æ

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

TERMÉSZETTUDOMÁNYI KAR

Kozics Sándor

Az Ada programozási nyelv

Budapest, 1992

Előszó

Ez a jegyzet a programozó matematikus szakot végző hallgatók számára készült. Feltételezi a Bevezetés a programozáshoz c. tárgy anyagának, s a Modula-2 nyelvnek az ismeretét.

A jegyzet célja a nyelv önálló tanulásának támogatása. Ennek érdekében a jegyzet végén különböző nehézségű feladatokat is elhelyeztünk.

A jegyzet készítéséhez felhasználtuk az alábbi cikkeket és könyveket:

[1]Coat, D., "Pascal, Ada and Modula-2", BYTE 8(1984) 215-232.

[2]Young, S.J., An Introduction to Ada (Ellis Horwood Limited, 1983).

[3]Pyle, I.C., Az Ada programozási nyelv (Műszaki Könyvkiadó, Budapest, 1987).

[4]Reference Manual for the Ada Programming Language (january 1983)

[5]Gehani, N., Ada, An Advanced Introduction (Prentice-Hall Inc., Englewood Cliffs, 1984.)

[6]Miller, N.E., Petersen, C.G., File Structures with Ada (The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1990.)

[7]Dershem, H.L., Jipping, M.J., Programming Languages: Structures and Models (Wadsworth Publishing Company, Belmont, California, 1990.)

1. Az Ada nyelv áttekintése

1.1. Az Ada történeti háttere

Az 1970 -es évek elején az US DoD (United States Department of Defense) úgy döntött, hogy tennie kell a software-költségek rohamos növekedése ellen. 1973 -ban például a DoD software-költsége 3 milliárd dollár körül volt, aminek 56 %-át az alap-software jellegű rendszerek fejlesztése emésztette fel. Összehasonlításul: adatfeldolgozásra 19 %-ot, a tudományos programokra 5 %-ot költöttek.

A legnagyobb költségmegtakarítás nyilvánvalóan úgy érhető el, ha ezek közül az alap-software rendszerek költségeit csökkentik. Míg az adatfeldolgozás és a tudományos programok nyelve széles körben a COBOL és a FORTRAN, a rendszerprogramok készítéséhez nem alakult ki hagyományos nyelv. E célra számos nyelvet használtak, mint pl. a JOVIAL, CMS-2, TACPOL, SPL/1, és még sok mást. Ezért arra lehetett következtetni, hogy jótékonyan hatna a költségekre ennek a területnek az egységesítése. Ennek érdekében szervezte meg az US DoD a HOL (High Order Language) elnevezésű munkát, kettős céllal. Az első, rövidtávú cél egy vagy több ideiglenesen használható nyelv kiválasztása, a második, hosszútávú cél pedig egy nagy megbízhatóságú alap-software készítésére alkalmas nyelvnek a kijelölése ill. definiálása volt.

Az első célt hamarosan elérték, és a következő nyelveket választották ki: TACPOL, CMS-2, SPL/1, JOVIAL J3 és JOVIAL J73.

A hosszútávú cél elérésének első lépése a követelmények rögzítése és a létező nyelvek értékelése volt. A követelményeket egyre finomodó dokumentumokban rögzítették, amelyek a STRAWMAN, WOODENMAN, TINMAN, IRONMAN, végül a STEELMAN nevet kapták (szalma-, fa-, bádog-, vas-, acélember). Ezeket a dokumentumokat széleskörű és nyílt vitákon finomították.

A létező nyelvek értékelését 1976-ban, a munka harmadik szakaszában végezték el. A FORTRAN, COBOL, PL/I, HAL/S, TACPOL, CMS-2, CS-4, SPL/1, J3B, Algol 60, Algol 68, CORAL 66, Pascal, SIMULA 67, LIS, LTR, RTL/2, EUCLID, PDL2, PEARL, MORAL és EL-1 nyelvek értékelését publikálták. Az értékelés eredményét négy pontban foglalták össze:

1. Nincs olyan létező nyelv, ami egymagában megfelelne a célnak.
2. Egyetlen nyelvet kell kialakítani.
3. A követelmények definiálásának meg kell maradnia a "state of art" szintjén.
4. A fejlesztést a meglevő értékek felhasználásával kell végezni.

Az értékelt nyelveket három osztályba sorolták:

- nem használható fel : Ide tartoznak az elavult, vagy elhibázott nyelvek.

Ezekről nem adtak részletesebb elemzést. Ebbe a kategóriába került pl. a FORTRAN és a CORAL 66.

- megfelelő : Ide kerültek azok a nyelvek, amelyek ugyan közvetlenül nem alkalmasak, de néhány elemük felhasználható. Ebbe a csoportba került pl. az RTL/2 és a LIS.
- alapnyelvként használható : Három ilyen nyelvet találtak, a Pascalt, a PL/I-et és az ALGOL 68-at, amiket az új nyelv tervezésének kiindulópontjául ajánlottak.

Ezzel a TINMAN dokumentumot befejezték, és a következő szakaszban négy vállalkozóra bízták a tervezést, akik valamelyik alapnak alkalmas nyelvből indulhattak ki. Tizenhét jelentkezőből választották ki azt a négyet, akik párhuzamosan folytatták a nyelv tervezését. A négy vállalkozót színekkel kódolták:

CII Honeywell Bull	—	Zöld
Intermetrics	—	Piros
Softech	—	Kék
SRI International	—	Sárga

Mind a négyen a Pascalt választották alapnyelvnek. (A színkódolást az elfogulatlan véleményezés érdekében vezették be.)

Az első fázis végén Kék és Sárga kiesett a küzdelemből, de Zöld és Piros folytatták a versengést. Ezt a tervezési szakaszt 1979 májusában fejezték be, és a győztes Zöld lett.

A Zöld és Piros közötti választás nem volt könnyű a DoD számára. Azt mondták, hogy a Piros egy jobb nyelv lehetőségét hordozta magában, de az IRONMAN második szakaszában ez a nyelv olyan mértékben megváltozott, hogy a tervezők nem tudták befejezni a munkát. A Zöld nyelv keveset változott, és a definíciója jóval teljesebb volt, így a Zöld választása járt kisebb kockázattal. Piros nyelv néhány fogalmának implementálása igen problematikus volt, míg a Zöld nyelvvel már lehetett dolgozni.

Ebben az időben határozta el a DoD, hogy az új nyelvet Adának fogják hívni. Ada — Lord Byron lánya, és Babbage asszisztense volt. Ő volt a világon az első programozó.

A HOL ezzel elérkezett az utolsó szakaszba. Ennek a szakasznak a feladata a nyelv végső finomítása és definiálása volt, s itt nagy részt tett ki a Tesztelési és Értékelési munka. Ez utóbbi abban állt, hogy különböző csoportok programokat írtak Ada nyelven, hogy annak használhatóságáról meggyőződjenek. Ezekről a munkákról 82 elemzés készült, amiket 1979 októberében a bostoni konferencián adtak elő. Az általános vélemény az volt, hogy a nyelv jó, de néhány ponton módosításra szorul.

A Tesztelés és Értékelés nem volt minden probléma nélküli. A megadott idő nem volt elég ahhoz, hogy Ada-stílusban tervezzék meg a programokat, ezért általában meglevő programokat használtak fel, amiket azután eljárásonként

átírtak Adára. Így az Ada alkalmasságát nagy programok írására végül is nem sikerült tesztelni. Emellett nagyobb programok esetében problémát okoztak a nyelv teszteléséhez elkészített interpreter zavarai. Ezt az interpretert még az IRONMAN második szakaszában abból a célból fejlesztették ki, hogy megmutassák, a nyelv végrehajtható. A harmadik probléma az Ada kézikönyv tömörsége volt. Bár a kézikönyv tökéletesen leírta a nyelvet, mint oktatókönyv nem volt túlságosan jó. Ezzel együtt a Tesztelési és Értékelési szakaszban igen alapos és intenzív tesztelést végeztek, amelynek az eredményei alapján a nyelvet tovább finomították.

1.2. Az Ada rövid jellemzése

Az Ada tervezésének [1,2,3,4,5,6] fő szempontjai a program megbízhatósága, karbantarthatósága, emberközelisége (olvashatósága) és hatékonysága voltak. A nyelv alapja a Pascal, de néhány vonást átvett a következő nyelvekből is: Euclid, Lis, Mesa, Modula, Sue, Algol 68, Simula 67, Alphard, CLU.

Az Ada programegységei az
- alprogramok, amelyek a számításokat elvégző algoritmusból állnak,
- a package, ami alprogramok, típusok, konstansok és változók gyűjteménye,
- a task, ami párhuzamosan végrehajtható számításokat tartalmaz, és
- a generic, ami típussal és alprogrammal is paraméterezhető makro-szerű package-t vagy alprogramot jelent.

A programegységekből (kivéve a taskot) kialakíthatók önállóan fordítható egységek. Ezek között a főprogram egy olyan eljárás lehet, amelynek nincs paramétere.

