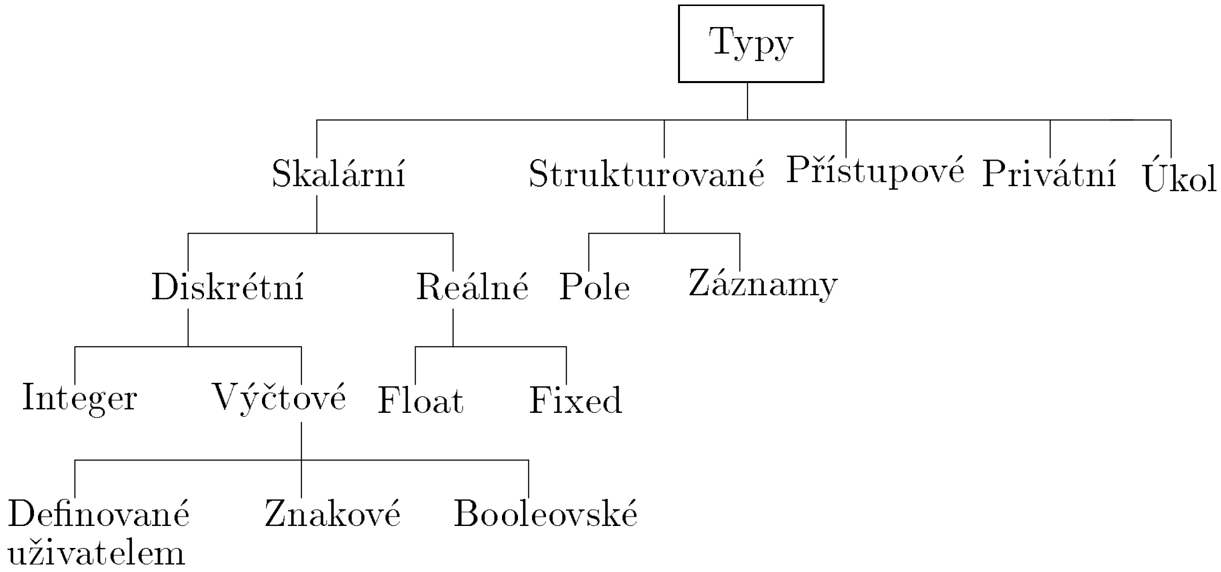
Číselné konstanty mohou být zaváděny bez specifikování jejich typu. Např. bude-li deklarován `type SMER is (NAHORU, DOLU);` pak:

```
PROM_SMER: SMER := NAHORU; -- deklaruje inicializovanou proměnnou
KONST_SMER:constant SMER := NAHORU; -- definuje konstantu typu SMER
PI: constant:=3.141592; -- zavádí konstantu univerzálního typu REAL
DESET: constant := 10; -- zavádí konstantu univ. typu INTEGER
```

Univerzální typ je kompatibilní s libovolným typem odvozeným ze stejné číselné báze.

Údaje o vlastnostech typů v konkrétní implementaci lze získat pomocí atributů. Pomocí atributů je možné ověřovat přenositelnost programů mezi počítači s různou délkou slova (zjištěním mezních hodnot daného typu), určit sousední hodnotu diskrétního typu, rozsah indexu pole apod.

Typový systém jazyka **ADA** je značně rozsáhlý. Klasifikaci typů uvádí obr.3.8.



Obr. 2.1: Klasifikace typů v jazyce ADA

Skalární datové typy

Typ INTEGER

Typ `INTEGER` je předdefinován v modulu `STANDARD`. Spolu s ním může implementace nabízet i `SHORT_INTEGER` či `LONG_INTEGER`.

Z předdefinovaných celočíselných typů lze odvozovat další celočiselné typy, obvykle s rozsahovým omezením hodnot:

```
type INT  is new INTEGER;
type DNY  is range 1..366;
type ROKY is range 0..2000;
```

Tím, že uživatel neuvede, ze kterého předdefinovaného typu odvozuje svůj typ, dává překladači možnost vybrat nejvhodnější, který zaručí požadovanou přesnost.
Deklarujeme-li pak proměnné:

```
CISLO_DNE: DNY;
CISLO_ROKU:ROKY;
```

můžeme je použít v příkazech:

```
CISLO_DNE:=CISLO_DNE+1;
CISLO_ROKU:=ROKY(CISLO_DNE);-- explicitní konverze
```

příkaz `CISLO_ROKU:=CISLO_DNE;` je však chybný a nepřeloží se důsledkem silné typové kontroly.

Z atributů použitelných pro celočíselné typy jmenujme alespoň dva:

`JMENO_TYPU‘FIRST`	udává nejmenší hodnotu udaného typu,
`JMENO_TYPU‘LAST`	udává největší hodnotu udaného typu.

Oba atributy jsou použitelné pro libovolný diskrétní typ.

Výčtový typ

Způsob zavedení je obdobný **Pascal**u s tím rozdílem, že **ADA** dovoluje přetěžování (vícevýznamové použití) literálů. Je tedy možné deklarovat:

```
type BARVA is (bila, zluta, cervena, modra, cerna);
type SEMAFOR is (zelena, oranzova, cervena);
```

Při použití literálu je však nutné buď z kontextu, nebo na základě tzv. kvalifikace jeho typ determinovat. Kvalifikace má tvar:

```
SEMAFOR‘(cervena)
BARVA‘(cervena)
```

Obecně lze kvalifikovat výraz zápisem `JMENO_TYPU‘(výraz)`.

Z atributů, které jsou užitečné pro práci s výčtovým typem (použitelné však pro libovolný diskrétní typ) uveďme:

`PRED(X)`	předchůdce `X`,
`SUCC(X)`	následník `X`,
`POS(X)`	poziční číslo `X`,
`VAL(I)`	hodnota typu určená pozičním číslem `I`.

Např.:	`BARVA‘VAL(0)`	má hodnotu bílá
	`BARVA‘POS(BARVA‘FIRST)`	má hodnotu 0

Typy `BOOLEAN` a `CHARACTER` jsou předdefinovanými výčtovými typy v modulu `STANDARD`.

```
type BOOLEAN is (FALSE, TRUE);
type CHARACTER is (nul, ...,‘A‘,...,‘z‘,...,del);
```

Netisknutelné hodnoty typu `CHARACTER` jsou pojmenovány literály, takže tento typ obsahuje všech 128 ASCII znaků.

Typ FLOAT

Je předdefinovaným typem pro hodnoty v pohyblivé řádové čárce. Implementace může opět obsahovat jeho kratší (méně přesnou), či delší (přesnější) alternativu.

Uživatel má možnost definovat své vlastní typy v pohyblivé čárce. Přesnost uvede počtem míst mantisy (digits). Ten určuje dolní limit přesnosti. Implementací je vybrána nejbližší vyhovující přesnost. Přenositelnost programů je tím zaručena. Počet míst mantisy je zadáván statickým výrazem. Součástí definice může být i uvedení přípustného rozsahu hodnot (range). Např.:

```
type REALNE is digits 4; – neuvedený rozsah hodnot
type PRESNE is digits 10 range 0.0 .. 5.0E3;
```

Předpokládejme existenci typu `F` v pohyblivé čárce. Pak jeho charakteristiky můžeme zjišťovat pomocí atributů:

`F‘DIGITS`	–	počet desítkových číslic mantisy typu `F`,
`F‘MANTISA`	–	počet bitů mantisy typu `F`,
`F‘EPSILON`	–	jemnost zobrazení typu `F`,
`F‘EMAX`	–	nejvyšší hodnota exponentu,
`F‘SMALL`	–	nejmenší zobrazitelné kladné číslo typu `F`,
`F‘LARGE`	–	největší zobrazitelné kladné číslo typu `F`.

Typ Fixed

Není předdefinovaným typem. Uživateli je umožněno definovat si reálný typ v pevné řádové čárce uvedením jeho absolutní přesnosti a rozsahu hodnot. Např.:

```
type VYPLATA is delta 0.01 range 0.00 .. 100_000.00;
```

Hodnoty typu `VYPLATA` budou zobrazovány s přesností jedné setiny. Přesnost typu v pevné řádové čárce umožňuje zjistit atribut `DELTA`.

Komplikace, které způsobuje zvětšení počtu cifer výsledku aritmetických operací nad typy v pevné řádové čárce, je nutno řešit explicitně vyjádřenou konverzí typu. Např. při deklaraci:

```
type P is delta 0.1 range 0.0 .. 1000.0;
A,B,C: P range 0.0 .. 100.0 := 1.5;
```

je nutné konvertovat hodnotu vzniklou násobením `A` s `B` na původní přesnost, chceme-li ji přiřadit `C`.

```
C := P(A*B);
```

Záměr konverze je obecně (pro všechny druhy typů) vyjádřen formou:

```
JMENO_TYPU (KONVERTOVANY_VYRAZ),
```

přičemž převod reálného typu na celočíselný se provádí zaokrouhlením.

Operátory skalárních typů

Předdefinované operátory jsou sumarizovány v tab. 3.1, 3.2 a 3.3. Jejich vlastnosti jsou konvenční s několika výjimkami:

- Pro vyjádření zkráceného výpočtu jsou zavedeny operátory `and then` a `or else`. Takže např. ve výrazu

  ```
  A > B and then C >= 1
  ```

  bude druhá část podmínky vyhodnocována pouze v případě, že prvá část je pravdivá.

- Operátory `rem` a `mod` vytváří zbytek po celočíselném dělení. Jsou-li oba operandy těchto operátorů stejného znaménka, jejich výsledek se neliší. Platí, že výsledek rem má znaménko levého operandu a výsledek mod má znaménko pravého operandu. Např.:

  ```
  2  rem -5 = 2,  2 mod -5 = -3, -1 rem 5 = -1, -1 mod 5 = 4
  ```

- Operátor dělení s celočíselnými operandy dává celočíselný výsledek, takže např. $7/4 = 1$, neboť frakční část je zanedbávána.

- Není dovoleno používat záporný exponent s celým číslem, takže výraz $2**(-5)$ způsobí runtime error.

operátor	operand	výsledek
`+`	numerický	stejný
`-`	numerický	stejný
`not`	boolean	stejný
`abs`	numerický	stejný

Tab.3.1.Unární operátory

operátor	operandy		výsledek
and, or, xor **and then,** **or else**	boolean		stejný
`=, /=`	libovolné		boolean
`<, <=, >, >=`	skalární diskr. jednor. pole		boolean
in,	hodnota interval		boolean
not in	hodnota podtyp		boolean
`+ -`	numerické		stejný
&	řetězce či znaky		řetězec
*	integer	integer	stejný
	integer	fixed	stejný fixed
	float	float	stejný float
/	integer	integer	stejný
	float	float	stejný,
	real	integer	real
mod rem	integer	integer	stejný
**	integer	nezáp. integer	stejný integer,
	float	integer	stejný float

Tab.3.2. Binární operátory

`and`	`or`	`xor`	`and`	`then`	`or`	`else`		
`=`	`/=`	`<`	`<=`	`>`	`>=`	**in**	**not**	**in**
bin.	`+`	`-`						
un.	`+`	`-`						
`*`	`/`							
`**`	`abs`	`not`						

Tab.3.3. Priorita operátorů

Podtypy a odvozené typy

Podtyp reprezentuje podmnožinu nějakého již existujícího, tzv. bázového typu. Oproti oboru hodnot bázového typu bývá jeho obor hodnot zúžen. Užitečnost podtypu spočívá v možnosti mixovat ve výrazech dle něj deklarované proměnné s proměnnými bázového typu bez uvedení konverze. Deklarace podtypu nezavádí nový typ. Kromě omezení má podtyp stejné vlastnosti jako typ bázový. Např.:

```
subtype PRIROZENA_CISLA is INTEGER range 1 .. INTEGER'LAST;
subtype PRACOVNI_DNY is DNY_V_TYDNU range PONDELI .. PATEK;
```

Vlastnosti bázového typu lze zjišťovat prostřednictvím atributu `BASE`. Např. `PRACOVNI_DNY'BASE'LAST` bude mít hodnotu posledního dne v týdnu.

Odvozený typ je podobně jako podtyp získán z již existujícího typu. Zavádí se jím však nový typ, s vlastnostmi zděděnými od bázového typu. Proto ve výrazech nemohou být mixovány objekty bázového typu s objekty odvozeného typu bez explicitně vyjádřené konverze. Tvar jeho deklarace ilustrují příklady:

```
type NOVE_INTEGER is new INTEGER;
type NOVE_FLOAT is new FLOAT digits 5;
```

Druhý příklad má však nevýhodu, neboť nutí překladač odvodit typ z typu `FLOAT`, i když by mohl být např. použit úspornější typ `SHORT_FLOAT`. Z tohoto důvodu je výhodnější dříve zavedená konstrukce

```
type NOVE_FLOAT is digits 5;
```

která překladač neomezuje. I ta však zavádí odvozený typ, neboť dle definice je ekvivalentní deklaracím:

```
type ANONYMNI is new PREDDEFINOVANY_FLOAT;
subtype NOVE_FLOAT is PREDDEFINOVANY_FLOAT digits 5;
```

kde `ANONYMNI` je fiktivním typem a `PREDDEFINOVANY_FLOAT` je některý z implicitně deklarovaných float typů, vybraný překladačem. Z obdobných důvodů jsou zkracovány i konstrukce deklarací odvozených typů z typů celočíselných a reálných v pevné řádové čárce. Např.

```
type VEK is range 0 .. 200;
```

je odvozeným typem z vhodného celočíselného předdefinovaného typu.