A programegységek két részből állnak: specifikációs részből, ami a más egységekből látható információt tartalmazza, és a törzsből, ami az implementációt tartalmazza, és ami más egységekből nem látható.

A specifikációs rész és a törzs szétválasztása, és a külön való fordítás lehetővé teszi nagymértékben független programkomponensek tervezését, írását és tesztelését.

A programegység törzse két részből áll: deklarációs részből, ami definiálja a programegységben használt konstansokat, változókat, hibaeseményeket (**exception**) és programegységeket, és egy utasítássorozatból, ami definiálja a programegység végrehajtásának hatását. A törzs végrehajtása ennek megfelelően két lépésben történik: az első lépés a deklaráció kiértékelése, a második az utasítássorozat végrehajtása. A végrehajtás hibaeseményeket is előidézhet (ilyen pl. a túlcsordulás, az indextúllépés, vagy egy, a felhasználó által definiált hiba fellépése), és ezeknek a hibáknak a lekezelésére a törzs végén speciális programrészek (exception handler) helyezhetők el.

Az Ada erősen típusos nyelv. Minden objektum rendelkezik egy a fordítás során egyértelműen megállapítható típussal, s a típusok megfelelő használata ellenőrzött. Vannak a nyelvnek elemi típusai, mint a felsorolási típus és a különböző numerikus típusok, valamint összetett típusai, mint a rekord, a tömb és a pointer (**access**) típus. A felhasználó is definiálhat típust, ún. private típust úgy, hogy megadja az értékhalmazt és a műveletek programjait egy package programegységben.

Az Ada az input-output-ot predefinit generic package-kben definiálja.

A lefordított Ada egységek egy speciális könyvtárban, egy adatbázisban helyezkednek el. Kezdetben a könyvtár csak predefinit package-ket tartalmaz, ezekhez illesztheti hozzá a felhasználó a maga könyvtári egységeit. Egy Ada program kifejlesztése a könyvtári egységek adatbázisának felépítését jelenti.

Az Ada a nyelv különböző variánsainak kialakulása ellen is védekezik. A nyelv szabványa tartalmazza azt a kitételt, hogy *csak az a fordítóprogram használhatja az "Ada fordítóprogram" megnevezést, ami a nyelv minden szabályát pontosan alkalmazza, s ami nem vezet be sem módosításokat, sem kiterjesztéseket.* Azt, hogy egy fordítóprogram viselheti-e az "Ada fordítóprogram" címkét, egy New Yorkban felállított intézetben döntik el. Az ide elküldött fordítóprogram validálása úgy történik, hogy lefordítanak és lefuttatnak egy több ezer programból álló tesztsorozatot.

Az Ada szintaxisának leírására a további fejezetekben egy kiterjesztett BNF nyelvet használunk. A bal- és jobboldalt *::=* jellel, a jobboldal alternatíváit | jellel választjuk el. A nemterminális elemeket < és > jelek közé írjuk. A szögletes zárójelel közé zárt rész tetszés szerint elhagyható vagy használható a programban, a kapcsos zárójelbe tett rész pedig tetszőlegesen sokszor (akár nullaszor is) egymás után írható.

2. Lexikális elemek

<graphic character> ::= *<basic graphic character>*
|*<lower case letter>*|*<other special character>*
<basic graphic character> ::= *<upper case letter>*
|*<digit>*|*<special character>*|*<space character>*
<identifier> ::= *<letter>* { [*<underline>*] *<letter or digit>*}

Az Ada az azonosítókban és az alapszavakban nem tesz különbséget a kis- és nagybetűk között. Az azonosító minden karaktere szignifikáns. A nyelvben — a modern gyakorlatnak megfelelően — az alapszavakat külön jelölni nem kell, de ezek kötött szavak, azonosítóként nem használhatók. Vannak a nyelvben predefinit szavak is (pl. a típusok felépítésében használt `INTEGER`, `TRUE`, `STRING` stb.), ezek újradefiniálhatók. Az Adában az a szokás, hogy az alapszavakat kis, az azonosítókat nagybetűkkel írják.

<numeric literal> ::= *<decimal literal>*|*<based literal>*
<decimal literal> ::= *<integer>* [. *<integer>*] [*<exponent>*]
<integer> ::= *<digit>* { [*<underline>*] *<digit>*}
<exponent> ::= `E` [`+`] *<integer>*| `E` `-` *<integer>*
<based literal> ::= *<base>* `#` *<based integer>*
[. *<based integer>*] `#` *<exponent>*
<base> ::= *<integer>*
<based integer> ::= *<extended digit>* { [*<underline>*] *<extended digit>*
<extended digit> ::= *<digit>*|*<letter>*

A számoknak két alapformája van: a decimális és az alapszámmal felírt forma. A számrendszer alapszáma kettőtől tizenhatig bármelyik szám lehet, de maga az alapszám mindig decimális. A számok tagolására használhatjuk az aláhúzásjelet, aminek a szám értéke szempontjából nincs jelentősége.

```
123_456_789             -- decimális egész szám
1E6                     -- decimális egész exponenciális alakban
2#1011#                 -- bináris egész szám
3.14159                 -- decimális valós
16#F.FF#E2              -- hexadecimális valós exponenciális alak
2#1.1111_1111_111#E11   -- a kitevőhöz tartozó alap a számrendszer
                        -- alapszáma, tehát a két utóbbi érték
                        -- egyaránt 4095.0
```

A valós számokat az különbözteti meg az egész számoktól, hogy tizedespontot tartalmaznak.

<character literal> ::= ´ *<graphic character>* ´
<string literal> ::= " {*<graphic character>*} "

Az Ada karakter- és stringliterálokban csak megjeleníthető karaktereket használhatunk. A karakterliterálokat úgy írjuk, hogy a karaktert aposztrófok közé

zárjuk. A stringliterálokat idézőjelek közé tesszük, és nem nyúlhatnak túl a sor végén. A nem megjeleníthető karaktereket karakteres konstans formájában a predefinit `ASCII` package definiálja.

```
´A´ ´a´ ´´´          -- ez egy nagy és kis A betű és egy aposztróf
""                   -- üres string
""""                 -- egy idézőjelet tartalmazó string
"abc" & ´d´ & "ef" & ´g´-- ez a konkatenáció fordítási időben értékelődik ki
ASCII.ESC & "[9B"
```

Az elemi szintaktikai egységek (alapszavak, azonosítók, literálok stb.) közé tetszőlegesen sok szóköz elhelyezhető, de ezek belsejében nem lehet szóköz. (Nem lehet tehát szóköz a több karakterrel kifejezett szimbólumok, pl. `:=`, `/=`, `=>` stb. belsejében sem.)

A megjegyzést két mínuszjel vezeti be, és a sor végéig tart.

```
-- ez egy megjegyzés .
```

Megjegyzés a program bármely sorában elhelyezhető. A megjegyzésnek nincs hatása sem a program szintaktikus helyességére, sem a jelentésére.

3. Deklarációk

Deklarációnak nevezzük egy azonosítónak és egy egyednek az összekapcsolását. Általában maga a deklarációs utasítás tartalmazza az egyed definícióját is (pl. változó és konstans esetén), egyes esetekben azonban a deklaráció és a definíció szét is válhat (pl. típus és alprogram esetén). Ha egy azonosítóra hivatkozunk, annak a szövegben korábban deklaráltnak kell lennie (kivéve a címkék esetét, ld. 5.8.).

Az egyedek definíciójának kiértékelése futási időben történik, emiatt pl. a típusdefiníció változót és függvényhívást is tartalmazhat.

<declarative part> ::= {*<basic declarative item>*}{*<later declarative item>*}
<basic declarative item> ::= *<basic declaration>*|*<represetation clause>*
|*<use clause>*
<later declarative item> ::= *<body>*|*<subprogram declaration>*
|*<package declaration>*|*<task declaration>*|*<generic declaration>*
|*<use clause>*|*<generic instantiation>*
<body> ::= *<proper body>*|*<body stub>*
<proper body> ::= *<subprogram body>*|*<package body>*|*<task body>*
<name> ::= *<simple name>*|*<character literal>*|*<operator symbol>*
|*<indexed component>*|*<slice>*|*<selected component>*|*<attribute>*
<simple name> ::= *<identifier>*
<basic declaration> ::= *<object declaration>*|*<number declaration>*
|*<type declaration>*|*<subtype declaration>*|*<subprogram declaration>*
|*<package declaration>*|*<task declaration>*|*<generic declaration>*
|*<exception declaration>*|*<generic instantiation>*|*<renaming declaration>*
|*<deferred constant declaration>*

A különféle elemek deklarációjának és a deklarációs részben levő definícióknak a sorrendje a következő: előbb a típus- és objektumdeklarációk, valamint az alprogramspecifikációk következnek, majd ezek után jöhet a taskok, package-k és alprogramok törzse. A taskok és package-k specifikációja bárhol lehet, de annak meg kell előznie a hozzá tartozó törzset.

Ha egy deklarációs részben szerepel egy alprogram, package vagy task specifikációja, ugyanebben a deklarációs részben kell lennie a megfelelő törzsnek is.