Strukturované datové typy

Typ pole

Deklarace typu pole má obecně tvar:
type *JMÉNO_TYPU_POLE* **is array** *(SPECIFIKACE_INDEXŮ)* **of** *TYP_PRVKŮ;*

Je-li v deklaraci typu určen rozsah indexů, nazýváme typ pole **determinovaný** (angl. constrained), není-li určen, nazýváme takový typ pole **nedeterminovaný** (angl. unconstrained).

```
type VEKTOR_10 is array (INTEGER range 1..10) of FLOAT;
                                                -- nebo
type VEKTOR_10 is array (1..10) of FLOAT;--determinovaný typ
type DIÁŘ is array (PRACOVNI_DNY, 1..24) of CO_DELAT;-- " "


type VEKTOR is array (INTEGER range <>) of FLOAT;
                                     -- nedeterminovaný typ
type MATICE is array (INTEGER range <>,INTEGER range <>)
                      of FLOAT;      -- nedeterminovaný typ
```

Pro specifikaci indexů lze použít libovolný diskrétní typ. Nedeterminované typy pole používáme při konstrukci algoritmů zpracovávajících pole s různými rozsahy indexů. Určení rozsahu je odloženo až do okamžiku použití nedeterminovaného typu k deklaraci proměnné, konstanty či podtypu. Např.:

```
subtype VEKTOR_100 is VEKTOR(1..100);
V1, V2: VEKTOR_100;
KONSTANTA: VEKTOR := (0.0, 0.1, 0.4, 10.0, 12.55);
M1: MATICE(1..5, 10..20);
```

Z příkladů je patrné, že v případě konstantních polí je jejich rozměr určen zadáním přiřazované hodnoty.

!! Použijeme-li anonymního typu k deklaraci proměnných, např.

```
P, Q: array(1..2, 1..5);
```

budou `P` a `Q` odlišného typu, neboť uvedený zápis je dovoleným zkrácením dvou deklarací s anonymními typy.

Meze polí nemusí mít vždy statický charakter, mohou být dány výrazem vyhodnotitelným až v době výpočtu. Taková pole jsou nazývána *dynamická*. Dovolují ekonomicky využívat paměť v těch případech, kdy předem rozměry pole neznáme.

Vlastnosti polí je možné zjišťovat v programu pomocí atributů. Pro objekt `A` typu pole, či typ `A` s určenými mezemi udávají:

`A'FIRST`	spodní mez prvního indexu,
`A'LAST`	horní mez prvního indexu,
`A'LENGTH`	počet hodnot prvního indexu,
`A'RANGE`	interval `A'FIRST..A'LAST`

Pro zjištění charakteristik n-tého rozměru jsou použitelné zápisy

```
A'FIRST(n), A'LAST(n), A'LENGTH(n), A'RANGE(n).
```

Řetězce jsou považovány za nedeterminovaná jednorozměrná pole znaků. V modulu `STANDARD` je pro ně předdefinován typ `STRING`.

```
type STRING is array (POSITIVE range <>) of CHARACTER;
```

Operátor zřetězení `&` je aplikovatelný na libovolná jednorozměrná pole. Dolní mez výsledného pole je dána dolním indexem levého operandu.

Pole stejného rozsahu a typu prvků lze přiřazovat jako celek, což platí i pro řetězce. Řetězce s proměnnou délkou předdefinovány nejsou.

V případě jednorozměrných polí lze přiřazovat i jejich řezy. Řezem se rozumí souvislá část původního pole. Pro výše deklarované proměnné je např. přípustný zápis:

```
V1 := V2;
V1(10..25) := V2(41..56);
```

Polím je možné přiřazovat složenou hodnotu, tzv. **agregát**. Agregáty mohou být **poziční** nebo **jmenné**. V prvém případě jsou jednotlivé hodnoty uváděny postupně (viz výše uvedená `KONSTANTA`), ve druhém případě jsou hodnoty prvku sdruženy s hodnotou indexu.
Např.

```
M: MATICE(1..2,1..2) := ((1.1, 1.2), (2.1, 2.2)); -- poziční
M := (1=>(1=>1.1, 2=>1.2),
      2=>(1=>2.1, 2=>2.2));    -- jmenný agregát
```

Obě formy je povoleno mixovat, avšak ne v jednom rozměru. Další možnosti zápisu agregátů uvedeme formou příkladů.

```
M := (1..2=>(1..2=>0.0));
MUJ_DIAŘ:=(PONDELI|PATEK=>(1..8=>SPÁT,others=>CESTOVAT),
          others=>(1..6=>SPAT,7..22=>PRACOVAT,others=>SPAT));
```

Pokud je použita alternativa `others`, musí být uvedena v daném rozměru jako poslední. Při jejím použití musí být rozsah pole patrný z kontextu, což lze zajistit např. kvalifikací jménem typu

```
JMENO_TYPU‘ agregát
```

Na jednorozměrná booleovská pole jsou aplikovatelné operátory

not **or** **xor**

Jednorozměrná diskrétní pole lze porovnávat relačními operátory, což je nejčastěji použitelné pro řetězce. Platí např.

```
"AA" < "B"     "A " > "A"     " Z" < "A"
```

Budeme-li deklarovat typ pole tak, že typ jeho prvků je opět typem pole, vzniká pole polí. Např. matici lze vytvořit jako vektor vektorů. Na prvky takového pole pak budeme odkazovat zápisem

```
JMENO_POLE (INDEX1) (INDEX2) .
```

Tento zápis není zaměnitelný s `JMENO_POLE (INDEX1, INDEX2)` při odkazu na vícerozměrné pole.

Záznamy

Pro práci se záznamy jsou použity konvence známé z **Pascal**u či **C** jazyka. Deklarace záznamu bez variantní části má např. tvar:

```
type DATUM is
      record
            DEN: INTEGER range 1..31;
            MESIC: JMENO_MESICE;
            ROK: INTEGER range 1900..2000;
      end record;
type KOMPLEXNI is
      record
            RE: FLOAT := 0.0;
            IM: FLOAT := 0.0;
      end record;
```

Těchto typů pak lze využít k deklaraci proměnných:

```
VCERA, DNES, ZITRA: DATUM;
X, Y: KOMPLEXNI;
```

Proměnné `X` a `Y` budou inicializovány počáteční hodnotou, uvedenou v deklaraci typu. Inicializaci lze provést i při deklaraci proměnných prostřednictvím pozičního či jmenného agregátu.

```
Z: KOMPLEXNI := (1.0, 5.0);           -- poziční notace
NAROZEN: DATUM := (ROK=>1993, DEN=>25, MESIC=>LISTOPAD);
                                      --jmenná notace
```

Na komponenty záznamu se odkazuje pomocí tečkové notace. Např.

```
NAROZEN.ROK := 1989;
```

Předdefinovanými operacemi pro záznamy jsou přiřazení a testy na rovnost a nerovnost.

Takové typy záznamů, které mají pevně zadaný počet, typ a velikost komponent, jsou nazývány *typy záznamů bez diskriminantů*. Oproti tomu tzv. *typy záznamů s diskriminanty* dovolují vytvářet záznamy:

a) s komponentami pole různé délky,

b) s variantní částí.

ad a)
Je-li komponentou záznamu nedeterminovaný typ pole, lze pomocí parametru (diskriminantu) určit velikost pole. Např.

```
type VYROVNAVACI_PAMET (ROZSAH: CELOCISELNY_TYP := 100) is
      record
            PAMET: STRING(1..ROZSAH);
      end record;
```

Poněvadž při deklaraci typu VYROVNAVACI_PAMET jsme uvedli předběžnou hodnotu diskriminantu 100, nemusíme při deklaraci proměnné tohoto typu hodnotu diskriminantu již uvádět. Pokud jej uvedeme, bude tato nová hodnota akceptována.

```
VPAMET1: VYROVNAVACI_PAMET; --rezervované místo pro 100 znaků
VPAMET2: VYROVNAVACI_PAMET(50);--  "      "     "    50   "
VPAMET3: VYROVNAVACI_PAMET(ROZSAH=>10);     "     "    10   "
```

Pokud by v deklaraci typu nebyla předběžná hodnota uvedena, je uvedení hodnoty diskriminantu povinné při deklaraci záznamu. Diskriminant je ve skutečnosti jednou z položek záznamu. Nelze mu však samostatně přiřazovat hodnotu. Těm záznamům, které jsou deklarovány bez uvedení hodnoty diskriminantu, je možné v rámci přiřazení celému záznamu přiřadit i hodnotu diskriminantu. Např.

```
VPAMET1 := (400,"ahoj");
```

Takové záznamy jsou nazývány nedeterminované.

!! Opatrně používat, řada překladačů si zjednodušuje práci tím, že rezervuje paměť pro největší možný záznam, který může vzniknout (v našem příkladu s polem o rozsahu 1..CELOCISELNY_TYP'LAST).

Záznamy, které jsou deklarovány s uvedením hodnoty diskriminantu se označují **determinované záznamy**.

Pro zjištění, zda záznam je či není determinovaný je k dispozici atribut CONSTRAINED, nabývající pro

VPAMET1'CONSTRAINED hodnotu FALSE a pro
VPAMET2'CONSTRAINED hodnotu TRUE.

Diskriminantů, kterými je záznam parametrizován může být libovolný počet (např. je-li komponentou záznamu vícerozměrné pole, lze jimi parametrizovat libovolný počet dimenzí).

ad b)

Variantní záznamy dovolují sdružit několik alternativních záznamů do jednoho celku. Mohou být rovněž determinované či nedeterminované, na základě určení či neurčení hodnoty diskriminantu při deklaraci záznamu. Variantní část musí být jediná, uvedená v záznamu jako poslední. Necitované alternativy variantní části je možné zahrnout do části označené **others**. Např.

```
type TYP_STAVU is (SVOBODNY, ZENATY);
type OSOBA (STAV: TYP_STAVU := SVOBODNY) is
      record
            NAROZEN: DATUM;
            VAHA, VYSKA: FLOAT;
            case  STAV is
                  when SVOBODNY => null; --prázdná položka
                  when ZENATY   => DATUM_SNATKU: DATUM;
                                   POCET_DETI: INTEGER;
            end case;
      end record;
```

Pak je možné deklarovat záznamy:

```
OSOBA1: OSOBA;        --nedeterminovaný záznam
OSOBA2: OSOBA(ZENATY);--determinovaný záznam
OSOBA3: OSOBA:=(SVOBODNY,(17,LISTOPAD,1989),1.55,38.5);
        --nedeterminovaný, inicializovaný pozičním agregátem
```

Stejně jako v případě a) je možné přiřazovat determinovaným záznamům jako celku pouze takové agregáty, které mají souhlasnou hodnotu diskriminantu a nedeterminovaným záznamům i agregáty, které hodnotu diskriminantu mění. Diskriminant může být použit i pro deklaraci podtypu. Např.

```
subtype MLADENEC is OSOBA(SVOBODNY);
```

Dle něj deklarované záznamy budou pak determinované. Důvodem výše uváděných konstrukcí je zajištění bezpečnosti konstrukcí se záznamy. Nepromyšlená změna samotného diskriminantu (pokud ji jazyky připouští – viz **Pascal**) způsobí nesmyslnost celého obsahu variantní části.

Typ ukazatel

Deklarace typu ukazatel má v jazyku **ADA** tvar:

```
type JMENO_TYPU_UKAZATEL is access JMENO_TYPU_NA_KTERY_UKAZUJE;
```

Např.
```
type I_UKAZ is access INTEGER;
UK1, UK2: I_UKAZ; -- deklarace ukazatelových proměnných
```

Pro vytvoření paměťového místa zpřístupněného ukazatelem je nutno použít alokátoru **new**.

Např.
```
UK1 := new INTEGER'(10); -- UK1 ukazuje na hodnotu 10
UK2 := new INTEGER; -- UK2 ukazuje na místo pro INTEGER
```

Existence paměťového místa trvá, pokud výpočet neopustí rozsahovou jednotku, ve které je deklarován ukazatel, který místo zpřístupňuje. Poté může čistič paměti provést jeho zrušení. Brání se tím opomenutí uvolnění paměti programátorem. Pokud však by chtěl sám řídit uvolňování paměti, může si vytvořit z knihovního podprogramu uvolňovací proceduru příkazem:

```
procedure FREE is new UNCHECKED_DEALLOCATION
      (JMENO_UVOLNOVANEHO_TYPU, JMENO_UKAZATELOVEHO_TYPU);
```

Provedením příkazu `FREE(UK);` způsobí pak dosazení prázdné hodnoty do `UK` a uvolnění paměti kam `UK` ukazoval.

Hodnota proměnné, na kterou ukazatel odkazuje, je dosažitelná kvalifikací **all**.

```
UK2.all := UK1.all+1;--přiřadí do místa kam ukazuje UK2 hodnotu 11
UK2 := UK1; --          UK2 ukazuje do stejného místa jako UK1
```

Nejčastějším použitím ukazatelů jsou odkazy na záznamy. Je-li komponentou záznamu opět ukazatel, mohou vznikat rekurzivní struktury. Jelikož je požadováno, aby všechny typy byly deklarovány před použitím, je takový typ uveden formou neúplné deklarace, a jeho úplná deklarace musí být provedena v téže deklarační části.

```
type UDAJ;  --neúplná deklarace
type SPOJ is access UDAJ;
type UDAJ is
      record
            HODNOTA: INTEGER;
            PREDCHUDCE,NASLEDNIK: SPOJ;
      end record;
```

Literálem **null** se označuje prázdná hodnota ukazatele. Budou-li např. deklarovány proměnné

```
ZACATEK, DALSI: SPOJ := null;
```

provede následující sekvence příkazů zařazení dalšího prvku na začátek seznamu:

```
ZACATEK := new UDAJ'(22, null, null);
DALSI   := new UDAJ'(55, null, ZACATEK);
ZACATEK.PREDCHUDCE := DALSI; --dereference neni expl.označena
ZACATEK := DALSI;
```

Z příkladu vyplývá, že **ADA** nepoužívá explicitně vyjádřenou dereferenci (viz **Pascal**), kromě případu odkazování na celý obsah zpřístupňované struktury pomocí kvalifikace `all`, jako např. v příkazu

```
DALSI.all := ZACATEK.all;
```

Základní příkazové struktury

Přiřazovací příkaz

Při zpracování přiřazovacího příkazu je kontrolován jak soulad typů levé a pravé strany, tak i splnění rozsahových omezení. V případech, které mají smysl, je možné zajistit kompatibilitu explicitně vyjádřenou konverzí hodnoty pravé strany. Existuje i možnost provedení konverze ve smyslu bitových posloupností, za pomoci speciální knihovní funkce. Její používání by však mělo být výjimečné.

Přiřazovat je možné i strukturám a jejich částem, jak bylo již dříve uvedeno.

Podmíněný příkaz

Obecný tvar podmíněného příkazu je:

> `if` *podmímka1* `then`
>   *posloupnost_příkazů1;*
> `elsif` *podmínka2* `then`
>   *posloupnost_příkazů2;*
> `elsif` ...
> ...
> `else` *posloupnost_příkazůN;*
> `end if;`

Části elsif a else jsou nepovinné. Význam zápisu je zřejmý.

Příkaz case

Formou je podobný variantním záznamům. Obecně má tvar:

> `case` *výraz* `is`
>   `when` *výběr1 => posloupnost_příkazů1;*
>   `when` *výběr2 => posloupnost_příkazů2;*
>   ...
> `end case;`

Výraz musí nabýt hodnotu diskrétního typu. Každý z výběrů představuje jednu nebo více statických hodnot tohoto typu. Např.

```
case DNES is
     when PONDELI => ODJEZD_NA_PRACOVISTE; PRACUJ;
     when UTERY..CTVRTEK => PRACUJ;
     when PATEK => PRACUJ; ODJEZD_DOMU;
     when SOBOTA | NEDELE => null;
end case;
```

Obsahuje-li výběr několik nesouvislých hodnot, jsou tyto odděleny znakem |. Prázdný příkaz je ve všech konstrukcích označen `null`. Neuvedené alternativy lze shrnout pod výběr `others`, který může být uveden pouze jako poslední. Poněvadž výběry musí pokrýt plně obor hodnot nabývaných výrazem, je konstrukce `when others` pro některé případy dokonce povinná.

Příkazy cyklu

Kromě nekonečného cyklu je zaveden i cykl **while** a cykl **for**. Jejich podobu ilustrují následující příklady.

```
loop        --nekonečný cykl
      VSTUP(DATA);   -- za loop následuje
      ZPRACUJ(DATA); --   posloupnost
      VYSTUP(DATA);  --   příkazů
end loop;

while DALSI /= null  loop
      SUM := DALSI.HODNOTA;
      DALSI := DALSI.NASLEDNIK;
end loop;

for I in A‘RANGE  loop
      A(I) := B(I) + C(I);
      B(I) := 0; C(I) := 0;
end loop;
```

Poznámky k cyklům:

- Výstup z nekonečného cyklu je možný buď pomocí příkazu skoku, nebo příkazem `exit` . Ten rovněž tvrdě ukončí cykl, jeho účinek je však možné vázat na splnění podmínky, zapíše-li se ve tvaru:

  ```
  exit when {\em podmínka;}
  ```

  Příkazem `exit` lze ukončit provádění i ostatních druhů příkazu cyklu. Za normálních okolností ukončuje vždy nejvnitřnější cykl. Pokud však cykl označíme jménem, které pak uvedeme v příkazu `exit`, dojde k ukončení pojmenovaného cyklu.

  ```
  HLAVNI_CYKL: loop ...
                   for ... loop
                         ...
                         exit HLAVNI_CYKL when PODMINKA;
                         ...
                   end loop;   ...
               end loop HLAVNI_CYKL;
  ```

- Parametr cyklu **for** je implicitně zaveden překladačem, nedeklaruje se. Mimo cykl neexistuje. Jeho typ si překladač zjistí z kontextu. Parametru cyklu nelze programem změnit hodnotu. Pokud je rozsah provádění cyklu prázdný, považuje se provádění za ukončené. Rozsah, určující počet průchodů je vyhodnocován jen jednou, před začátkem provádění cyklu. Je určen libovolným diskrétním rozsahem, například intervalem -1 ... 99. Pro reverzní provádění cyklu je zavedena konstrukce:

  `for` *parametr* `in reverse` *diskrétní_rozsah* `loop` ...

Příkaz skoku

Příkaz skoku je zaveden v obvyklé formě

`goto` *jméno_návěští*;

Není přípustné provádět skok dovnitř jazykových konstrukcí, tedy ani do strukturovaných příkazů. Skok ven z konstrukcí povolen je, kromě speciálních případů (z programové jednotky, z kritické sekce, ze sekvence zpracování výjimečné situace). Označení cílového místa je prováděno zápisem

`<<TO_JE_NAVESTI>>` *příkaz;* ...

Kapitola 3

Bloky, procedury, funkce

Bloky, procedury i funkce představují rozsahové jednotky, tj. takové části programu, které sestávají z deklarační části a z příkazové části. Později zavedeme ještě jejich nepovinnou část pro zpracování výjimečných situací.

Deklarační část obsahuje deklarace:

- typů a podtypů,
- proměnných,
- konstant,
- procedur a funkcí,
- modulů (viz později),
- úkolů (viz později).

Příkazová část musí explicitně obsahovat alespoň jeden příkaz. Může jím být i prázdný příkaz **null.**

Bloky

Na rozdíl od procedur a funkcí, které jsou umístěny mezi deklaracemi, jsou bloky součástí příkazů. Není dovolen jejich separátní překlad. Slouží k zavedení lokálních objektů nebo k vyjádření zpracování výjimek. Blok má obecně tvar:

```
declare
    deklarace
begin
    příkazy
end;
```

Deklarační část může být včetně slova declare vynechána. Jeho použití by pak mělo smysl jen při uvedení části s reakcemi na výjimky.

Blok může být pojmenován (obdobně jako cykl). Motivem je umožnit zviditelnění globálních jmen, majících shodné jméno s lokálními jmény. Např.

```
VNEJSI: declare
             X: INTEGER;
        begin
             declare
                   X: TYP_T;
             begin
                   X:= ...      --vnitřní X
                   VNEJSI.X:=...--vnější X
             end;
        end VNEJSI;
```

Procedury a funkce

Spolu s moduly, úkoly a generickými jednotkami (budou probrány později) patří procedury a funkce mezi tzv. kompilační jednotky, které mohou být samostatně překládány. Základní vlastnosti procedur i funkcí se shodují s běžnými zvyklostmi. Řada vlastností je ale nekonvenční, v jiných jazycích nezavedená.

Základní tvar procedury a funkce vyplývá z následujících dvou příkladů.

```
procedure RAZENI(X: in out VEKTOR) is
      POM: FLOAT;
begin
      for I in X‘FIRST+1..X‘LAST loop
            for J in reverse I..X‘LAST loop
                  if X(J-1) > X(J) then
                        POM:=X(J);
                        X(J):=X(J-1); X(J-1):=POM;
                  end if;
            end loop;
      end loop;
end RAZENI; --pro čitelnost se doporučuje zopakovat jméno
```

```
              --zároveň se tím zvyšuje bezpečnost

function SKALARNI_SOUCIN(A,B: VEKTOR) return FLOAT is
      SUMA: FLOAT := 0.0;
begin
      for I in A'RANGE loop
            SUMA := SUMA + A(I) * B(I);
      end loop;
      return SUMA;
end SKALARNI_SOUCIN;
```

Příklady předpokládají deklarovaný typ

```
type VEKTOR is array (INTEGER range <>) of FLOAT;
```

což umožní použít podprogramy pro pole s různými počty prvků.

Příkaz return

Příkaz `return` následovaný výrazem slouží k výstupu z funkce. Je však použitelný i k výstupu z procedury v podobě prostého zápisu `return`. Častější je ukončování procedury průchodem výpočtu koncovým `end`.

Parametry

Parametry mohou být vstupní, výstupní, vstupně-výstupní (`in`, `out`, `in out`). Není-li druh parametru explicitně uveden, předpokládá se `in`.

Všechny parametry funkcí musí být `in`. Brání se tím používání vedlejších efektů funkcí.

Skutečné parametry druhu `out` a `in out` musí být proměnné. Typy korespondujících si formálních a skutečných parametrů musí být stejné.

Při volání jsou hodnoty skutečných parametrů druhu `in` a `in out` kopírovány do formálních parametrů. Při výstupu jsou hodnoty formálních parametrů druhu `out` a `in out` kopírovány do skutečných parametrů.

Při volání je možné uvést parametry nejen v obvyklé poziční notaci, ale také ve jmenné notaci, či v jejich kombinaci (pak musí poziční předcházet jmenné). Např.

```
...SKALARNI_SOUCIN(B => SK_P_2, A => SK_P_1);
...SKALARNI_SOUCIN(SK_P_1, B => SK_P_2);
```

Parametr může mít libovolný typ. Bezparametrová funkce se zapisuje bez závorek.

Parametry `in` lze inicializovat předběžnou hodnotou. Skutečný parametr pak nemusí být uveden. Je-li uveden, předběžná hodnota se neuplatní. Takový postup nelze doporučit, snižuje přehlednost.

Separace

Pro dosažení větší volnosti v uspořádání programového textu je možné oddělit záhlaví podprogramu. To je nazýváno specifikační částí, poněvadž specifikuje všechny vnější znaky podprogramu. Oddělení specifikační části od těla podprogramu je používáno při separátním překladu podprogramu a k přehlednějšímu použití rekurzivních podprogramů. Pro případ funkce znázorňuje separaci příklad:

```
function F(X: TYPX) return TYPF; -- specifikační část funkce

function F(X: TYPX) return TYPF is --tělo funkce
      --lokální deklarace
begin
      --příkazy
end;
```

Separací specifikace od těla zabráníme hlubokému vnořování rekurzivních podprogramů obdobně jako příkazem `FORWARD` v **Pascal**u.

Dalším přínosem separace je možnost použití tzv. podjednotek (angl. *subunits*). Ty se uplatní zejména v případě rozsáhlého textu podprogramu. Důsledkem dlouhé deklarace je jednak nepřehlednost (podprogram je volán z místa textově vzdáleného jeho deklaraci), ale i zdlouhavý překlad a nemožnost program dokončit při neznámých podrobnostech podprogramu. Podjednotka je tvořena separátně překládaným tělem podprogramu, který je v místě, kde by měl být normálně deklarován, nahražen tzv. zbytkem tvaru:

specifikace_podprogramu `is separate;`

Podjednotka má podobu samostatného souboru, tvořeného tělem podprogramu, kterému předchází klauzule

`separate` *(jméno_rodičovské_jednotky)*

Např.

```
procedure MAIN;         --samostatný soubor
       --deklarace konstant, typů a proměnných

       procedure P_SEPAROVANA is separate;
       function F_SEPAROVANA return TYP is separate;

begin
       --prikazy MAIN
end MAIN;
-------------------------------------------------------

separate (MAIN)         --samostatný soubor
procedure P_SEPAROVANA is
       --deklarace konstant, typů a proměnných procedury
begin
       --příkazy procedury
end P_SEPAROVANA;
--------------------------------------------------------

separate (MAIN)         --samostatný soubor
function F_SEPAROVANA is
       --deklarace konstant, typů a proměnných funkce
begin
       --příkazy funkce
end F_SEPAROVANA;
```

Poznamenejme, že separovaný podprogram může rovněž ve svých deklaracích obsahovat další separovaný podprogram. Ten by pak měl na svém začátku klauzuli tvaru např.:

```
separate(MAIN.P_SEPAROVANA)
```

Pro oddělený překlad podprogramů existuje ještě další možnost. Vytvořit z podprogramu samostatnou kompilační jednotku – soubor a v té jednotce, kde je používán, jej uvést v klauzuli `with`. V tomto případě je nutné překládat podprogram před překladem jednotky, která jej používá. Při použití konstrukce `separate` je tomu naopak.

Funkční hodnoty

Funkce mohou vracet i hodnoty, které jsou polem nebo záznamem. Např.

```
procedure P is
      type VE is array (INTEGER range <>) of INTEGER;
      R4, V4: VE(1 ..4);
      R6, V6: VE(1 ..6);
      function ARRAY_ABS(V: VE) return VE is
            R: VE(V‘RANGE);
      begin
            for J in V‘RANGE loop
                  R(J) := abs V(J);
            end loop;
            return R;
      end ARRAY_ABS;
begin
      V4 := (-1, 9, 5, 0);
      V6 := (1, 5, 21, 4, -81, 7);
      R4 := ARRAY_ABS(V4);
      R6 := ARRAY_ABS(V6);
end P;
```

Přetěžování

Přetěžováním podprogramů se míní použití stejného jména pro různé podprogramy. Důvodem je umožnit, aby logicky stejné funkce a procedury mohly mít stejné jméno. Např.:

```
function  PRIDEJ(F: FRONTA; P: PRVEK) return FRONTA is ...
procedure PRIDEJ(A: in out POLE; H: HODNOTA;KAM: INDEX) is...
procedure PRIDEJ(S: in out SEZNAM; P: PRVEK;KAM: MISTO) is...
```

Při volání těchto podprogramů je z kontextu (dle typů a počtu parametrů či typu funkční hodnoty) rozhodnutelné, který z podprogramů je volán. K odlišení je používáno i pořadí parametrů (při poziční notaci), či jmen formálních parametrů (při jmenné notaci).

Podobně jako podprogramy je dovoleno přetěžovat i operátory. Programátor si tak může zavést operátory pro své vlastní typy. Musí však dodržet

zásadu, že přetěžovat lze jen ty operátory, které jsou v jazyku **ADA** zavedeny a jejich aritu musí zachovat. Přetížení je vyjádřeno deklarováním funkce, jejímž jménem je operátor.
Např.

```
function "+"(A,B: VEKTOR) return VEKTOR is
      VYSLEDEK: VEKTOR;
begin
      for I in A'RANGE loop
            VYSLEDEK(I) := A(I) + B(I);
      end loop;
      return VYSLEDEK;
end "+";
```

Takto zavedenou funkci pak můžeme použít ke sčítání vektorů buď ve funkční notaci:

```
VEKTOR1 := "+"(VEKTOR2, VEKTOR3);
```

nebo v názornější notaci operátorové:

```
VEKTOR1 := VEKTOR2 + VEKTOR3;
```

Kapitola 4

Moduly

Moduly jsou prostředkem pro vytváření rozsáhlých programových celků. Umožňují sdružovat programové a datové prvky (procedury, funkce, typy, konstanty ...) do samostatně přeložitelných knihovních souborů. **ADA** je nazývá **package**. Již v dřívějším výkladu jsme se setkali např. s modulem `TEXT_IO`. Nemá sice podobu souboru, neboť je součástí prostředí jazyka **ADA**. Jeho používání i formální popis jsou však stejné jako těch modulů, které si programátor vytváří sám. Moduly mohou být také použity jako prostředek řízení viditelnosti jmen a vnořovány do jiných rozsahových jednotek. Tento způsob využití je však méně častý.