<object declaration> ::= *<identifier list>* : [**constant**]
<subtype indication> [:= *<expression>*] ;
|*<identifier list>* : [**constant**]
<constrained array definition> [:= *<expression>*] ;

Objektumnak nevezzük azokat az elemeket, amelyeknek valamilyen típusú értéke lehet. Objektum a változó, a konstans, a formális paraméter, a ciklusváltozó, a tömbelem és a résztömb. Objektum definiálásakor az objektum kap-

hat kezdőértéket is. Az értéket adó komponens értéke futási időben kerül meghatározásra.

```
COUNT, SUM : INTEGER;
SORTED : BOOLEAN := FALSE;
COLOR_TABLE : array (1 .. N) of COLOR;
OPTION : BIT_VECTOR(1 .. 10) := (OPTION'RANGE => TRUE);
K : INTEGER := K * K;    -- hibás
INTEGER : INTEGER;       -- hibás
```

Azokat a deklarációkat, amelyekben a jobboldal ugyanaz, összevonhatjuk egyetlen deklarációs utasításba (ilyen esetet mutat be a fenti példák közül az első). Ez az összevonás nem változtatja meg a program jelentését, tehát az alábbi deklarációk ekvivalensek (a **new** dinamikus memória allokálására szolgáló művelet, ld. 4.7.):

```
P1, P2 : POINTER := new INTEGER'(0);
```

illetve

```
P1 : POINTER := new INTEGER'(0);
P2 : POINTER := new INTEGER'(0);
```

A konstansok definíciójában a kettőspont után a **constant** szót kell írni. A konstans értékét megadó kifejezés is futási időben értékelődik ki.

```
LOW_LIMIT : constant INTEGER := LIMIT / 10;
TOLERANCE : constant COEFFICIENT := DISPERSION(1.15);
```

Egy objektumdeklaráció futás közbeni kiértékelése a következő lépésekben zajlik le:

a) Kiértékelésre kerül az objektum altípusa.
b) Kiértékelésre kerül az objektum kezdőértékét megadó kifejezés, vagy ha ilyen nincs, akkor az a) pontban meghatározott típus definíciójában megadott kezdőértékadás.
c) Létrejön (allokálásra kerül) az objektum.
d) Ha van az objektumnak, vagy valamelyik komponensének kezdeti értéke, felveszi azt.

<number declaration> ::= *<identifier list>* : **constant** :=
    <universal static expression> ;
<identifier list> ::= *<identifier>* { , *<identifier>*}

Az Adában a számok nemcsak mint a típusok típusértékei léteznek, hanem használhatók a típusoktól függetlenül is, mint ahogy pl. a fizikában is használnak számokat mértékegység nélkül is. (Az Adában ilyen univerzális számértékeket adnak vissza egyes attributumok, pl. amelyik megadja a valós számok gépi ábrázolásában a mantissza hosszát bitben, amelyik megadja a program rendelkezésére álló memória méretét, vagy amelyik megadja, hogy egy adott rekordtípuson belül valamelyik mező hányadik bitpozíción kezdődik stb.) Az ilyen, univerzális számokat elláthatjuk névvel, és így definiálhatunk ún. univerzális ("típustalan") konstansokat. Ezekben a konstansdefiníciókban a kifejezések kiértékelése fordítási időben történik. Az uni-

verzális számok értéktartományára az Ada nem vezet be korlátozást.

```
LIMIT : constant := 100;
ZERO : constant := 0;
SIZE : constant := LIMIT * 2;
LIMIT2 : constant INTEGER := 100;
VAL1 : INTEGER;
VAL2 : MY_INT;
VAL1 := LIMIT2;
VAL1 := LIMIT;
VAL2 := LIMIT;
PI : constant := 3.14159;
VAL2 := LIMIT2; -- hibás (konvertálni kellene!)
```

4. Típusok

Az Ada a programok megbízhatóságának növelése céljából igen erős típusfogalmat használ. Ez a következőket jelenti:

- Minden objektumnak egyetlen típusa van.
- Minden típus egy értékhalmazt és egy művelethalmazt definiál, amely műveletek ezekre az értékekre alkalmazhatók. Egy típus nem lehet komponense saját magának.
- Az érték és az értéket kapó objektum típusának minden értékadásban meg kell egyezni.
- Ha egy értékre alkalmazunk egy típusműveletet, annak a típusműveletek halmazában szerepelnie kell.

<type declaration> ::= <full type declaration>
|<incomplete type declaration>|<private type declaration>
<full type declaration> ::= **type** *<identifier> [<discriminant part>]* **is**
<type definition> ;
<type definition> ::= <enumeration type definition>
|<integer type definition>|<real type definition>
|<array type definition>|<record type definition>
|<access type definition>|<incomplete type definition>

Az Adában a következő típusosztályok léteznek :

- skalár típusok, ezen belül diszkrét és valós típusok;
- diszkrét típusok, ezen belül felsorolási és egész típusok;
- összetett típusok (tömb és rekord típusok);
- access típus;
- private típusok;
- task típusok.

<attribute> ::= <prefix> ´ *<attribute designator>*
<attribute designator> ::= <simple name>
[(*<universal static expression>*) *]*
<prefix> ::= <name>|<function call>

Az Ada az egyes típusosztályokhoz is bevezet típusműveleteket, ún. attributumokat. Ezeket az osztály minden típusára használhatjuk. Pl. az egész típusra alkalmazható a `FIRST` és a `LAST` attributum, és a típus legkisebb ill. legnagyobb értékére `INTEGER´FIRST` ill. `INTEGER´LAST` formában hivatkozhatunk. Az `=` és `/=` reláció a **limited private** típus (ld. 7.4.) kivételével bármilyen típusú objektumra alkalmazható.

A `V´ADDRESS` attributum bármilyen `V` objektumra alkalmazható, s megadja az objektumnak a memóriabeli címét. A `T´SIZE`, `V´SIZE` attributum bármilyen `T` típusra és `V` objektumra alkalmazható, s megadja a típus értékeinek

ill. az objektumnak a tárolásához szükséges bitek számát. (Mindkét attributumnak elsősorban más nyelvű, pl. assembly rutinok hívásakor van szerepe.)

Minden típus csak önmagával ekvivalens. Különböző definíciók különböző típusokat definiálnak.

```
A : array (1 .. 10) of BOOLEAN;
B : array (1 .. 10) of BOOLEAN;  -- A és B típusa különböző
X, Y:array (1 .. 10) of BOOLEAN; -- X és Y típusa különböző
```

4.1. Az altípus fogalma

<subtype declaration> ::= **subtype** *<identifier>* **is** *<subtype indication>* ;
<subtype indication> ::= *<type indication>* [*<constraint>*]
<type mark> ::= *<type name>*|*<subtype name>*
<constraint> ::= *<range constraint>*|*<floating point constraint>*
|*<fixed point constraint>*|*<index constraint>*|*<discriminant constraint>*

Az egyik eszköz, amit a biztonság növelésére az Ada bevezet, az altípus fogalma. Altípust a típus értékhalmazának megszorításával (azaz erősebb típusinvariáns megadásával) állíthatunk elő. A megszorítás módja függ a típusosztálytól (pl. diszkrét típusok esetében a megszorítás az értéktartomány egy intervallumának kiválasztása). *Az altípusképzés nem vezet be új típust*, ennek csak a futás közbeni ellenőrzésben van szerepe. Ez a megoldás ezen kívül értékes információkat ad a program olvasójának a változó használatáról.

```
type COLOR is (WHITE, RED, YELLOW, GREEN, BLUE, BROWN, BLACK);
type MATRIX is
     array (INTEGER range <>,INTEGER range <>) of REAL;
type TEXT (MAX:POSITIVE) is
  record
    LENGTH : NATURAL := 0;
    VALUE : array (1 .. MAX) of CHARACTER;
  end record;
subtype RAINBOW is COLOR range RED .. BLUE;
subtype RED_BLUE is RAINBOW;
subtype SQUARE is MATRIX(1 .. 10, 1 .. 10);
subtype LINE is TEXT(80);
```

Bármely `T` altípusra alkalmazható a `T´BASE` attributum, ami megadja a `T` bázistípusát. `T´BASE` csak attributumok prefixeként használható, pl. `T´BASE´FIRST`.

(Megjegyzés. Az altípusképzés az értékhalmazra vonatkozó művelet. A programozás szempontjából hasznosak lennének az olyan altípusképző műveletek is, amelyek a típus művelethalmazára vonatkoznak, ilyeneket azonban az Ada nem tartalmaz. Ugyanilyen szempontból hasznosak lennének a típusdefiniálásnak olyan formái is, amikor az új típus művelethalmaza kibővül.)

4.2. Származtatott típusok

Az Ada nemcsak a különböző típusok összekeveredése ellen ad védelmet, hanem a programban szereplő különböző, de ugyanolyan típusú objektumoknak a szintaxis szintjén való elkülönítésére is eszközökkel szolgál. Ez az eszköz az új típus *származtatás*a egy már létező típusból.