Základní vlastnosti modulů

Stejně jako procedury a funkce, patří i moduly mezi programové jednotky. Všechny programové jednotky mohou mít specifikační část a tělo. Specifikační část modulu uvádí prostředky, které modul zveřejňuje pro použití svým klientům (exportuje). Tělo obsahuje nezveřejňované implementační detaily exportovaných služeb, s nimi související lokální deklarace a případně i inicializační část. Inicializační část je umístěna před závěrečným end modulu a je provedena v okamžiku zpracování modulu v době výpočtu.

Příklad 3.5 Modul exportující zásobník

```
package STACK1 is            --specifikační část modulu
  subtype STACK_ITEM is INTEGER range -10_000..10_000;
  procedure PUSH(ITEM:STACK_ITEM);
  function POP return STACK_ITEM;
end STACK1;                  --konec specifikační části
```

```
package body STACK1 is        --tělo modulu
  STACK_SIZE: constant INTEGER := 1_000;
  STACK_ARRAY: array(1..STACK_SIZE) of STACK_ITEM;
  TOP: INTEGER range 0..STACK_SIZE;

  procedure PUSH(ITEM: STACK_ITEM) is
  begin
    TOP := TOP + 1;
    STACK_ARRAY(TOP) := ITEM;
  end PUSH;
  function POP return STACK_ITEM is
  begin
    TOP := TOP - 1;
    return STACK_ARRAY(TOP + 1);
  end POP;

begin                                --inicializační část
  TOP := 0;
end STACK1;                          --konec těla modulu
```

Klientem, který využívá služeb modulu `STACK1`, by mohl být např. hlavní program ve tvaru:

```
with STACK1,TEXT_IO; use STACK1,TEXT_IO;
procedure STACK1H is
  package STACK_IO is new INTEGER_IO(STACK_ITEM);
  use STACK_IO;
  X,Y: STACK_ITEM;

begin
  PUSH(5);
  X := 4;
  PUSH(X);
  PUT(POP);  --výstup 4
  Y := POP;  --výstup 5
  PUT(Y);    --výstup 5
  NEW_LINE;
```

```
end STACK1H;
```

Služby poskytované modulem `STACK1` jsou nedokonalé. Exportuje pouze jeden zásobník, neošetřuje překročení jeho rozsahu, jeho velikost je statická, typ prvků v zásobníku je neměnný. Možnosti jeho zlepšení uvedeme jako příklady dále probíraných jazykových konstrukcí.

Specifikační část modulu může být buď překládána s tělem společně (v jednom souboru), nebo odděleně. Pokud je překládána odděleně, musí její překlad předcházet překladu modulu klienta. Tělo modulu neovlivňuje propojení na uživatele, může být překládáno kdykoliv později a změna těla nevyžaduje rekompilaci modulů využívajících jeho služeb.

Specifikace kontextu

Moduly mohou ke své činnosti používat služeb jiných modulů. Ty, příp. jejich specifikační části, pak musí být v procesu překladu zpracovány dříve. V tomto smyslu existuje nad kompilačními jednotkami částečné uspořádání.

Jména modulů, ze kterých jsou importovány do modulu zdroje, musí být uvedena na začátku modulu formou klauzule.

`with` *jména_kompilačních_jednotek*;

Ta způsobí vyhledání příslušných jednotek v knihovně. Obvykle je následována use klauzulí:

`use` *jména_kompilačních_jednotek*;

která zviditelní importované zdroje a umožní jejich používání bez kvalifikace jménem modulu z něhož pochází. V případě, že některá jména nebudou tímto způsobem jednoznačně určena, je možné i nadále používat kvalifikace tečka-notací (*jméno_modulu.jméno_zdroje*).

Klauzule **with** nemá tranzitivní účinek.

Privátní typy

Pokusme se modifikovat program z př. 3.5 tak, aby modul exportoval typ zásobník a dovolil tím uživateli pracovat s více než jedním zásobníkem. Specifikační část by pak navíc obsahovala popis

```
type STACK_TYPE is
      record
            STK: array(1..1000) of STACK_ITEM;
            TOP: INTEGER range 0..1000;
      end record;
```

Uvedení této struktury ve specifikační části má však za následek nejen možnost deklarovat ve všech modulech exportujících `STACK1` libovolný počet zásobníků, ale i zpřístupnění vnitřní struktury zásobníku, což je nežádoucí. Zásobník by měl být chráněn proti destrukci ukrytím jeho interní podoby. Jediné uživateli povolené operace s ním by měly být ty, které pro něj modul exportuje.

ADA dovoluje znepřístupnit interní detaily typu protřednictvím tzv. privátních typů. Deklarace privátního typu má tvar:

a) `type` *jméno_typu* `is private`;

b) `type` *jméno_typu* `is limited private`;

Skutečná podoba privátního typu je popsána v privátní části, která je uváděna na konci specifikace modulu a od veřejné části je oddělena slovem **private**. Její přítomnost ve specifikační části je potřebná pro možný překlad uživatelů modulu dříve než těla modulu.

Nad privátními typy má uživatel možnost provádět modulem exportované operace a v případě a) ještě:

- přiřazování,
- test na rovnost,
- test na nerovnost.

Použití privátního typu ukazuje příklad 3.6. Exportovaný typ zásobníku je typu záznam s diskriminantem, který dovoluje určit velikost zásobníku. Uživateli jeho struktura přístupná není, je označen **limited private**, a proto všechny s ním manipulující operace jsou exportovány modulem.

Příklad 3.6 Modul exportující typ pro zásobník

```
package STACK2 is                   --specifikační část
  type STACK_TYPE(STACK_SIZE: NATURAL) is limited private;
  function IS_EMPTY(S:in STACK_TYPE) return BOOLEAN;
  function IS_FULL(S:in STACK_TYPE) return BOOLEAN;
  procedure PUSH(S: in out STACK_TYPE; ITEM:in INTEGER);
  procedure POP(S: in out STACK_TYPE; ITEM:out INTEGER);
  function "="(LEFT,RIGHT:in STACK_TYPE) return BOOLEAN;
  procedure COPY(FROM:in STACK_TYPE; TO:out STACK_TYPE);
  procedure CLEAR(S:in out STACK_TYPE);

private         --pro uživatele nepřístupná privátní část
  type INTEGER_ARRAY is array(NATURAL range<>)of INTEGER;
  type STACK_TYPE(STACK_SIZE: NATURAL) is
    record
      STK: INTEGER_ARRAY(1..STACK_SIZE) := (others=>0 );
      TOP: NATURAL := 0;
    end record;
end STACK2;                  --konec specifikační části
------------------------------------------------------------

with TEXT_IO; use TEXT_IO;      --tělo modulu, samostatně
                                --přeložitelné
package body STACK2 is
  function IS_EMPTY(S:in STACK_TYPE) return BOOLEAN is
  begin
    return S.TOP = 0;
  end IS_EMPTY;
  function IS_FULL(S:in STACK_TYPE) return BOOLEAN is
  begin return S.TOP = S.STACK_SIZE;
  end IS_FULL;

  procedure PUSH(S:in out STACK_TYPE; ITEM: in INTEGER) is
  begin
    if IS_FULL(S) then
       PUT_LINE("ERROR-OVERFLOW");
     else
       S.TOP := S.TOP + 1;
```

```
      S.STK(S.TOP) := ITEM;
     end if;
  end PUSH;

  procedure POP(S:in out STACK_TYPE;ITEM:out INTEGER) is
  begin
    if IS_EMPTY(S) then
       PUT_LINE("ERROR-UNDERFLOW");
    else
       ITEM := S.STK(S.TOP);
       S.TOP := S.TOP - 1;
    end if;
  end POP;

  function "="(LEFT,RIGHT:in STACK_TYPE) return BOOLEAN is
  begin
    return LEFT.TOP = RIGHT.TOP and then
        (LEFT.STK(1..LEFT.TOP)=RIGHT.STK(1..RIGHT.TOP));
  end "=";

  procedure COPY(FROM:in STACK_TYPE; TO:out STACK_TYPE) is
  begin            --kontrola zda je dostatek mista pro copy
    if FROM.TOP > TO.STACK_SIZE then
       PUT_LINE("ERROR-OVERFLOW");
    else               --kopirovani
       TO.TOP := FROM.TOP;
       TO.STK(1..FROM.TOP) := FROM.STK(1..FROM.TOP);
    end if;
  end COPY;

  procedure CLEAR(S: in out STACK_TYPE) is
  begin
    S.TOP := 0;              --stačí nastavit TOP na nulu
  end CLEAR;
end STACK2;                  --konec těla modulu
```

S takto vytvořeným zásobníkem pak může uživatel pracovat ve svém programu např. způsobem:

```
with STACK2, ...; use STACK2, ...;
procedure STACK2H is
  S1,S2:   STACK_TYPE(20);  --zde deklaruje své vlastní
                                                      zásobníky
  I,J,K:INTEGER;
begin
  PUSH(S1,1);PUSH(S1,2);PUSH(S1,3);
  COPY(S1,S2);
  POP(S2,I);POP(S2,J);POP(S2,K);
  PUT(I);PUT(J);PUT(K);NEW_LINE;
end STACK2H;
```

Kapitola 5

Zpracování výjimečných situací

K zajištění co největší bezpečnosti programů **ADA** poskytuje mechanismus zpracování nenormálních a chybových stavů výpočtu – tzv. výjimečných situací (angl. *exception*). Uživatel má možnost popsat jejich zpracování v závěru příkazových částí programových konstrukcí. Výjimečné situace (krátce *výjimky*) **ADA** rozděluje na standardní (předdefinované) a definované uživatelem.

Výjimky jsou mimořádné situace při výpočtu programu, vedoucí k run-time chybám.

Standardní výjimečné situace

V modulu `STANDARD` jsou předdefinovány výjimečné situace:

1. `CONSTRAINT_ERROR`
2. `NUMERIC_ERROR`
3. `STORAGE_ERROR`
4. `TASKING_ERROR`
5. `PROGRAM_ERROR`

`CONSTRAINT_ERROR` je důsledkem překročení rozsahu přípustných hodnot. Vzniká např.:

- při hodnotě indexu mimo meze pole,
- je-li přiřazována hodnota překračující povolený rozsah pro levostrannou proměnnou,

- je-li předávaný parametr podprogramu mimo rozsah formálního parametru.

`NUMERIC_ERROR` je způsobována takovými aritmetickými operacemi, jejichž výsledek je ilegální, nebo operace je neproveditelná. Např. přetečení, podtečení, dělení nulou apod.

`STORAGE_ERROR` je důsledkem vyčerpání datové oblasti (zásobníku, či hromady). Nastává při vstupu do bloku, podprogramu, při zpracování deklarace, příp. alokace dynamické proměnné. Poněvadž samotné zpracování výjimky nemá paměťové nároky, je možné je provést.

`TASKING_ERROR` vzniká při pokusu o komunikaci s paralelně probíhajícím procesem, který není schopen komunikovat (viz později).

`PROGRAM_ERROR` zahrnuje ostatní druhy chybových situací. Častou příčinou je nedefinovaná hodnota funkce (výstup přes závěrečné end).

Ošetření výjimečných situací naznačuje příklad:

```
...
    begin
        ...  --posloupnost příkazů
    exception
        when NUMERIC_ERROR => příkazy záchranné akce1
        when CONSTRAINT_ERROR => příkazy záchranné akce2
    end;
...
```

Dojde-li v *posloupnosti příkazů* k některé z citovaných chyb, je výpočet přenesen na *příkazy záchranné akce*. Po jejich provedení je rozsahová jednotka (např. blok nebo podprogram), ve které došlo k chybě opuštěna, výpočet se do místa vzniku chyby nevrací. Forma popisu výjimek je obdobná příkazu **case**. Připouští i použití others alternativy, pro zahrnutí všech ostatních (explicitně necitovaných výjimečných situací. Oproti **case** příkazu nemusí však **exception** část obsahovat vyčerpávající výčet výjimek.

Standardně předdefinované výjimečné situace mohou nastat i při zpracování deklarací. V takovém případě je zpracování deklarací opuštěno a výjimka předána do dynamicky nadřazené konstrukce.

Uživatelem zavedené výjimečné situace

Uživateli je umožněno definovat své vlastní výjimky. Způsobí je pak příkazem

raise *jméno_výjimečné_situace*;

Tímto příkazem je ostatně možné vyvolat i předdefinované výjimky. Deklarace výjimky je podobná deklaraci proměnné, nemá však její vlastnosti (není dovoleno ji přiřazovat ani předávat jako parametr).

Časté je exportování výjimečných situací modulem. Jsou vlastně také jednou ze služeb, které může modul poskytovat, neboť představují nenormální situaci při obsluze uživatele modulu. Jak ale postupovat při nestandardní situaci, to musí určit uživatel modulu. Demonstrujme použití výjimek na příkladu modulu zásobníku. Pro úspornost zápisu použijme jako výchozí text modul STACK1.

Příklad 3.7 Modul exportující výjimky

```
package STACK3 is              --specifikační část modulu
  subtype STACK_ITEM is INTEGER range -10_000..10_000;
  procedure PUSH(ITEM:STACK_ITEM);
  function POP return STACK_ITEM;
    FULL, EMPTY: exception;     -- deklarace výjimek
end STACK3;                --konec specifikační části

package body STACK3 is     --tělo modulu
  STACK_SIZE: constant INTEGER := 1_000;
  STACK_ARRAY: array(1..STACK_SIZE) of STACK_ITEM;
  TOP: INTEGER range 0..STACK_SIZE;

  procedure PUSH(ITEM: STACK_ITEM) is
  begin
 if TOP = STACK_SIZE then raise FULL; end if; -- způsobí
                                                 výjimku
    TOP := TOP + 1;
    STACK_ARRAY(TOP) := ITEM;
  end PUSH;
```

```
  function POP return STACK_ITEM is
  begin
 if TOP = 0 then raise EMPTY; end if;   -- způsobí výjimku
    TOP := TOP - 1;
    return STACK_ARRAY(TOP + 1);
  end POP;

begin                               --inicializační část
  TOP := 0;
end STACK3;                         --konec těla modulu
```

Uživatel modulu pak může programovat akce prováděné při překročení rozsahu zásobníku.

```
with STACK3,...; use STACK3,...;
   ...
  begin ...
     PUSH(NĚCO);  ...
     NĚCO := POP; ...
     ...
    exception
      when FULL   => PUT("zásobník je plný, hodnota"
                                        & "neuložena");
      when EMPTY  => PUT("zásobník je prázdný, pracuj"
                                 & "s náhradní hodnotou");
                     NĚCO := NÁHRADNÍ_HODNOTA;
      when others => PUT("jina chyba");
   end;
   ...
```

Propagace výjimečných situací

Zpracování výjimky je dynamickým procesem. Pokud její zpracování (ovladač výjimky) není popsáno v rozsahové jednotce kde vznikla, je tato jednotka opuštěna a výpočet přenesen do dynamicky nadřazené jednotky.