<derived type definition> ::= **new** *<subtype indication>*

A származtatott típus átveszi az eredeti típus struktúráját, értékhalmazát és a műveleteit, átveszi annak kezdeti értékeit is, de a két típus nem ekvivalens. (A típusműveletekben természetesen a régi típus helyére a származtatott típus kerül.)

(Megjegyzés. Ha például egy programnak fizikai jellemzőkkel kell számolnia, a legtöbb nyelvben a változókat így deklaráljuk :

```
D1,D2 : REAL;   -- távolságok
V1,V2 : REAL;   -- sebességek
```

Így semmilyen mód nincs a `D1 := V1;` utasítás nyilvánvaló hibájának felismerésére. A származtatás segítségével a különféle valós mennyiségekhez viszont új típusokat vezethetünk be:

```
type TAVOLSAG is new REAL;
type SEBESSEG is new REAL;

D1, D2 : TAVOLSAG;
V1, V2 : SEBESSEG;
```

Ekkor `D1 := V2;` szintaktikusan hibás utasítás, mert a baloldal és a jobboldal típusa nem egyezik meg. Emlékezzünk arra, hogy az Adában minden típus csak önmagával ekvivalens.)

A származtatás összeköthető az altípusképzéssel.

type NEW_TYPE **is new** OLD_TYPE <megszorítás>;

ekvivalens a következővel:

type <new type> **is new** <base type of OLD_TYPE>;
subtype NEW_TYPE **is** <new type><megszorítás>;

Az eredeti típus (őstípus) értékeit átkonvertálhatjuk a származtatott típusba és viszont. Ilyenkor a típus nevét használhatjuk konverziós függvényként.

```
type STATUS is (OFF,ON);
type BIT is new STATUS;
S : STATUS;
B : BIT;
```

Ekkor `B := BIT(S)` és `S := STATUS(B)` egyaránt helyes konverzió.

4.3. Diszkrét típusok

Az Ada diszkrét típusai a felsorolási és az egész típusok.

4.3.1. Felsorolási típusok

<enumeration type definition> ::= (*<enumaration literal specification>*
{ , *<enumaration literal specification>*})
<enumaration literal specification> ::= <enumeration literal>
<enumeration literal> ::= <identifier>|*<character literal>*

A felsorolási típus definíciójában a típusértékeket azonosítóikkal felsoroljuk, és az azonosítók sorrendje írja le a nagyságrendi viszonyaikat. Ezek az azonosítók a programban literálként használhatók.

```
type NAP is (HET,KED,SZE,CSU,PEN,SZO,VAS);
type VEGYES is ('a',VAS);  -- VAS átlapolt konstans
type STATUS is (OFF,ON);
type SWITCH is (OFF,ON);
type HEXA is ('A', 'B', 'C', 'D', 'E', 'F');
```

Egy felsorolási típus literáljainak nevei különbözők kell legyenek. Egy azonosítóval (vagy karakteres literállal) azonban több típusnak egy-egy értékét is jelölhetjük — ezt a nevek *átlapolás*ának nevezzük. Ha ekkor a program egy pontján nem egyértelmű, hogy melyik típus értékéről van szó, a literálra ún. minősített kifejezéssel hivatkozhatunk. Pl. `NAP'(VAS)`.

<qualified expression> ::= <type mark> ' (*<expression>*)
|*<type mark>* ' *<aggregate>*

Példák származtatott típusokra és altípusokra :

```
type BIT is new SWITCH;
type MUNKANAP is range HET .. PEN;
subtype RAINBOW is COLOR range RED .. BLUE;
```

Példák típuskonverzióra és értékadásra :

```
ST : STATUS;
SW : SWITCH;
B : BIT;
ST := STATUS(SW);-- illegális !
SW := SWITCH(B); -- helyes
B := BIT(SW);    -- helyes
```

A `CHARACTER` típus egy predefinit felsorolási típus, ami az ASCII karaktereket tartalmazza. A nem megjeleníthető karakterekre való hivatkozáshoz az Ada bevezet egy predefinit `ASCII` nevű package-t, s a nem megjeleníthető karakterekre `ASCII.LF`, `ASCII.CR`,... formában hivatkozhatunk.

A logikai típus is egy predefinit felsorolási típus, amelynek definíciója a következő :

```
type BOOLEAN is (FALSE,TRUE);
```

A logikai értékekre alkalmazhatók az **and**, **or**, **xor** és **not** műveletek. Az **and then** és

or else operátorok megfelelnek a logikai "és" és "vagy" műveleteknek, de a második operandus nem kerül kiértékelésre, ha az eredmény az első operandus alapján is megadható.

4.3.2. Az egész számok típusai

<integer type definition> ::= <range constraint>
<range constraint> ::= **range** *<range>*
<range> ::= <range attribute>|<simple expression> .. <simple expression>

Az Ada az `INTEGER`, `SHORT_INTEGER`, `LONG_INTEGER` predefinit egész típusokat tartalmazza. Az `INTEGER` típusból új típust pl. a

```
type INT is new INTEGER range -1000 .. 1000;
```

vagy rövidebben

```
type INT is range -1000 .. 1000;
```

formában származtathatunk (ez egyszerre származtatás és altípusképzés is). Az `INTEGER` típusból származtatott típusoknak abban van a jelentőségük, hogy míg az `INTEGER` típus implementációja gépenként változhat, az így származtatott és megszorított típusoké mindenütt egyforma. Ez nagyban elősegíti a program átvitelét más gépre.

A `NATURAL` és a `POSITIVE` az `INTEGER` típusnak egy predefinit altípusa. A definíciójuk a következő :

```
subtype NATURAL is INTEGER range 0 .. INTEGER'LAST;
subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST;
```

Az `INTEGER` típus értékeit egész számliterálokkal írjuk le, a műveletei a szokásos aritmetikai műveletek (+, -, *, /, **mod**, **rem**, **, **abs**). Az `A` **rem** `B` kifejezés előjele megegyezik az `A` előjelével, értéke pedig abszolút értékben kisebb, mint **abs**`(B)`. Az `A` **mod** `B` kifejezés előjele megegyezik a `B` előjelével, értéke pedig abszolút értékben kisebb, mint **abs**`(B)`.

4.3.3. A diszkrét típusok típusműveletei

A diszkrét típusú értékekre értelmezve vannak az alábbi relációk : <, <=, >, >=, **in**, **not in**.

A diszkrét típusosztálynál használható attributumok a következők: `PRED, SUCC, POS, VAL, FIRST, LAST, IMAGE, VALUE, WIDTH`. Ha `T` egy diszkrét típus és `V:T` egy változó, akkor `T'FIRST` a típus első (legkisebb) értéke, `T'LAST` a típus utolsó (legnagyobb) értéke, `T'POS(V)` a `V` érték sorszáma a típusértékek között, `IMAGE: T` $\longrightarrow$ `STRING` és `VALUE : STRING` $\longrightarrow$ `T` egy-egy konverziós függvény (pl. `INTEGER'IMAGE(20)="20"`, `COLOR'IMAGE(RED)="RED"`). `T'WIDTH` a `T'IMAGE(V)` értékek hosszainak maximuma. Érvényesek az alábbi egyenletek :

```
T'POS(T'SUCC(v)) = T'POS(V) + 1
T'POS(T'PRED(V)) = T'POS(V) - 1
T'VAL(T'POS(V)) = V
T'POS(T'VAL(k)) = k ( 0 ≤ k ≤ T'POS(T'LAST))
T'SUCC(T'PRED(V)) = V
T'VALUE(T'IMAGE(V)) = V
```

4.4. Valós típusok

Bár az Adát nem speciálisan a numerikus analízis feladatainak megoldására tervezték, mégis alkalmasabb a kényes numerikus problémák megoldására mint pl. a FORTRAN. Az itt definiált valós típus nagyban támogatja a hordozhatóságot.

A valós értékek hibájának pontosabb definiálásához az Ada új fogalmakat vezet be. Minden valós típushoz definiálja azoknak az értékeknek a halmazát, amit a nyelv minden implementációjának pontosan kell ábrázolnia. Ezeket az értékeket a típus *modellszám*ainak nevezzük. A modellszámok csak a típusdefiníciótól függenek, az implementációtól nem. Minden valós típushoz definiálja a *biztonságos szám*ok halmazát, ami a típusnak azokat az értékeit jelenti, amelyek az adott implementációban pontosan ábrázolhatók. A biztonságos számok halmaza lehet bővebb a modellszámok halmazánál.

A típusértékek egy olyan intervallumát, amelynek mindkét végpontja modellszám, *modell intervallum*nak nevezzük. Minden elméleti valós számhoz hozzárendelhetünk egy modell intervallumot: azt a legkisebb modell intervallumot, ami a számot tartalmazza. (A modellszámokhoz egy olyan intervallum tartozik, amelynek kezdő- és végpontja egyaránt maga a szám.) A műveletek és a relációk a modell intervallumokon vannak definiálva. Egy művelet eredménye mindig az a legkisebb modell intervallum, ami tartalmazza azt a legkisebb és legnagyobb matematikailag pontos eredményt, amit a operandus intervallumok értékeire megkaphatunk. Ha az eredményül kapott modell intervallum valamelyik határa abszolút értékben nagyobb, mint a legnagyobb biztonságos szám, az `NUMERIC_ERROR` hibát vált ki.

A valós típusú értékeket valós számliterálokkal adjuk meg, az értékekre használhatjuk a szokásos aritmetikai műveleteket (a hatványozás kitevője egész szám kell legyen), és a matematikai könyvtár függvényeit (pl. `SIN, COS, SQRT`). A valós típusokon értelmezve vannak a szokásos rendezési relációk is.

<real type definition> ::= *<floating point constraint>*
|*<fixed point constraint>*

Az Ada valós típusai két osztályba sorolhatók: fixpontos és lebegőpontos számok.

4.4.1. Lebegőpontos típusok

A lebegőpontos számok egy fix hosszúságú mantisszából, és egy ehhez tartozó előjeles egész exponensből állnak. Normalizált formának hívjuk azt az alakot, amikor a mantissza egészrésze nulla, s az első tizedesjegy nullától különböző (eltekintve a 0.0 szám esetétől).

<floating point constraint> ::= *<floating accuracy definition>*
[<range constraint>]
<floating accuracy definition> ::= **digits** *<static simple expression>*

Az Ada a `FLOAT` predefinit lebegőpontos típust vezeti be, aminek az ábrázolása implementációfüggő. Ezért a programokban általában nem ezt, hanem az ebből származtatott típusokat használjuk.

```
type REAL is new FLOAT;
```

Egy lebegőpontos valós típus pontosságát úgy definiálhatjuk, hogy megadjuk a mantissza ábrázolásához szükséges decimális jegyek számát. Jelölje ezt most D. Általában a gépeken az ábrázolás kettes számrendszerben történik, a modellszámok mantisszájának ábrázolásához B bitre van szükség, ahol $B = D \cdot log(10)/log(2) + 1$. A modellszámokra a kitevő értéktartományát az Ada pontosan definiálja: ez a $-4 \cdot B$ .. $4 \cdot B$ tartomány.

```
type REAL is new FLOAT digits 10;
```
vagy rövidebben
```
type REAL is digits 10;
```

Példák lebegőpontos típusok definiálására:

```
type MY_FLOAT is digits 8 range -1.0E20 .. 1.0E20; -- helyes
X : MY_FLOAT;
Z : MY_FLOAT range -1000.0 .. 1000.0;

type COEFFICIENT is digits 10 range -1.0 .. 1.0;
type REAL is digits 8;
type MASS is new REAL digits 7 range 0.0 .. 1.0E10;
```

Az egész-lebegőpontos ill. lebegőpontos-egész számkonverzióra a következő két függvényt használhatjuk:

`FLOAT(`*integer value*`)`
`INTEGER(`*float value*`)` `--` csonkítás a törtrész elhagyásával

A lebegőpontos típusok attributumai:

`T'DIGITS`	ugyanaz a **digits** érték, amit a típus definíciójában megadtunk
`T'MANTISSA`	a típus értékei mantisszájának ábrázolásához felhasznált bitek száma
`T'EPSILON`	az 1.0 és az azt követő első modellszám eltérése
`T'SMALL`	a típus legkisebb pozitív modellszáma

T´LARGE	a típus legnagyobb modellszáma
T´EMAX	a típus legnagyobb (normalizált) modellszámának kitevője

Az attributumokra érvényesek az alábbi egyenletek:

```
T´EMAX = 4 * T´MANTISSA   (definíció szerint)
T´EPSILON = 2.0 ** (1 - T´MANTISSA)
T´SMALL = 2.0 ** (-T´EMAX - 1)
T´LARGE = 2.0 ** T´EMAX * (1.0 - 2.0 ** (-T´MANTISSA))
```

4.4.2. Fixpontos típusok

<fixed point constraint> ::= *<fixed accuracy definition>*
[<range constraint>]
<fixed accuracy definition> ::= **delta** *<static simple expression>*

Az Adában nincs predefinit fixpontos típus, de a programozó definiálhat fixpontos típusokat. A fixpontos számokat fix számú számjeggyel, és egy képzeletbeli tizedesponttal ábrázoljuk. Ha a törtrész k db jegyből áll, akkor a 10^{-k} értéket az adott fixpontos számhoz tartozó *delta*-nak nevezzük.

```
type MY_FIXED is delta 0.01 range -100.0 .. 100.0;
```

Ez a definíció azt jelenti, hogy a számok két tizedesjegyet tartalmaznak, így a fenti típus ábrázolásához 1+7+7=15 bit szükséges (7 bit az egészrész, és 7 bit a törtrész ábrázolásához). Mivel a számítógépeken a számok ábrázolása kettes számrendszerben történik, ezért a típus értékei nem az $x \cdot 10^{-k}$, hanem az $x \cdot 2^{-p}$ számok, ahol p a legkisebb olyan pozitív egész, amire $2^{-p} \leq 10^{-k}$ (x alkalmas egész). A 2^{-p} számot a típushoz tartozó *small* értéknek nevezzük.

```
DEL : constant := 1.0 / 2 ** (WORD_LENGTH - 1);
type FRAC is delta DEL range -1.0 .. 1.0 - DEL;
-- FRAC így egy olyan típus, ami egy egész szót elfoglal
type VOLT is delta 0.125 range 0.0 .. 255.0;
subtype S_VOLT is VOLT delta 0.5;
```

A fixpontos-integer konverzióra a fixpontos típus neve használható konverziós függvényként.

```
A1 : AFIX; A2 : AFIX; C : CFIX; I : INTEGER;
C := CFIX(A1 * A2);
I := INTEGER(A1 * A2);
```

Mivel a *delta* értékét 10-es számrendszerben adjuk meg, *small* értéke pedig kettő hatványa, ezért a fixpontos típusok értékei pontatlanok is lehetnek.

```
type CENTS is delta 0.01 range 0.0 .. 1.0;
```

A `0.01` pontosság helyett *small* értéke 1/128 (= 0.0078...). Ezért, ha pl. a `0.05` értéket a `CENTS` típusba konvertáljuk, akkor ott a `0.046...` értéket kap-

juk. Azért, hogy az ilyen hibákat elkerüljük, célszerű reprezentációs előírással definiálni a típushoz a *small* értéket (ld. 13.):

```
type CENTS is delta 0.01 range 0.0 .. 1.0;
for CENTS'SMALL use 1.0 / 2 ** (WORD_LENGTH - 1);
```

A fixpontos típusok attributumai:

T'DELTA	ugyanaz a **delta** érték, amit a típus definíciójában megadtunk
T'MANTISSA	a típus értékei mantisszájának ábrázolásához felhasznált bitek száma
T'SMALL	a típus legkisebb pontosan ábrázolt pozitív értéke
T'LARGE	a típus legnagyobb pontosan ábrázolt értéke

Az attributumokra érvényes az alábbi egyenlet:

```
T'LARGE = (2 ** T'MANTISSA - 1) * T'SMALL
```

4.5. Tömb típusok

A tömb egy olyan összetett típus, ami két komponens típusból, az indextípusból és az elemtípusból áll. Minden típusérték az elemtípusból vett értékek sorozata, amiket az indextípusból vett értékekkel folyamatosan sorszámoztunk. Az elemtípus bármilyen típus, az indextípus bármilyen diszkrét típus lehet. A tömbnek több indexe is lehet.

<array type definition> ::= <unconstrained array definition>
|<constrained array definition>
<unconstrained array definition> **::= array** (*<index subtype definition>*
{ **,** *<index subtype definition>*}) **of** *<component subtype indication>*
<constrained array definition> **::= array** *<index constraint>* **of**
<component subtype indication>
<index subtype definition> **::=** *<type mark>* **range <>**
<index constraint> **::=** (*<discrete range>* { **,** *<discrete range>*})
<discrete range> **::=** *<discrete subtype indication>*|*<range>*

A tömb típus definiálásának két formája van : a határozott és a határozatlan indexhatárral való definiálás.

```
type TABLAZAT is array (INTEGER range 1 .. 6) of INTEGER;
type VECTOR is array (INTEGER range <>) of INTEGER;
type MATRIX is
  array (INTEGER range <>, INTEGER range <>) of REAL;
type SCHEDULE is array (NAP) of BOOLEAN;
A : MATRIX(1 .. 2, 1 .. 5);
MIX : array (COLOR range RED .. GREEN) of BOOLEAN;
INVERSE : MATRIX (1 .. N, 1 .. N); -- N lehet változó is
Z : constant MATRIX := ((0.0,0.0),(0.0,0.0));
```

Bár egy tömb típusban az indexhatárok lehetnek határozatlanok, egy objek-

tum (változó vagy konstans) indexhatárai már mindig határozottak kell legyenek. Egy határozatlan indexhatárú tömb típusból indexmegszorítás megadásával képezhetünk olyan altípust, amely már alkalmas objektumok definiálására. Szerepelhet viszont meghatározatlan méretű tömb alprogram formális paramétereként (ld. a következő példát), ilyenkor az alprogram meghívásakor az aktuális paraméter definiálja az indexhatárokat.

```
procedure ADD_TO(A:in VECTOR; B:in out VECTOR) is
begin -- Feltételezzük, hogy A'RANGE = B'RANGE.
    for I in A'RANGE loop
        B(I) := B(I) + A(I);
    end loop;
end ADD_TO;
```

<indexed component> ::= <prefix> (*<expression>* { , *<expression>*})
<slice> ::= <prefix> (*<discrete range>*)

A tömbök elemeire a szokásos módon, indexeléssel hivatkozhatunk. Pl. `MIX(RED)`, `M(I,J)`, vagy vektorokból képzett vektor esetén `X(I)(J)`. Ha érvényes a

```
type FRAME is access MATRIX;
```

típusdefiníció (ld. 4.7.) és a

```
function NEXT_FRAME(K:POSITIVE) return FRAME;
```

függvénydefiníció (ld. 6.1.), akkor `NEXT_FRAME(L)(I,J)` is szabályos indexelés. Tehát nemcsak a változók, hanem a tömbértéket visszaadó függvényhivatkozások is indexelhetők.