Buď do místa vyvolání v případě vzniku výjimky v podprogramu, či za koncové end v případě jejího vzniku v bloku. Ovladač výjimky je pak hledán v této nadřazené jednotce. Dojde-li se při této propagaci (šíření) výjimky až do hlavního programu a ani tam není ovladač uveden, výpočet je ukončen.

Vzhledem k dynamickému šíření výjimek mohou nastávat situace, kdy výjimku je třeba částečně ošetřit v místě, ve kterém její jméno není viditelné. K tomuto účelu lze využít alternativy **others**, spolu s příkazem **raise** bez udaného jména výjimky. Tuto podobu může mít příkaz **raise** pouze v exception části. Důsledkem pak je propagování té výjimky, která provádění exception části způsobila. Například:

```
exception
  when MOJE_CHYBA | TVOJE_CHYBA => PUT("naše chyba");
  when others  => PUT("jiná chyba");    --částečné ošetření
                  raise; --propagování výše k dokončení ošetření
end;
```

Provádění kontrol při výpočtu může být neúnosně časově náročné. Proto se připouští i možnost jejich potlačení pomocí direktiv překladači (nazývaných *pragma*). Např.

```
pragma SUPPRESS(INDEX_CHECK);
```

potlačí kontroly rozsahu indexů pole. Potlačení se může týkat také jen určité proměnné či typu, např.

```
pragma SUPPRESS(RANGE_CHECK, INTEGER);
```

Použití by se mělo pečlivě zvažovat, neboť snižuje bezpečnost programu. Je pouze doporučením pro implementaci, které nemusí být překladačem plně realizováno.

Kapitola 6

Generické programové jednotky

Generické programové jednotky jsou předlohami, které reprezentují abstraktní datové struktury

Umožňují programátorovi aplikovat logicky stejné funkce na různé datové objekty. Jejich konkrétní, proveditelné exempláře jsou vytvářeny příkazem generického ustavení - instalace (viz generování modulu `INT_IO` v příkladu 3.4.). V programovém textu jsou generické jednotky odlišeny jim předřazeným symbolem **generic**. Za ním následují generické formální parametry, které při generickém ustavení jsou nahrazovány skutečnými parametry a pak teprve je uveden obvyklý tvar programové jednotky. Rozdělují se na:

- generické funkce,
- generické procedury,
- generické moduly.

Generické jednotky nejsou přímo proveditelné, jsou vzorem pro vytvoření proveditelné programové jednotky

Generické procedury

Generické procedury mají obecný tvar:

generic
generické formální parametry generická specifikace
procedure JMENO(*formální parametry*);

procedure JMENO(*formální parametry*) **is**
deklarace
begin tělo
příkazy
end JMENO;

Práci s generickou procedurou ilustruje program `PLANETY`, který používá generický podprogram `VYJMENOVANY_TYP` pro tisk hodnot různých vyjmenovaných typů. Samotná procedura `VYJMENOVANY_TYP` je vytvořena s pomocí generického modulu pro I/O operace s výčtovými typy.

Příklad 3.8

```
with TEXT_IO; use TEXT_IO;
procedure PLANETY  is
   type SLUNECNI_SOUSTAVA is (MERKUR,VENUSE,ZEME,MARS,
                          JUPITER,SATURN,URAN,NEPTUN,PLUTO);
   type TYP_HUDBY is (JAZZ,FOLK,SONATA,METAL);

   generic                          -- generická procedura
      type VYJMENOVANY_TYP is (<>); -- gen.parametr je diskr.
                                    -- typu
   procedure TISK_VYJMENOVANY_TYP;
   procedure TISK_VYJMENOVANY_TYP is
       package VYJMENOVANY_TYP_IO is new
                           ENUMERATION_IO(VYJMENOVANY_TYP);
       use VYJMENOVANY_TYP_IO;
   begin for I in VYJMENOVANY_TYP loop
         PUT(I); NEW_LINE;
       end loop;
   end TISK_VYJMENOVANY_TYP;
   procedure TISK_PLANETY is new
       TISK_VYJMENOVANY_TYP(VYJMENOVANY_TYP =>
                                        SLUNECNI_SOUSTAVA);
```

```
   procedure TISK_HUDBY is new
       TISK_VYJMENOVANY_TYP(VYJMENOVANY_TYP => TYP_HUDBY);

begin -- hlavni program
   TISK_PLANETY;
   TISK_HUDBY;
end PLANETY;
```

Generické funkce

Generické funkce mají obecný tvar:

generic — generická specifikace
    generické formální parametry
function *JMENO (formální parametry)* **return** *TYP*;

function *JMENO (formální parametry)* **return** *TYP* **is** — tělo
    deklarace
begin
    příkazy
end *JMENO;*

Způsob práce s generickými funkcemi je obdobný práci s generickými procedurami.

Generické formální parametry

Generický formální parametr může udávat:

- typ,
- hodnotu (lze označit **in**) nebo proměnnou (musí být označen **in out**),
- podprogram.

Záměna generických formálních parametrů je prováděna v době překladu, při ustavení konkrétního exempláře programové jednotky.

Nejčastěji jsou používány generické formální typy. Jejich jednotlivé možnosti uvádí následující souhrn:

`type OBECNÝ is limited private;` skutečným parametrem může být libovolný typ, veškeré operace s ním jsou zadány generickými formálními podprogramy

`type PRIVÁTNÍ is private;` skutečný parametr je typem, který oproti předchozímu dovoluje navíc operace `:=`, `=`, `/=`

`type PŘÍSTUPOVÝ is access KAM;` skut. par. může být ukazatel zpřístupňující typ `KAM`, který bývá také generický

`type DISKRÉTNÍ is (<>);` lze jej nahradit diskrétním typem

`type INTEGER_TYP is range <>;` lze jej nahradit libovolným celočíselným typem

`type FIXED_TYP is delta <>;` lze jej nahradit libovolným typem v pevné řádové čárce

`type FLOAT_TYP is digits <>;` může být nahrazen libovolným typem v pohyblivé řádové čárce

`type DETERMINOVANÉ_POLE is array (INDEXY) of ELEMENTY;`
je přípustný pro libovolné determinované pole stejných dimenzí, typů indexů a typu komponent

`type NEDETERMINOVANÉ_POLE is array (INDEXY range <>) of ELEMENTY;`
je přípustný pro libovolné nedeterminované pole stejného počtu rozměrů, typů indexů a typů komponent

Parametrizace generickými hodnotami a proměnnými je formálně zapisována stejně jako deklarace proměnných. Pokud není uveden druh, předpokládá se implicitně **in**. Např.

```
generic
        MAX: INTEGER;
        DELKA: INTEGER := 60;          gener. hodnotové
        SIRKA: in INTEGER := 80;       parametry
procedure STRANKY ...
```

Parametrizaci generickými proměnnými nelze doporučit, neboť zpřístupňuje globální proměnné.

Generickými formálními parametry mohou být také podprogramy. Jejich použití je motivováno zejména potřebou zavést operace na generických typech. Proto se nejčastěji setkáváme s formálními generickými funkcemi. Způsob jejich použití uvedeme příkladem.

Příklad 3.9 Generická funkce vyhodnocující sumu z prvků vektoru. Předpokládejme obecnost rozsahu pole i typu prvků. V takovém případě musí uživatel funkce vyjádřit smysl operace sčítání pro příslušný typ prvků (čísla, matice, barvy ...).

```
generic
     type H is private;
     type P is array(INTEGER range <>) of H;
     NULA: H;
with function "+"(U,V: H) return H; --form. gener. funkce
function SUMA(X: P) return H is
     S: H := NULA;
begin
     for I in P'RANGE loop
           S := S + X(I);
     end loop;
     return S;
end SUMA;
```

Programový segment, který funkci používá, pak může např. mít tvar:

```
... declare
```

```
function MOJE_SUMA is new SUMA(FLOAT, MUJ_VEKTOR, 0.0,"+");
```

```
      V1: MUJ_VEKTOR;
      CELKEM: FLOAT;
      ...
    begin
          ...  CELKEM := MOJE_SUMA(V1);  ...
    end;  ...
```

Generické moduly

Generické prostředky jsou užitečné především pro vytváření programových knihoven. Také většina I/O operací jazyka **ADA** je poskytována z předdefinovaných generických modulů.

Tvar generického modulu se od normálního modulu liší pouze předřazením slova **generic** a generickými formálními parametry před jeho specifikační část.

Příklad, který dále uvádíme implementuje zásobník pro hodnoty libovolného typu, který připouští přiřazení a test na rovnost a nerovnost. Pro úspornost použijeme za výchozí text modulu STACK1 z př. 3.5.

Příklad 3.10 Generický zásobník

```
generic
  STACK_SIZE: POSITIVE; --generický parametr - velikost
                                               -- zásobníku
  type ITEM is private; --generický parametr - typ prvků
package STACK4 is
  procedure PUSH(A: ITEM);
  function POP return ITEM;
end STACK4;             --konec specifikační části
```

```
package body STACK4 is
  --deklaracni cast
  STACK_ARRAY: array(1..STACK_SIZE) of ITEM;
  TOP: INTEGER range 0..STACK_SIZE;

  procedure PUSH(A: ITEM) is
  begin
    TOP := TOP + 1;
    STACK_ARRAY(TOP) := A;
  end PUSH;
  function POP return ITEM is
  begin
    TOP := TOP - 1;
    return STACK_ARRAY(TOP + 1);
  end POP;

begin          --inicializační část
  TOP := 0;
end STACK4;
-------------------------------------------------------

--hlavní program, využívá kompilační jednotku STACK4
with STACK4,TEXT_IO; use TEXT_IO;
procedure STACK4H is
  package INTSTACK_IO is new INTEGER_IO(INTEGER);
  type TCIGARET is (PETRA, SPARTA, START, CAMEL);
  package ENUMSTACK_IO is new ENUMERATION_IO(TCIGARET);

    package STACK_100 is new STACK4(100, INTEGER);
    package STACK_100 is new STACK4(60, TRIGARET);

  use INTSTACK_IO, STACK_100, STACK_60;
  X,Y: INTEGER;
begin
  PUSH(5);  PUSH(SPARTA);
  X := 4;
  PUSH(PETRA);  PUSH(X);
  INTSTACK_IO.PUT(POP);  --výstup 4, kvalifikace to rozliší
```

```
  Y := POP;              --výstup 5
  PUT(Y);
  ENUMSTACK_IO.PUT(POP);  --vyžaduje kvalifikaci
  ENUMSTACK_IO.PUT(POP);  --vyžaduje kvalifikaci
  NEW_LINE;
end STACK4H;
```

Kapitola 7

Prostředky pro paralelní výpočty

Paralelně proveditelná aktivita (proces) je nazývána *task* (překl. úkol). Forma jeho zavedení je obdobná modulu. Jeho deklarace může být součástí deklarační části modulu, podprogramu, bloku nebo jiného úkolu.

Konstrukce **task** představuje program paralelně proveditelného procesu, schopného komunikace s ostatními procesy.

Podle tohoto programu je automaticky založen proces se stejným jménem, tj. nerozlišuje se program a podle něj probíhající proces (pokud není použita konstrukce typ úkolu, kterou probereme později). Příkaz aktivace není zaveden explicitně. Všechny procesy–úkoly jsou potomky hlavního programu, který lze rovněž chápat jako úkol, i když syntaktickou podobu úkolu nemá. Úkoly jsou aktivovány v okamžiku, kdy výpočet úkolu, který zpracovává jejich deklarace dosáhne **begin** za deklarační částí.

Úkol je složen ze specifikační části a z těla.

task *JMÉNO* **is**
        deklarace jmen komunikačních vstupů
end *JMÉNO*;
task body *JMÉNO* **is**
        lokální deklarace a příkazy
end *JMÉNO*;

– Úkoly mohou sdílet jeden procesor (multiprogramový režim), přepínání procesoru mezi úkoly zajišťuje podpůrný systém jazyka bez nutnosti jazykového vyjádření.

– Úkoly mohou sdílet více procesorů (multiprocesorový režim), jsou pak vykonávány fyzicky paralelně.

– Úkoly mohou být prováděny na více počítačích (distribuovaný systém). Jazykem **ADA** se to rovněž nevyjadřuje.

Oproti jiným programovým jednotkám platí pro úkoly jistá omezení:

a) Úkol nemůže být generickou jednotkou.

b) Úkol nemůže být knihovní jednotkou, tzn. jeho deklarace musí být součástí podprogramu nebo modulu. Jeho tělo však může být v rodičovské jednotce nahraženo zbytkem

   `task body` *JMÉNO* `is separate;`

   a překládáno jako podjednotka.

c) Textově lze umístit specifikaci úkolu stejným způsobem jako specifikaci modulu, s výjimkou uvedenou v b).

K normálnímu ukončení práce úkolu dojde průchodem end na konci jeho těla. Nadřazený úkol může skončit činnost až v okamžiku, kdy skončily všechny jím vytvořené procesy. K násilnému ukončení z jiného úkolu je možné použít příkazu

`abort` *jméno ukončovaného úkolu;*

Jeho uplatnění by mělo být výjimečné. Úkol je v jazyku **ADA** synonymem procesu. Ukončení procesu obecně implikuje ukončení všech jím vytvořených procesů. Pozastavené procesy je nutné před ukončením vyřadit z front, ve kterých čekají. V případě ukončování komunikujícího procesu je nutné uvědomit jeho partnery. Z těchto důvodů se použitím příkazu **abort** neplýtvá.