Az Adában nemcsak a vektor egészére vagy egy elemére, hanem pl. `V1(TOL..IG)` formában a vektor egy részére is lehet hivatkozni. (Megjegyezzük, hogy `V1(1..1)` és `V1(1)` nem egyenértékű!)

```
V1,V2 : VECTOR(0 .. 9);
V3 : VECTOR(1 .. 10);
V4 : VECTOR(0 .. 10);

V1 := V2;
V3 := V1;
V3(1) := V4(0);
V1(0 .. 4) := V2(5 .. 9);
V4(0 .. 9) := V1;
V4 := V1;                  -- szintaktikusan hibás
V4(0) := V1(0 .. 4)(K);-- helyes, ha 0 ≤ K ≤ 4
```

A tömbökre alkalmazható az `=` és a `/=` reláció, továbbá tetszőleges diszkrét értékű vektorokra a `<`, `<=`, `>`, `>=` (lexikografikusan értelmezett) relációk, azonos hosszúságú logikai vektorokra az **and**, **or**, **xor**, **not** műveletek (amelyek eredménye olyan logikai vektor, amelynek indexhatárait a baloldali argumentum indexhatárai adják), és általában a vektorokra a konkatenáció (`&`). A konkatenáció

egyik operandusa lehet skalár is.

A tömbökkel kapcsolatban használható attributumok a következők (itt `X` egy k-dimenziós tömb, n egy statikus kifejezés) :

`X´FIRST(`n`)`	az n. index kezdőértéke
`X´LAST(`n`)`	az n. index végértéke
`X´LENGTH(`n`)`	`= X´LAST(`n`) - X´FIRST(`n`) + 1`
`X´RANGE(`n`)`	`= X´FIRST(`n`) .. X´LAST(`n`)`

($n = 1$ esetben n a zárójelekkel együtt elhagyható.) A tömb lehet üres is, ekkor definíció szerint `X´LAST = X´FIRST - 1`.

<aggregate> ::= `(` *<component association>* { `,` *<component association>*} `)`
<component association> ::= [*<choice>* { | *<choice>*} `=>`] *<expression>*

A tömb és rekord típusú összetett értékeket az Ada *aggregátum*nak nevezi. Az aggregátumokat kétféleképpen lehet megadni. Pozíciós formában,

```
A : TABLAZAT := (2, 4, 4, 4, 0, 0);
A : TABLAZAT := (2, 4, 4, 4, others => 0);
```

vagy név szerinti formában (ebben az esetben nem kötelező az értékeket az indexek növekvő sorrendjében megadni):

```
A := (1 => 2, 2 | 3 | 4 => 4, others => 0);
M := (1 => (1.1,1.2,1.3), 2 => (2.1,2.2,2.3));
M := (1 .. 2 => (1 .. 2 => 0.0)) :
B : TABLAZAT := TABLAZAT´(2 | 4 | 6 => 1, others => 0);
C : constant MATRIX := (1 .. 5 => (1 .. 8 => 0.0));
X : BIT_VECTOR(M .. N) := (X´RANGE => TRUE);
```

Az **others** nem használható akkor, ha az aggregátum indexhatárait a környezete nem határozza meg egyértelműen.

```
if X < (others => TRUE) then ...  -- hibás!
```

A `STRING` egy predefinit vektor típus a következő definícióval:

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

Helyesek a következő típusdefiníciók és deklarációk:

```
type ROMAN_DIGIT is ('I','V','X','L','C','D','M');
type ROMAN_NUMBER is array(POSITIVE range <>) of ROMAN_DIGIT;
KILENCVENHAT : constant ROMAN_NUMBER := "XCVI";
S1 : constant STRING := "GIN";
S2 : constant STRING := ('G','I','N');
S3 : constant STRING := 'G' & FUNC(0);
NULL_STRING : constant STRING(1..0);
```

Végül készítünk egy eljárást a `STRING` típus használatának bemutatására:

```
procedure REVERSE_STRING(S:in out STRING) is
    L: constant NATURAL := S'FIRST;
    U: constant NATURAL := S'LAST;
    C: CHARACTER;
begin
    for I in L .. (L + U) / 2 loop
        C := S(I); S(I) := S(U - I + 1); S(U - I + 1) := C;
    end loop;
end REVERSE_STRING;
```

4.6. A rekord típus

<record type definition> ::= **record** *<component list>* **end record**
<component list> ::= *<component declaration>* {*<component declaration>*}
| {*<component declaration>*} *<variant part>*| **null** ;
<component declaration> ::= *<identifier list>* :
<component subtype definition> [:= *<expression>*] ;
<component subtype definition> ::= *<subtype indication>*

A rekord egy összetett típus, amelynek komponens típusai különbözők lehetnek. A rekordértékekben a komponensekre névvel, ún. *szelektor*ral hivatkozhatunk.

```
type HONAP is (JAN,FEB,MAR,MAJ,JUN,JUL,AUG,SZE,OKT,NOV,DEC);
type DATUM is
  record
    NAP : INTEGER range 1 .. 31;
    HO : HONAP;
    EV : INTEGER;
  end record;
type COMPLEX is
  record
    RE, IM : FLOAT := 0.0;
  end record;
D : DATUM := (4,JUL,1776);
C1 : COMPLEX;
C2 : COMPLEX := (1.0,2.0);
```

A rekord mezőire a szokásos szelekciós jelöléssel hivatkozhatunk, tehát pl. `D.EV`.

<selected component> ::= *<prefix>* . *<selector>*
<selector> ::= *<simple name>*|*<character literal>*|*<operator symbol>*| **all**

A rekord az egyetlen típusfajta, ahol a típus definíciójában megadhatók a típus objektumainak kezdőértékei. Csak akkor nem használhatunk kezdőértéket, ha a rekord **limited** (ld. 7.4.).

```
type NAME is
  record
    VAL : STRING(1 .. MAX_SIZE);
```

```
    SIZE: INTEGER range 0 .. MAX_SIZE := 0;
end record;
```

Az `N1:NAME;` deklaráció pillanatában az `N1.SIZE` komponens értéke `0` lesz. (A kezdeti érték természetesen felülírható.)

A rekord aggregátumokat — a tömb aggregátumokhoz hasonlóan — kétféleképpen is írhatjuk: vagy felsoroljuk a mezők sorrendjében azok értékeit (ez az ún. pozíciós forma), vagy felírjuk a szelektorokat, és mindegyik mellé odaírjuk a mező értékét (ún. név szerinti forma). Az utóbbi esetben a mezők felsorolásának sorrendje közömbös.

```
D2 : DATUM := (4, JUL, 1776);
D1 : DATUM := (HO => JUL, NAP => 4 , EV => 1776 );
```

Ha egy aggregátum típusa nem egyértelmű, akkor minősítést használhatunk: `DATUM´(4,JUL,1776)`.

4.6.1. A rekord diszkriminánsai

<discriminant part> ::= `(` *<discriminant specification>*
`{ ;` *<discriminant specification>*`}` `)`
<discriminant specification> ::= *<identifier list>* `:`
<type mark> `[ :=` *<expression>* `]`
<discriminant constraint> ::= `(` *<discriminant association>*
`{ ,` *<discriminant association>*`}` `)`
<discriminant association> ::= `[` *<discriminant simple name>*
`{ |` *<discriminant simple name>*`}` `=>` `]` *<expression>*

Az Ada, ugyanúgy, ahogy a tömb típus esetében lehetőséget ad paraméteres típus definiálására, megengedi a rekord típusok paraméterezését is egy vagy több paraméterrel. A rekord típus paraméterét a rekord *diszkriminánsának* hívjuk. A rekord diszkriminánsa csak diszkrét érték lehet.