Rendezvous v ADě

Pro interakci procesů používá **ADA** principu asymetrického rendezvous (viz čl. 2.3), kterým eliminuje potřebu semaforů (umí je nahradit), umožňuje synchronní a nepřímo (zavedením pomocného procesu) i asynchronní komunikaci procesů zasíláním zpráv. Dovoluje také elegantní konstrukci monitorů.

Místům, ve kterých úkol akceptuje rendezvous se říká **entry** (tj. vstup). Popis vstupu je syntakticky stejný jako deklarace procedury. Příkazy, kterými jsou volány služby jiného úkolu, jsou nazývány volání vstupu (angl.

call entry). Jedná se o příkaz k vlastnímu provedení rendezvous. Na úkol lze zevně pohlížet jako na množinu jeho vstupů. Specifikace úkolu pak zřejmě obsahuje specifikaci všech jeho vstupů.

Příkaz volání vstupu je obdobou volání procedury. Má tvar:

JMÉNO_ÚKOLU.JMÉNO_VSTUPU (parametry);

Pro každý vstup musí volaný úkol obsahovat alespoň jedno místo, ve kterém tento vstup akceptuje. Místo, kde je vstup akceptován má tvar příkazu:

```
accept JMÉNO_VSTUPU (formální_parametry) do
          seznam příkazů
end JMÉNO_VSTUPU;
```

Seznam příkazů (tělo **accept** příkazu) je prováděn v režimu vzájemného vyloučení, je kritickou sekcí.

Příkaz **accept** musí být uveden v základní úrovni těla úkolu, nelze jej vnořovat. Je však možné použít stejného jména příkazu **accept** v jednom úkolu několikrát.

Následující příklad představuje úkol, který realizuje paměťovou schránku pro jednu celočíselnou proměnnou. Úkoly používající tuto schránku mohou do ní zapsat je-li prázdná a číst z ní je-li plná. Mechanismem rendezvous se zabraňuje kolizi ve čtení a zápisu.

Příklad 3.11 Paměťová schránka

```
task SCHRANKA is
      entry PUT(X: in INTEGER);            specifikační
      entry GET(X: out INTEGER);           část
end;

task body SCHRANKA is
      V:  INTEGER
begin
      loop
            accept PUT(X: in INTEGER) do
                  V := X;
```

```
            end PUT;
            accept GET(X: out INTEGER) do
                  X := V;
            end GET;
      end loop;
end SCHRANKA;
```

Uspořádáním příkazů `accept` v nekonečné smyčce je zajištěno, že číst bude možné až po zapsání údaje do schránky. Po aktivaci úkol čeká nejprve na volání vstupu `PUT` z jiného úkolu, které může mít tvar:

```
SCHRANKA.PUT(22);
```

Pak teprve může akceptovat vstup `GET`, např. tvaru:

```
SCHRANKA.GET(J);
```

V případě, že jeden vstup je volán z více úkolů, jsou volání frontována. Je zřejmé, že úkol může být jen v jedné frontě. Dokud není obsloužen, je pozastaven. Pomocí atributu `COUNT` je možné zjišťovat počet úkolů, které čekají na daný vstup. Např.

```
SCHRANKA.PUT'COUNT
```

Atributem `TERMINATED` je zjistitelné, zda úkol včetně svých potomků byl ukončen. Pro úkoly není použitelná klauzule **use**.

Vznikne-li uvnitř úkolu výjimečná situace, zpracuje se normálním způsobem. Singulární je však případ, kdy vznikne v průběhu komunikace úkolů, tj. při zpracování příkazu **accept**. Pokud tento příkaz neobsahuje lokální zpracování výjimky, je příkaz **accept** opuštěn a výjimka se propaguje jak za příkaz **accept**, tak i do volajícího úkolu. Při volání vstupu úkolu, který již skončil svoji činnost nebo byl ukončen příkazem **abort**, je ve volajícím úkolu způsobena výjimečná situace `TASKING_ERROR`.

Příkaz select – zavedení nedeterminismu

Dosud bylo uvažováno sekvenční zpracování programu úkolu. Pokud výpočet došel k příkazu **accept**, úkol byl nucen vyčkat volání vstupu právě tohoto příkazu **accept**. Úkol však často poskytuje množinu služeb, které mohou být volány v libovolném pořadí. Takový způsob zpracování umožňuje příkaz **select**.

Forma příkazu **select** je:

```
select
  accept  něco( parametry)  do
    příkazy;      --vlastní rendezvous
  end;
  příkazy;        --již mimo rendezvous. Např. úpravy
                  --předaných dat. Volající úkol pokračuje
                  --dál, ale volaný ještě neakceptuje další
                  --volání vstupu
or
  accept  něco_jiného( parametry) do
   ...
  end;
or
   ...
end select;
```

V příkazu **select** se akceptuje volání kteréhokoliv ze vstupů. Při souběhu volání několika vstupů z úkolů se stejnou prioritou *není determinováno v jakém pořadí se budou provádět.*

Alternativ **accept** v příkazu select může být libovolný počet a může jim předcházet zpřístupňující klauzule (tzv. *hlídka* – ang. *guard*) ve tvaru:

`when` *podmínka* =>

Ta zabrání provedení přislušného **accept** příkazu není-li podmínka splněna.

select
  when *podmínka* =>
    accept
      příkazy kritické sekce
    end;
    příkazy mimo kritickou sekci
or
  ...
end select;

Další možnosti alternativ v příkazu **select**:

a) Časově omezený příkaz **select**

```
select
  accept ...
or
  delay nějaká_doba;
end select;
```

Parametr *nějaká_doba* je speciálního datového typu **duration** (trvání), předdeklarovaného v modulu `STANDARD`. Způsobí pozastavení úkolu na udanou dobu, nemůže-li se uplatnit žádný z příkazů **accept**. Je použitelný i mimo konstrukci **select**.

b) Alternativa terminate

```
select
  accept ...
or
  terminate;
end select;
```

Úkoly, které čekají v příkazu **select** s alternativou **terminate** na volání některého ze svých vstupů, jsou potenciálně ukončitelné. Skončí svoji činnost při ukončení činnosti své nadřízené jednotky (programové jednotky v níž jsou deklarovány). Tou může být blok, podprogram, úkol či knihovní modul. Nadřízená jednotka (ang. *master*) může skončit činnost (např. průchodem **end**), pokud všechny v ní aktivované procesy již skončily činnost a nebo připouštějí své ukončení alternativou **terminate**. Alternativa **terminate** je potřebná zejména tehdy, je-li úkol realizován nekonečným cyklem. Současné použití příkazu `delay` se ovšem vylučuje.

c) Alternativa else

```
select
  accept ...
or
  ...
else
  příkazy prováděné v případě neuplatnění žádného accept
end select;
```

Kombinace alternativ **delay, terminate** a `else` není smysluplná a proto se ani nedovoluje.

Příklad 3.12

Abstraktní datový typ `PAMĚŤOVÁ_SCHRÁNKA` dovolující libovolný počet čtení ze sdílené proměnné současně.

```
generic
     type PRVEK is private;
package SDILENA_PROMENNA is
     procedure PUT(X: in PRVEK);
     procedure GET(X: out PRVEK);
end SDILENA_PROMENNA;

package body SDILENA_PROMENNA is
     V: PRVEK;

     task RIZENI is
           entry START; --je možný i vstup bez parametrů
           entry STOP;
           entry PIS;
     end RIZENI;

     task body RIZENI is
           POCET: INTEGER := 0;
     begin
        accept PIS(X: in PRVEK) do
           V:=X;   --zajistí naplnění proměnné první
                   --hodnotou
        end PIS;
        loop
           select
             when PIS'COUNT = 0 =>
              accept START; --"do příkazy" je nepovinné
              POCET:=POCET+1;--je mimo accept, probíhá
                             --paralelně
           or
```

```
            accept STOP;
            POCET:=POCET-1;
         or
           when POCET = 0 =>
            accept PIS(X: in PRVEK) do
                  V := X;
            end PIS;
         end select;
      end loop;
   end RIZENI;

   procedure GET(X: out PRVEK) is
   begin RIZENI.START;  X := V; RIZENI.STOP;
   end GET;

   procedure PUT(X: in PRVEK) is
   begin RIZENI.PIS(X);
   end PUT;

end SDILENA_PROMENNA;
```

Úkol `RIZENI` je aktivován po zpracování deklarací v těle modulu, vygenerovaného ze šablony `SDILENA_PROMENNA`. Vstupy `START` a `STOP` mají funkci signálů. Počet žádostí o čtení je obsažen v proměnné `POCET`. Čtení ze sdílené proměnné je dovoleno zahájit pouze tehdy, nechce–li žádný proces do ní zapisovat. Zapisovat je možné v tom případě, kdy nikdo právě její obsah nečte. Uvedené podmínky jsou zajišťovány hlídkami.

Příklad 3.13. ADT zásobník dovolující paralelní použití

Předpokládejme, že provádět zápis a výběr mohou různé procesy. Z toho plyne, že zásobník musí být koncipován jako monitor.

```
   generic
   SIZE: POSITIVE;
   type  POLOZKA is private;
   package STACK5 is
```

```
    task ZASOBNIK is
      entry VLOZ( X :  in POLOZKA);
      entry UBER( X : out POLOZKA);
    end ZASOBNIK;
  end STACK5;

  package body STACK5  is          -- tělo
    task body ZASOBNIK is
    S  : array  (1..SIZE)  of POLOZKA;
    SP : INTEGER range 0..SIZE;
    begin
      SP := 0;
      loop
        select
          when SP < SIZE =>
            accept VLOZ( X : in POLOZKA) do
              SP := SP + 1;
              S(SP) := X;
            end VLOZ;
        or
          when SP > 0 =>
            accept UBER( X : out POLOZKA) do
              X  := S(SP);
              SP := SP - 1;
            end UBER;
        or
          terminate;
        end select;
      end loop;
    end ZASOBNIK;
  end STACK5;
```

Příklad použití v hlavním programu:

```
with STACK5, TEXT_IO;  use TEXT_IO;
procedure MAIN is
   package STACKI is new STACK5(20, INTEGER);
   package STACKC is new STACK5(100, CHARACTER);
```

```
   C: CHARACTER := ‘A‘;
   I: INTEGER := 10;
begin ...
   STACKI.ZASOBNIK.VLOZ(I);
   STACKC.ZASOBNIK.VLOZ(C);
   STACKI.ZASOBNIK.VLOZ(5);
   STACKC.ZASOBNIK.VLOZ(‘B‘);
   STACKI.ZASOBNIK.UBER(I);
   STACKC.ZASOBNIK.UBER(C);
   PUT(C); NEW_LINE;...
end MAIN;
```

Vše, co bylo doposud řečeno o příkazu **select**, se týká jeho použití v obsluhujících (volaných) úkolech. Možnosti podmíněného a časově omezeného volání vstupu pomocí příkazu **select** má ale i aktivní (tj. volající) úkol.

d) Podmíněné volání vstupu
(řeší situaci není-li obsluha připravena)
select
jméno_úkolu.jméno_obsluhy (aktuální parametry);
případné další příkazy po provedení rendezvous
else
příkazy alternativního výpočtu není-li rendezvous ihned proveditelné
end select;

e) Časované volání vstupu (volání s timeoutem)
Volající úkol tím vyjadřuje, že akceptovat jeho volání vstupu lze v zadaném čase.
select
jméno_úkolu.jméno_obsluhy (aktuální parametry);
případné další příkazy po provedení rendezvous
or
delay *nějaká_doba;*
případné další příkazy při neuskutečnění rendezvous v zadané době
end select;

Typ úkol

Konstrukce typového úkolu představuje program, podle kterého může paralelně probíhat libovolný počet úkolů-procesů (tzn. je šablonou pro vytváření konkrétních úkolů). Úkoly-procesy musí být explicitně vytvořeny (a pojmenovány), a to buď staticky (deklarací úkolu příslušného typu), nebo dynamicky (pomocí new, kdy se k pojmenování používají ukazatelové proměnné). Platí, že typ úkol a proces-úkol dle něj probíhající, jsou ve stejném vztahu jako datový typ a proměnná tohoto typu.

Příklad 3.14. Typ zásobníku pro paralelní přístup

```
with EXPORTER_TYPU_POLOZKA;
use  EXPORTER_TYPU_POLOZKA;

 package STACK6 is
  task type ZASOBNIK is
     entry VLOZ( X: in POLOZKA);
     entry UBER( X: out POLOZKA);
  end ZASOBNIK;
  type REF_ZASOBNIK is access ZASOBNIK;
end STACK6;

package body STACK6 is
   task body ZASOBNIK is
     SIZE: constant INTEGER := 200;
     -- ostatní, počínaje deklarací S: array ...
     -- je stejné jako v předchozím příkladě 3.13.
end STACK6;
```

Použití v programu:

```
with STACK6, EXPORTER_TYPU_POLOZKA;
use  STACK6, EXPORTER_TYPU_POLOZKA;

procedure MAIN is
   ...
  Z1, Z2 : ZASOBNIK;
  Z3 : REF_ZASOBNIK := new ZASOBNIK;
```

```
    ...
  begin
    ....
    Z1.VLOZ(NĚCO);
    ...
  end MAIN;
```

Poznámka:
Vytvořené úkoly `Z1, Z2, Z3` jsou paralelně spuštěny ihned za příkazem **begin** v hlavním programu.
Nevýhodou současného tvaru příkladu 3.14 oproti generické konstrukci příkladu 3.13 je:

- potřeba importovat typ `POLOZKA` (nelze jej bohužel importovat z hlavního programu, do kterého je `STACK6` importován – viz kompilační závislosti),
- statický rozsah zásobníku.