Hasonlóan a határozatlan indexhatárú tömbhöz, a paraméteres rekord típus sem alkalmas közvetlenül objektum (változó vagy konstans) definiálására, szerepeltethető viszont alprogram formális paramétereként (ilyenkor az alprogram meghívásakor az aktuális objektum definiálja a diszkrimináns értékét). A rekordok esetében azonban mégis picit más a helyzet, mint a tömböknél láttuk, ui. a rekord diszkriminánsának lehet default értéke (ilyenkor a rekord típus mindegyik diszkriminánsának kell hogy legyen default értéke). Abban az esetben, ha a rekord diszkriminánsai default értékkel rendelkeznek, a típus megszorítás megadása nélkül is használható objektum-definiálásra, mert ilyenkor a diszkrimináns egyszerűen felveszi a default értékét.

Fontos szabály, hogy ha a típus felhasználásakor explicit módon megadunk a diszkriminánshoz egy megszorítást, ezzel a típus egy altípusát állítjuk elő. Ha viszont a diszkrimináns default értékét használjuk fel, azzal nem szűkítjük le a

típust.

A diszkriminánssal ellátott rekord alkalmazását mutatja be a következő két példa :

```
type TEXT (MAX : POSITIVE) is
  record
    LENGTH : NATURAL := 0;
    VALUE : STRING(1 .. MAX);
  end record;
```

A TEXT típus második komponensének értéke olyan string lehet, aminek maximális hossza MAX. A LENGTH komponenst arra használhatjuk, hogy jelezzük a VALUE vektorban elhelyezett karaktersorozat valódi hosszát.

```
NAP : TEXT(9);
NAP := (9,6,"MONDAY ");
NAP : TEXT; -- csak akkor lenne írható, ha default érték van
```

Diszkriminánssal rendelkező rekord típus esetében, ha az objektum definiálásakor nem adunk meg megszorítást (tehát a diszkrimináns default értékét használjuk fel), akkor a fordítóprogram a legnagyobb változathoz tartozó memóriát foglalja le.

```
subtype LINE_RANGE is INTEGER range 1 .. 100;
type LINE (SIZE: LINE_RANGE := 10) is
  record
    VAL : array (1 .. SIZE) of CHARACTER;
  end record;
```

Ha megszorítás nélkül deklarálunk változót a LINE típus felhasználásával (pl. L : LINE;), akkor a fordító 100 karakternek foglal le helyet. Ha megadjuk a megszorítást, pl. L : LINE(16);, akkor csak az altípus számára foglal le helyet, tehát itt most 16 karaktert.

4.6.2. Variáns rekord

<variant part> ::= **case** *<discriminant simple name>* **is**
    <variant> {*<variant>*}
    end case ;
<variant> ::= **when** *<choice>* { | *<choice>*} => *<component list>*
<choice> ::= *<simple expression>*|*<discrete range>*| **others**
    |*<component simple name>*

Az diszkriminánsfelhasználásával alakítható ki az ún. variáns részt tartalmazó rekord, ami unió típus definiálására használható.

```
type DEVICE is (PRINTER, DISK, DRUM);
type STATE is (OPEN, CLOSED);
type PERIPHERAL(UNIT : DEVICE := PRINTER) is
```

```
  record
    STATUS : STATE;
    case UNIT is
      when PRINTER =>
        LINE_COUNT : INTEGER range 1 .. PAGE_SIZE;
      when others =>
        CYLINDER : CYLINDER_INDEX;
        TRACK : TRACK_NUMBER;
    end case;
  end record;
subtype MMDDYY is STRING(1 .. 6);
subtype NAME_TYPE is STRING(1 .. 25);
type STATUS_TYPE is (DIVORCED, MARRIED, SINGLE, WIDOWED);
type SEX_TYPE is (FEMALE, MALE);
type PERSON (MARITAL_STATUS: STATUS_TYPE := SINGLE) is
  record
    NAME : NAME_TYPE;
    SEX : SEX_TYPE;
    DATE_OF_BIRTH : MMDDYY;
    NUMBER_OF_DEFENDENTS : INTEGER;
    case MARITAL_STATUS is
      when MARRIED => SPOUSE_NAME : NAME_TYPE;
      when DIVORCED => DATE_OF_DIVORCE;
      when SINGLE => null;
      when WIDOWED => DATE_OF_DEATH : MMDDYY;
    end case;
  end record;
```

Az esetmegjelöléseket (**when**) statikus kifejezésekkel kell megadni. Ha intervallumot használunk (pl. `A .. B`), ez nem lehet üres. Minden szóbajöhető értéket fel kell sorolni az esetmegjelölések között (vagy pedig az **others** lehetőséget kell használni.) Ha egy `PERIPHERAL` típusú `X` objektumban `UNIT = PRINTER`, akkor `X.TRACK := 1;` futás közben hibajelzést (exception-t) vált ki.

A variáns részt tartalmazó, diszkriminánssal ellátott rekord típus olyan rekordértékek halmaza, amelyek felépítése különböző. Ugyanúgy, ahogyan a diszkrét típusoknál az értékkészlet megszorítását használjuk altípusképzésre, a variáns rész egy variánsának megkötése szolgál a rekord típus megszorítására. Pl. a `PERIPHERAL` típusból altípust is definiálhatunk:

```
subtype LP is PERIPHERAL(PRINTER);
L : LP := (PRINTER,OPEN,72);
```

Altípusképzéskor az összes diszkriminánst meg kell adni (vagy a pozíciós, vagy a névvel jelölt formában).

Tekintsük a következő példát:

```
X : PERIPHERAL;
C : PERIPHERAL(PRINTER);
```

`X` most egy olyan rekord objektum, amely a `UNIT` diszkrimináns aktuális értékétől

függően más-más szerkezetű értékeket viselhet (ez futás közben megváltoztatható). C -ben paraméterezéssel rögzítettük a UNIT mező értékét, és ez már nem változtatható meg. C csak a PRINTER variánsnak megfelelő értéket vehet fel.

Az Ada a diszkriminánsokkal szemben megköveteli még a következőket:
- típusdefinícióban a diszkrimináns csak indexmegszorításként, a variáns rész neveként, vagy aktuális paraméterként használható. Minden esetben önmagában kell állnia, nem szerepelhet kifejezésben.
- A variáns rész neve nem kezelhető úgy, mint a rekord egy komponense. Csak úgy kaphat új értéket, ha az egész rekord új értéket kap.
- Ha a diszkriminánssal rendelkező típus újabb diszkriminánssal rendelkező típust tartalmaz, akkor ennek a diszkriminánsa csak a külső típus diszkriminánsaitól függhet.

```
type SHAPE is (RECT, CIRCLE, POINT);
type POSITION is record X, Y : FLOAT; end record;
type GEOMETRIC_FIGURE(S:SHAPE) is
  record
    PERIMETER : FLOAT;
    case S is
      when RECT => L, B : NATURAL;
      when CIRCLE => RADIUS : FLOAT := 1.0;
      when POINT => P : POSITION;
    end case;
  end record;
function AREA(F : GEOMETRIC_FIGURE) return FLOAT is
  PI : constant := 3.1614;
begin
  case F.S is
    when RECT => return FLOAT(F.L * F.B);
    when CIRCLE => return PI * F.RADIUS ** 2;
    when POINT => return 0.0;
  end case;
end AREA;
```

4.7. Pointer típusok

Valamely programegység vagy blokk deklarációs részében definiált objektum az illető egység memóriarészében kap helyet. Az objektum létezésének ideje alatt ennek az objektumnak a mérete és a címe állandó, az objektumra a nevével hivatkozhatunk.

Azokat a típusokat, amelyek objektumainak mérete vagy a címe a létrehozása után is változhat, *dinamikus típus*oknak mondjuk. A dinamikus objektumokat a programozási nyelvekben, s így az Adában is, pointerekkel építjük fel. Az Ada a "pointer", "pointer típus" elnevezések helyett az *access típus* elnevezést használja. Azt a típust, amilyen típusú objektumokra a pointerek mutatnak, a dinamikus típus *gyűjtőtípus*ának nevezzük.

<access type definition> ::= **access** *<subtype indication>*
<incomplete type definition> ::= **type** *<identifier>* [*<discriminant part>*] ;
<allocator> ::= **new** *<subtype indication>*| **new** *<qualified expression>*

Egy **acces** típus definiálásakor előfordulhat, hogy a definíció "rekurzív" (pl. ez előfordul már egy egyszerű lista definíciójakor is). Mivel az Adában az a szabály, hogy minden azonosítónak, amire hivatkozunk, már deklaráltnak kell lennie, ezért ilyen esetben a következőképpen járhatunk el :

```
type CELL;                    -- A CELL azonosítót típusnévnek deklaráljuk,
                              -- ez lesz majd a pointer típus gyűjtőtípusa.
type LINK is access CELL;     -- Ezután definiáljuk a LINK típust,
type CELL is                  -- végül megadjuk a gyűjtőtípus teljes definícióját.
  record
    VALUE : INTEGER;
    PRED,SUCC : LINK;
  end record;
```

Példák a fenti típus használatára:

```
HEAD : LINK := null;          -- default null érték
NEXT : LINK;
HEAD := NEXT;                 -- másolja a pointert
HEAD.PRED := NEXT;            -- szabályos
HEAD.all := NEXT.all;         -- A HEAD pointer által mutatott
          -- teljes objektumra HEAD.all formában hivatkozhatunk.
C:constant LINK := new CELL'(0,null,null);
```