Druhou z nevýhod je možno eliminovat prostřednictvím předání parametrů úkolu. Předání parametrů lze jednoduše zařídit doplněním speciálních vstupů, které slouží jen pro přenos parametrů, a přes které musí úkol přejít dříve než zahájí vlastní činnost. Upravme v tomto smyslu příklad 3.14.

```
with EXPORTER_TYPU_POLOZKA;use EXPORTER_TYPU_POLOZKA;
package STACK7 is
  task type ZASOBNIK is
    entry VLOZ( X :  in POLOZKA);
      entry PARAMETR(SI: in POSITIVE);
    entry UBER( X : out POLOZKA);
  end ZASOBNIK;
end STACK7;

package body STACK7  is          -- tělo
 task body ZASOBNIK is
  SIZE: POSITIVE;
 begin
```

```
accept PARAMETR(SI: in POSITIVE) do SIZE := SI;
```

```
      end PARAMETR;
      declare
       S  : array  (1..SIZE)  of POLOZKA;
       SP : INTEGER range 0..SIZE;
      begin
      ...
     end ZASOBNIK;
   end STACK7;
```

Hlavní program pak využívá typ `ZASOBNIK` způsobem např.

```
with STACK7, TEXT_IO;  use TEXT_IO, STACK7;
procedure MAIN is
   C: CHARACTER := ‘A‘;
   I: INTEGER := 10;
   STACKC: ZASOBNIK;
begin
     ...
   STACKC.PARAMETR(100);
   STACKC.VLOZ(C);
   STACKC.VLOZ(‘B‘);
   STACKC.UBER(C);
   PUT(C); NEW_LINE;
     ...
end MAIN;
```

Příklad 3.15.

Na závěr výkladu o paralelním programování v jazyce **ADA** uvedeme ještě jeden rozsáhlejší příklad. Jedná se o generický modul – typový monitor, který umožní asynchronní komunikaci úkolů zasíláním zpráv přes kruhovou vyrovnávací paměť. Typ zasílaných zpráv může být libovolný. Vyrovnávací paměť je implementována jako pole.

Program je rozdělen do tří kompilačních jednotek:

- Soubor `BUF_S.ADA` obsahuje specifikaci univerzálního knihovního modulu (generic package `G_BUFFER`) pro monitor – kruhový buffer,

- soubor BUF_B.ADA obsahuje tělo modulu,
- soubor TESTBUF.ADA obsahuje proceduru TESTBUF – hlavní program pro demonstrační ověření správné funkce knihovního modulu.

```
     -- Soubor BUF_A.ADA (první kompilační jednotka) --

generic
  -- PARAMETRY

  type MESSAGE is private;          -- typ přenášené zprávy
  LBUFF: positive;                  -- délka bufferu v počtu
                                    -- zpráv
  with procedure NULL_MES (THE_MES: out MESSAGE);
                                    -- procedura pro nulování
                                    -- záznamu zprávy

package G_BUFFER is

  task type SERVER is               -- úkol pro obsluhu
                                    -- bufferu
    entry RESET;                    -- reset bufferu
    entry DESTROY;                  -- likvidace bufferu
    entry ADD (NEW_MES: in MESSAGE);-- zápis zprávy
    entry READ(THE_MES: out MESSAGE);-- poskytuje nejstarší
                                      -- zprávu
    entry STATE (FREE_NUM: out NATURAL); -- poskytuje stav
                                          -- bufferu
  end;

  type BUF_SERVER is access SERVER;
                      -- typ ukazatel na úkol typu SERVER

end G_BUFFER;
-----------------------------------------------------------
```

```
      -- Soubor BUF_B.ADA (druhá kompil. jednotka) --

package body G_BUFFER is

  task body SERVER is

    THE_BUFF: array(INTEGER range 1..LBUFF) of MESSAGE;
                                     -- vlastní buffer
    TAIL :INTEGER range 1..LBUFF+1;  -- index na první
                                     -- volné místo

    procedure INIT is          -- inicializační procedura
    begin
      for I in 1..LBUFF loop
        NULL_MES(THE_BUFF(I));
      end loop;
      TAIL := 1;
    end INIT;

    begin                            -- tělo úkolu
      INIT;
      loop
        select
          accept RESET;
            INIT;
        or
          accept DESTROY;
            exit;
        or
          when TAIL < LBUFF+1 =>
            accept ADD(NEW_MES: in MESSAGE) do
              THE_BUFF(TAIL) := NEW_MES;
              TAIL:=TAIL+1;
            end;
        or
          when TAIL > 1 =>
            accept READ(THE_MES: out MESSAGE) do
              THE_MES := THE_BUFF(1);
```

```
              TAIL := TAIL - 1;
              THE_BUFF(1..(TAIL-1)) := THE_BUFF (2..TAIL);
              NULL_MES(THE_BUFF(TAIL));
            end;
        or
          accept STATE (FREE_NUM: out NATURAL) do
            FREE_NUM := LBUFF - TAIL + 1;
          end;
        or
          terminate;
        end select;
      end loop;
    end SERVER;

end G_BUFFER;
------------------------------------------------------------

    -- Soubor TESTBUF.ADA (třetí kompilační jednotka) --

with TEXT_IO; use TEXT_IO;
with G_BUFFER;

procedure TESTBUF is                       -- hlavní program

  L_STRING : constant POSITIVE := 40;   -- délka zprávy
  L_BUFF : constant POSITIVE := 4;      -- délka bufferu

  subtype MESSAGE is STRING(1..L_STRING); -- typ zprávy
  KBD_MES: MESSAGE;                  -- zpráva z klávesnice

  procedure NULL_MES (THE_MES: out MESSAGE); -- specif.
                        -- procedury pro nulování zprávy
  package B is new G_BUFFER (MESSAGE, L_BUFF, NULL_MES);
    -- konkrétní verze generického modulu G_BUFFER
  use B;
  BUF: SERVER;   -- deklarace bufferu
                 -- (tj. vytvoření obslužného úkolu)
```

```
task type CONSUMER is  -- typový úkol pro čtení zprávy
                       -- z bufferu
                       -- nemá žádné entry
end;

task body CONSUMER is      -- tělo typového úkolu
  READ_MES : MESSAGE;      -- lok. proměnná pro přečtenou
                           -- zprávu
begin
  delay 0.5;          -- aby se stihlo vypsat ohlášení
                      -- hlavního programu
  PUT_LINE("Task CONSUMER pripraven");
  loop
    NULL_MES(READ_MES);   -- čistění lokální proměnné
    BUF.READ(READ_MES);   -- !!! rendezvous s úkolem BUF
    if READ_MES(1..6) = "zhebni" then
      PUT_LINE("Task CONSUMER se s Vami louci s pozdravem"
               & "DDT ZDAR !");
      exit;
    else
      PUT("Task CONSUMER :");
      PUT_LINE(READ_MES); NEW_LINE;
    end if;
  end loop;
end CONSUMER;

procedure NULL_MES (THE_MES: out MESSAGE)is
  -- tělo procedury - těla procedur a úkolů musí být až
  -- na konci deklarační části (po dekl. typů a prom.),
  -- proto byla na začátku specif. část pro využití
  -- v generickém modulu G_BUFFER
begin
  for I in 1..L_STRING loop
    THE_MES(I) := ' ';
  end loop;
end NULL_MES;
```

```
begin
  declare
    CONS1: CONSUMER; -- deklarace (vytvoření) úkolu CONS1
  begin
    PUT_LINE("Demonstracni program - komunikace"
                               & "zasilanim zprav");
    delay 1.0;        -- aby se stihlo vypsat ohlášení
                      -- úkolu CONS1
    loop
      declare
        NUM_CHAR : NATURAL;
      begin
        PUT_LINE("Zadej nejaky text !");
        NULL_MES(KBD_MES);
        GET_LINE(KBD_MES, NUM_CHAR);
        BUF.ADD(KBD_MES); -- !!! rendezvous hlav. programu
                          -- s úkolem BUF
        delay 1.0;        -- aby úkol CONS1 stačil vypsat
                          -- zprávu
        if KBD_MES(1..6) = "zhebni" then
          exit;
        end if;
      end;
    end loop;
    PUT_LINE("Hlavni program - konec komunikace");
  end;                -- zde dojde k likvidaci úkolu CONS1
end TESTBUF;          -- až zde dojde k likvidaci úkolu BUF
```

Kapitola 8

Kompilační závislosti

V předchozím výkladu byl intuitivně zaveden a používán pojem kompilační jednotka jako úsek programu mající podobu souboru, který je schopný samostatného překladu. Vlastnosti a formy kompilačních jednotek nyní probereme podrobněji.

Kompilační jednotkou může být:

- specifikační část modulu,
- tělo modulu,
- specifikační část generického modulu,
- tělo generického modulu,
- specifikační část podprogramu,
- tělo podprogramu,
- instalace generického podprogramu,
- instalace generického modulu.

Výhody použití kompilačních jednotek zahrnují:

- rekompilaci jen části programu v případě jeho modifikace,
- průhlednější modulární strukturu celého programu,
- umožnění týmové práce.

Kompilační jednotky se dělí — primární (dále značeno 1.)
sekundární (dále značeno 2.)

Překládaný soubor (kompilace) může být tvořen jedinou kompilační jednotkou nebo může zahrnovat několik kompilačních jednotek. Možnosti rozčlenění na kompilační jednotky:

a) V případě procedur:

   1. specifikace procedury
   2. tělo procedury

   nebo

   1. tělo procedury se specifikací (neliší se od těla)
   2. neexistuje

b) V případě funkcí:

   1. specifikace funkce
   2. tělo funkce

   nebo

   1. tělo funkce se specifikací (neliší se od těla)
   2. neexistuje

c) V případě modulů:

   1. specifikace modulu
   2. tělo modulu

   nebo neobsahuje–li modul příkazy

   1. specifikace modulu
   2. neexistuje

d) V případě generických jednotek vyžaduje řada implementací, aby primární jednotky (generické specifikace) byly překládány se sekundárními (těly) v jedné kompilaci t.j. v jednom souboru.

e) V případě podjednotek:

   1. neexistuje (je součástí rodiče a tak ji nelze samostatně překládat)
   2. separátně překládaná podjednotka.

Zavedením podjednotek je podporován způsob vytváření programů *shora dolů*, používání samostatného překladu ostatních kompilačních jednotek podporuje metodologii *zdola nahoru*.

Pro realizovatelnost překladu musí na kompilačních jednotkách existovat částečné uspořádání, vyjadřující jejich závislost. Závislosti, které platí mezi kompilačními jednotkami jsou tyto:

i. Podjednotky jsou závislé na svém rodiči. Rodičem může být jak primární tak i sekundární jednotka.

ii. Existuje-li primární i sekundární jednotka, pak sekundární jednotka (tělo) závisí na primární jednotce (specifikaci).

iii. Použitím klauzule **with** vzniká závislost klienta na primární jednotce citované v klauzuli **with**. Tuto klauzuli může používat jak primární, tak i sekundární jednotka. Použije-li ji primární jednotka, bude i odpovídající sekundární jednotka závislá na jednotkách citovaných ve **with** klauzuli.

Závislé kompilační jednotky lze překládat až po přeložení všech kompilačních jednotek, na kterých závisí.

Závislosti určují rozsah rekompilace při modifikaci kompilační jednotky. Překladač udržuje knihovnu v tomto smyslu konzistentní. Zda automaticky nebo upozorněním uživatele, to již závisí na implementaci.

Kapitola 9

Programování vstupních a výstupních operací

Podobně jako v jazyce **C** či **MODULA**, nejsou periferní operace ani v ADě součástí samotného jazyka. Jsou poskytovány prostřednictvím předdefinovaných modulů `TEXT_IO`, `SEQUENTIAL_IO`, `DIRECT_IO` a `IO_EXCEPTIONS`. Specifikační část těchto modulů je standardizována formou dodatků referenčního manuálu, popisujících tvary použitelných procedur, funkcí a výjimek pro periferní operace. Považujeme za účelné nepopisovat v tomto textu uvedené moduly vyčerpávajícím způsobem, zvídavější odkázat na manuál a soustředit se pouze na principy a nejčastěji potřebné operace.

`SEQUENTIAL_IO` je generický modul, dovolující vygenerovat pro zadaný datový typ periferní operace nad sekvenčními soubory. Exportovanými podprogramy jsou např. `CREATE, OPEN, CLOSE, DELETE, IS_OPEN, READ, WRITE, END_OF_FILE` atd.

`DIRECT_IO` je generický modul poskytující periferní operace pro práci se soubory s přímým přístupem. Pojmenování operací je shodné s předchozími. Je použitelný i pro sekvenční zpracování, navíc však dovoluje indexový přístup k položkám. Jeho procedury `READ` a `WRITE` proto mohou kromě parametrů udávajících soubor, proměnnou (příp. hodnotu), obsahovat ještě třetí parametr – index. Položky v souboru jsou indexovány od 1 výše.

V obou uvedených případech nemají zpracovávané soubory čitelný (v textovém smyslu) tvar. Položky souborů však mohou být libovolného privátního typu, je proto možné číst a zapisovat např. celé záznamy.

Pro práci s textovými soubory je k dispozici modul `TEXT_IO`. Byl použit v předchozích příkladech, některé jím poskytované služby již známe. Exportuje operace pro práci s řetězci a se znaky a čtyři generické moduly.

Jeho strukturu znázorňuje obr.3.9.

Jsou k dispozici přetížené procedury pro I/O na standardní zařízení a do textových souborů. Doplňují je procedury pro formátování textu a pro manipulaci se soubory.

I/O operace pro znaky a řetězce

Generický modul INTEGER_IO
I/O operace s celými čísly

Generický modul FLOAT_IO
I/O operace s čísly v poh. řád. čárce

Generický modul FIXED_IO
I/O operace s čísly v pevné ř. čárce

Generický modul ENUMERATION_IO
I/O operace s výčtovými hodnotami

Obr. 9.1: Struktura modulu TEXT_IO

Negenerické prostředky modulu TEXT_IO

Pokud pracujeme se standardním I/O zařízením, máme k dispozici procedury:

```
procedure PUT(ITEM: in CHARACTER);
procedure PUT(ITEM: in STRING);
procedure PUT_LINE(ITEM: in STRING);
procedure GET(ITEM: out CHARACTER);
procedure GET(ITEM: out STRING);
procedure GET_LINE(ITEM: out STRING; LAST: out NATURAL);
```

U posledně jmenované je v druhém parametru vrácena délka přečteného řetězce. Dále jsou použitelné:

SET_COL (*číslo_sloupce*); – nelze se ale vracet k nižším číslům

SET_LINE (*číslo_řádky*); – nelze se ale vracet k nižším číslům

SET_LINE_LENGTH (*délka_řádky*);

SET_PAGE_LENGTH (*délka_stránky*);

NEW_LINE (*počet_řádků*) – nemá-li parametr, pak odřádkuje o 1 ř.

NEW_PAGE;

SKIP_PAGE;

Zjištění současných hodnot je možné funkcemi:

LINE_LENGTH	–	bezparametrová, vrací počet sloupců řádky,
PAGE_LENGTH	–	bezparametrová, vrací počet řádků stránky,
END_OF_LINE	–	vrací pravdivostní hodnotu,
END_OF_PAGE	–	vrací pravdivostní hodnotu,
COL	–	bezparametrová, vrací číslo aktuálního sloupce,
LINE	–	bezparametrová, vrací číslo aktuálního řádku,
PAGE	–	bezparametrová, vrací číslo aktuální stránky.

V případě, že pracujeme s textovým souborem, jsou použitelné všechny výše citované operace. Mají pouze navíc jeden in parametr, který je v poziční notaci uváděn jako prvý a určuje soubor, na nějž se operace vztahuje. Např. specifikace PUT pro znak má tvar:

```
procedure PUT(FILE: in FILE_TYPE; ITEM: in CHARACTER);
```

Navíc jsou k dispozici procedury:

```
procedure CREATE(FILE: in out FILE_TYPE;--interní jméno soub.
                 MODE: in FILE_MODE:=OUT_FILE;
                 NAME: in STRING:="";--vnější jméno souboru
                 FORM: in STRING:="");--speciální informace
```

Poznámka:

Typy FILE_TYPE a FILE_MODE jsou exportovány modulem TEXT_IO. Parametr FILE určuje jméno souboru používané v programu, např. v příkazu PUT. Parametr NAME je jménem tohoto souboru ve vnějším systému (např.

DATA.TXT). Neuvede-li se, je soubor považován za dočasný a při skončení programu je zrušen. Parametr FORM vyžadují některé implementace. Obsahuje např. údaj o maxim. velikosti souboru, password a pod. Třetí a čtvrtý parametr mají iniciované hodnoty a není proto nutné je uvádět. Jakmile soubor je vytvořen a má vnější jméno, je možné se na něj v programech obracet prostřednictvím i dalších uvedených procedur.

```
procedure OPEN(FILE: in out FILE_TYPE;
               MODE: in FILE_MODE;--OUT_FILE nebo IN_FILE
               NAME: in STRING;
               FORM: in STRING:="");
procedure CLOSE(FILE: in out FILE_TYPE);
procedure DELETE(FILE: in out FILE_NAME);
procedure RESET(FILE: in out FILE_NAME; MODE: in FILE_MODE);
procedure RESET(FILE: in out FILE_TYPE);--zachová hodn. MODE
function MODE(FILE: in FILE_TYPE) return FILE_MODE;
function NAME(FILE: in FILE_TYPE) return STRING;
function FORM(FILE: in FILE_TYPE) return STRING;
function IS_OPEN(FILE: in FILE_TYPE) return BOOLEAN;
function END_OF_FILE(FILE: in FILE_TYPE) return BOOLEAN;
```

Poslední funkce je použitelná i v bezparametrovém tvaru, testuje-li se konec souboru na standardním zařízení. Je-li toto zařízení interaktivní, bývá použití END_OF_FILE problémové.

Pro zjištění a určení přednastaveného I/O zařízení a zjištění standardního I/O zařízení jsou k dispozici podprogramy:

```
procedure SET_INPUT(FILE: in FILE_TYPE);
procedure SET_OUTPUT(FILE: in FILE_TYPE);
function CURRENT_INPUT return FILE_TYPE;
function CURRENT_OUTPUT return FILE_TYPE;
function STANDARD_INPUT return FILE_TYPE;
function STANDARD_OUTPUT return FILE_TYPE;
```

Příklad 3.16. Vytvoření souboru

```
with TEXT_IO; use TEXT_IO;
```

```
procedure ZAPISUJ is
      SOUBOR: FILE_TYPE;    --deklaruje soubor
      LINE: STRING(1..0);
      LINE_LENGTH: NATURAL;
begin
      CREATE(SOUBOR, OUT_FILE,"VYSTUP.TXT");--vytvoří soubor
      PUT_LINE("napis nejakou moudrost (max. 80 znaku)");
      PUT_LINE("napis ""DOST"" kdyz jiz zadnou nevis");
      GET_LINE( LINE, LINE_LENGTH);
      while LINE(1..LINE_LENGTH) /= "DOST" loop
            PUT_LINE(SOUBOR, LINE(1..LINE_LENGTH); --zapíše
            GET_LINE(LINE, LINE_LENGTH); --čte z terminálu
      end loop;
      CLOSE(SOUBOR);      --uzavře soubor
end ZAPISUJ;
```

Generické prostředky modulu TEXT_IO

Doplňují možnosti práce s textovými soubory o přetížené procedury `PUT` a `GET` pro číselné typy a výčtové typy. Dovolují při I/O operacích pracovat s implicitním I/O zařízením, s explicitně uvedeným souborem a i s interní řetězcovou proměnnou. Poslední z možností zde pouze připomínáme, ve výkladu není rozvedena, bude ale ilustrována příkladem.

Modul `INTEGER_IO` poskytuje tyto tvary procedur `GET` a `PUT`:

```
procedure GET(ITEM: in NUM;        --NUM je generickým
                                   --parametrem
              WIDTH: in FIELD:=0);--WIDTH udává počet
                                   --cifer čísla, pro 0
                                   --je volný formát, což
                                   --je nejčastější případ
procedure PUT(ITEM: in NUM;
              WIDTH: in FIELD:=NUM'WIDTH;
              BASE: in NUMBER_BASE:=10);--volitelný základ
                                         --čísla
```

V uvedeném tvaru procedury pracují s implicitním I/O zařízením. Jejich mutaci (zde pro stručnost neuváděnou), kde prvým parametrem je zadáno jméno souboru, je snadné si představit. Totéž platí i pro případ ostatních generických modulů z TEXT_IO.

Modul FLOAT_IO dává k dispozici tytéž služby, ovšem pro typy v pohyblivé řádové čárce.

```
procedure GET(ITEM: out NUM; WIDTH: in FIELD := 0);
```

Je-li zadán parametr WIDTH, očekává se, že přečtený údaj s tímto počtem znaků je číslem v pohyblivé řádové čárce.

```
procedure PUT(ITEM: in NUM;
              FORE: in FIELD := 2;        --míst před tečkou
              AFT: in FIELD := NUM'DIGITS;--míst za tečkou
              EXP: in FIELD := 3);        --míst exponentu
```

Z modulu ENUMERATION_IO jsou exportovány procedury tvaru:

```
procedure GET(ITEM: ENUM); --ENUM je generickým parametrem

procedure PUT(ITEM: in ENUM;
              WIDTH: in FIELD := 0 --0 znamená, že literál
                                   --bude vypsán jen na
                                   --potřebný počet míst
              SET: in TYPE_SET:=UPPER_CASE);
                                   --altern.je LOWER_CASE
```

Příklad 3.17

```
with TEXT_IO; use TEXT_IO;
procedure VYSTUPY is
     package INT_IO is new INTEGER_IO(INTEGER);
     package FLT_IO is new FLOAT_IO(FLOAT);
     package BOOL_IO is new ENUMERATION_IO(BOOLEAN);
     use INT_IO, FLT_IO, BOOL_IO;
     VYSTUP: STRING(1..8);
begin
     PUT(TO=>VYSTUP,ITEM=>22, BASE=>2);--zapíše cel.čís.
                                       --do řetězce
     PUT(VYSTUP); --2#10110# vypíše na terminál
```

```
    PUT(TO=>VYSTUP,ITEM=>3.141,AFT=>1,EXP=>0);
                                --float p. číslo
    PUT(VYSTUP):--     3.4  vypíše na terminál
    PUT(TO=>VYSTUP,ITEM=>TRUE,SET=>LOWER_CASE);--literál
    PUT(VYSTUP);--true a čtyři mezery vypíše na terminál
end VYSTUPY;
```

Dodatek A

A.1 Základní návod pro práci s jazykem Digital ADA

Vytvoření programu v jazyce **ADA** probíhá v následujících krocích [1]:

1. Vytvoření pracovního adresáře (využívaného pro zdrojové soubory programu). Pro každou aplikaci (program) je vhodné vytvořit separátní pracovní adresář.

2. Zdrojové soubory by měly mít příponu `.ada`. Dále je vhodné v názvu souborů obsahujících samostatně překládané moduly rozlišit, zda se jedná o specifikaci nebo o tělo modulu, např. použijeme `modul-s.ada` a `modul-b.ada` pro soubory obsahující specifikaci a tělo modulu (package).

3. Vytvoření knihovny pro objektové moduly (vytvořené překladem zdrojových modulů). Provede se z pracovního adresáře příkazem

   `amklib` *adalib*

   kde *adalib* je zvolené jméno podadresáře, ve kterém bude knihovna vytvořena. Do knihovny se překopírují přeložené standardní moduly (package ap.) dostupné v příslušné instalaci. Dále se pak do ní přidávají objektové moduly vzniklé překladem zdrojových modulů uživatelského programu. Uživatelské moduly se přidávají do knihovny automaticky - každým překladem se nahradí stará verze (pokud tam je) novou. Knihovní adresář *adalib* je vytvořen pod pracovním adresářem.

4. Definování aktuální knihovny (tzv. *kontext*). Provede se z pracovniho adresáře příkazem

[1] Uživatelsky volitelná jména v příkazech jsou psána kurzívou.

`setenv ADALIB @`*adalib*

kde *adalib* je jméno knihovniho adresáře.

!!! Po každém přihlášení do systému je zapotřebí přepnout se na pracovní adresář a definovat aktuální knihovnu.

5. Samostatný překlad zdrojového souboru *file*`.ada` se nejlépe provede příkazem

   `ada -V` *file*`.ada`

   Volba `-V` zajistí vytvoření protokolu o překladu, který se uloží do souboru *file*`.l`. Při bezchybném překladu se automaticky uloží do knihovny všechny moduly (**ADA** units), jejichž text se nachází v souboru *file*`.ada`. Pokud je specifikace modulu v samostatném souboru, musí se přeložit dřív než tělo.

6. Sestavení programu se provede příkazem

   `ald -o` *main_exe main_proc*

   kde *main_exe* je jméno souboru se sestaveným programem a *main_proc* je jméno procedury [2]. (obvykle samostatně kompilované), která slouží jako hlavní program. Všechny moduly (package, samostatně kompilované procedury), které využívá hlavní program už musí být přeložené v knihovně [3]. Sestavený program se nachází v pracovním adresáři.

   Můžeme také použít rovnou příkaz

   `ald` *main_proc*

   a sestavený program nalezneme v pracovním adresáři pod jménem `a.out`.

7. Pokud změníme zdrojový text některého modulu a nechceme opakovat jednotlivě překlad všech závislých modulů, využijeme příkaz

   `amake` em main_proc

[2]!!! Pozor - v příkazu `ald` se uvádí jméno procedury *main_proc* a nikoliv jméno zdrojoveho souboru, ve kterém je procedura obsažena. Nejlépe je dát proceduře hlavního programu i jejímu zdrojovému souboru stejné jméno.

[3]!!! Obecně platí, že v knihovnách jazyka **ADA** se vedou jména modulů (**ADA** units - package, procedury) a nikoliv jména zdrojových sou- borů, ve kterých jsou moduly obsaženy.

pro překlad procedury hlavního programu. Automaticky se vyhledají a ve správném pořadí provedou všechny potřebné překlady. Pokud chceme navíc ještě provést sestavení programu, použijeme příkaz

`amake -l -o` *main_exe main_proc*

8. Spuštění výpočtu se provede příkazem

   main_exe

   kde *main_exe* je jméno použité v příkazu `ald`, tj. jméno souboru se spustitelným programem.

9. Ladění programu je u konkurenčních výpočtů mnohem obtížnější, proto je součástí prostředí ADY ladicí program (*debugger*) `dbx`. Neodladěné moduly je třeba překládat s volbou `-g`, například

   `ada -g` *file*`.ada`

   Program se normálním způsobem sestaví a spustí příkazem:

   `dbx` *main_exe*

   Po ohlášení ladicího programu lze využitím příkazu `help` získat seznam použitelných příkazů (definice *breakpointů* ap.). Návrat z ladicího programu se provede kdykoliv příkazem `quit`.

10. Ještě uvedeme užitečné příkazy potřebné pro práci s knihovnou přeložených modulů. Obsah knihovny se zobrazí příkazem

    `als`

    Knihovní moduly, jejichž jména končí `-s` představují přeložené specifikace, zakončení `-b` představuje přeložené tělo modulu.

    Zajímá–li nás zdrojový text modulu (důležité zejména u specifikací standardních knihovních modulů, např. `TEXT_IO`), využijeme příkaz

    `acat TEXT_IO`

    pro zobrazení zdrojového textu specifikace i těla na standardní výstup, nebo

    `acat TEXT_IO-s`,

    jestliže nás zajímá jen specifikace.

Modul můžeme vypustit (specifikaci i tělo) z knihovny například příkazem

`arm -v` em muj_modul

Volba `-v` zajistí zprávu, které moduly byly vypuštěny. Například jenom specifikace se vypustí příkazem

`arm -v` *muj_modul*`-s`

Celou knihovnu zlikvidujeme příkazem

`armlib -i adalib`

kde `adalib` je jméno adresáře a volba `-i` zajistí dotaz, zda se má akce skutečně provést.

11. Stručný anglicky psaný manuál pro překladač programovacího jazyka **ADA** je možné prohlížet (ovládání klávesou `d`) po použití příkazu

    `man ada`

Literatura

[ADA91] DEC Ada Language Reference Manual, DEC 1991, 589 s.

[Bar97] BARNES, J.: Introduction to ADA95, http://www.ada9x.uk, 1997

[Digital] Firemní materiály Digital.

[Skan89] SKANSHOLM, J.: Ada from the beginning. Adison-Wesley 1989, 617s. 3rd ed. Springer–Verlag, Berlin, New York, 1985, 202 s.

[Young88] YOUNG S.J.: Programovací jazyky pro RT aplikace. SNTL, Praha 1988, 388 s.