Minden dinamikus objektum létrehozása allokátorral történik. Ha nem áll rendelkezésre elegendő memória, a **new** végrehajtása STORAGE_ERROR hibát vált ki.

```
HEAD : new CELL'(0,null,null);
NEXT := new CELL'(0,null,HEAD);
L := new CELL;
N := new CELL'(10,L); -- az új az előzőre mutat
```

Az **acces** típus predefinit konstansa a **null**. Az a pointer, amelynek az értéke **null**, nem mutat objektumra. Definiálhatók konstans pointerek is, amelyek valamely objektumra mutatnak. Megjegyezzük, hogy ebben az esetben a pointer a konstans, és nem az objektum.

```
CP1 : constant ITEM_PTR := new ITEM'(" ",null);
CP2 : constant ITEM_PTR := CP1;
CP1 := null;                  -- hibás, mert CP1 konstans
CP1.NAME := "+++";            -- helyes (CP2.NAME is értéket kap!)
```

Az **access** típus gyűjtőtípusa lehet diszkriminánssal ellátott rekord, és határozatlan méretű tömb is. Ilyen esetben legkésőbb egy objektum allokálásakor meg kell adni a megszorítást. Vagyis, miután dinamikusan allokáltunk egy paraméteres rekordot (pl. variáns részt, vagy határozatlan hosszúságú tömböt tartalmazó rekordot), futás közben a méretét már nem változtathatjuk meg.

```
type P is access PERIPHERAL;
V1 : P := new PERIPHERAL´( UNIT => DISK);
V2 : P := new PERIPHERAL´( UNIT => PRINTER);
V3 : P(UNIT => DISK);
type ACC_STRING is access STRING;
type VSTR(FIRST, LAST : INTEGER) is
  record
    DATA : ACC_STRING(FIRST .. LAST);
  end record;
X : VSTR(1 .. 3);
X := (1, 3, new STRING(1 .. 3));
```

`X : P := new PERIPHERAL;` (ld. 4.6.) hibás, ha `PERIPHERAL` -ban nincs default érték `UNIT`-ra. A második sorban a megszorítás csak a kezdeti értékre vonatkozik, a negyedikben viszont `V3` típusára. Tehát (a fenti értékek esetén) pl. `V1 := V2;` legális, sőt, `V3 := V1;` is az, de `V3 := V2;` hibás, mert `V3` nem vehet fel `PRINTER` szerkezetű értéket.

A dinamikus objektumok legfeljebb addig maradnak fenn, amíg a *típus definícióját* tartalmazó programegység vagy blokk aktív. A típus hatáskörének elhagyásakor az Ada futtató rendszere a felszabadult területet újrafelhasználhatóvá teszi. (Ha az access típust túlságosan magas szinten, pl. egy fordítási egység specifikációs részében definiáljuk, akkor egy belső blokkban lokális adatok életben maradhatnak a blokkjuk megszűnte után is, bár ezek már soha nem érhetők el.) Az Adában a dinamikus memória méretét konstansként rögzítik. Mivel az Ada biztonságosan és hatékonyan kívánja kezelni a dinamikus adatokat, ezért nem tartalmaz explicit felszabadító utasítást, hanem a felszabadítást a hatáskörrel hozza kapcsolatba.

```
type NODE;
type BRANCH is access NODE;
type NODE is
  record
    VALUE : INTEGER;
    LEFT, RIGHT : BRANCH;
  end record;
function SEARCH(R:in BRANCH; X:in INTEGER) return BOOLEAN is
begin
    if R = null then return FALSE;
    elsif X = R.VALUE then return TRUE;
    elsif X < R.VALUE then return SEARCH(R.LEFT, X);
    else return SEARCH(R.RIGHT, X);
    end if;
end SEARCH;
```

(Megjegyzés. A dinamikus típusok kérdésében nem sikerül az Adának meghaladnia a többi korabeli nyelv szintjét. A pointerekkel az a baj, hogy túlságosan is könnyen, egészen apró programhibákkal már inkonzisztens adatszerkezeteket kaphatunk. Ez a fogalom alacsonyabb szintű, mint az Ada többi típusa. Ugyanakkor, ha bevezették volna mondjuk a *lista* és a *gráf* összetett típusokat, a nyelv biztonságosságát tovább növelhették volna. Olyan ez, mintha nem lenne a nyelvben while ciklus, hanem **if** és **goto** utasításokkal kellene azt megszervezni.)

5. Utasítások

<sequence of statements> ::= *<statement>* {*<statement>*}
<statement> ::= {*<label>*} *<simple statement>*
| {*<label>*} *<compound statement>*

Az Ada programegységek törzse egy deklarációs részt és egy utasítássorozatot tartalmaz. Ezek az utasítások a leírásuk sorrendjében hajtódnak végre, s ezt a sorrendet csak egy **goto** utasítás vagy egy hibaesemény fellépése változtathatja meg.

<simple statement> ::= *<null statement>*|*<assignment statement>*
|*<procedure call statement>*|*<exit statement>*|*<return statement>*
|*<goto statement>*|*<entry call statement>*|*<delay statement>*
|*<abort statement>*|*<raise statement>*|*<code statement>*
<compound statement> ::= *<if statement>*|*<case statement>*
|*<loop statement>*|*<block statement>*|*<accept statement>*
|*<select statement>*
<label> ::= << *<label simple name>* >>
<null statement> ::= **null** ;

5.1. Az értékadás

<expression> ::= *<relation>* { **and** *<relation>*}
|*<relation>* { **and then** *<relation>*}
|*<relation>* { **or** *<relation>*}
|*<relation>* { **or else** *<relation>*}
|*<relation>* { **xor** *<relation>*}
<relation> ::= *<simple expression>*
[*<relational operator>**<simple expression>*]
|*<simple expression>* [**not**] **in** *<range>*
|*<simple expression>* [**not**] **in** *<type mark>*
<simple expression> ::= [*<unary adding operator>*] *<term>*
{*<binary adding operator>**<term>*}
<term> ::= *<factor>* {*<multiplying operator>**<factor>*}
<factor> ::= *<primary>* [** *<primary>*] | **abs** *<primary>*| **not** *<primary>*
<primary> ::= *<numeric literal>*| **null** |*<aggregate>*|*<string literal>*
|*<name>*|*<allocator>*|*<function call>*|*<type conversion>*
|*<qualified expression>*| (*<expression>*)
<logical operator> ::= **and** | **or** | **xor**
<relational operator> ::= = | /= | < | <= | > | >=
<binary adding operator> ::= + | – | &
<unary adding operator> ::= + | –
<multiplying operator> ::= * | / | **mod** | **rem**
<highest precedence operator> ::= ** | **abs** | **not**
<type conversion> ::= *<type mark>* (*<expression>*)

Az értékadás arra szolgál, hogy egy kifejezés értékét egy változóhoz rendeljük.

<assignment statement> ::= *<variable name>* := *<expression>* ;

Az értékadásban az érték és a változó típusának meg kell egyeznie. A kifejezés értéke nem felel meg a változó altípusának, az CONSTRAINT_ERROR hibát vát ki. Az értékadás a **limited private** típusú (ld. 7.4.) változóktól eltekintve minden változóra alkalmazható.

```
I,J : INTEGER range 1 .. 10;
K : INTEGER range 1 .. 20;
I := J;        -- azonos tartományúak
K := J;        -- összeegyeztethető tartományúak
J := K;        -- ha K > 10, akkor előidézi a CONSTRAINT_ERROR-t
```

A kifejezés egy formula, ami az érték kiszámítását definiálja. Az Ada kifejezés megfelel a szokásos kifejezésformáknak.

Az értékadás tömbökre is alkalmazható, ilyenkor a két tömb elemtípusának és az elemszámnak meg kell egyeznie, az indexhatárok eltérhetnek.

```
A : STRING(1 .. 31);
B : STRING(3 .. 33);
A := B;                      -- azonos elemszámúak
A(1 .. 9) := "tar sauce"
A(4 .. 12) := A(1 .. 9); -- A(1 .. 12) = "tartar sauce"
                         -- és nem A(1 .. 12) = "tartartartar" !
```

5.2. Az if utasítás

<if statement> ::= **if** *<condition>* **then** *<sequence of statements>*
{ **elsif** *<condition>* **then** *<sequence of statements>*}
[**else** *<sequence of statements>*]
end if ;
<condition> ::= *<boolean expression>*

Az **if** utasítás szemantikáját az Ada a szokásos módon (pl. mint a Modula-2) határozza meg.

```
if MONTH = DECEMBER and DAY = 31 then
   MONTH := JANUARY;
   DAY := 1;
   YEAR := YEAR + 1;
end if;
```

```
if INDENT then
   CHECK_LEFT_MARGIN; LEFT_SHIFT;
elsif OUTDENT then
   RIGHT_SHIFT;
else
   CARRIAGE_RETURN; CONTINUE_SCAN;
end if;
```

```
if MY_CAR.OWNER.VEHICLE /= MY_CAR then
  FAIL("Incorrect record");
end if;
```

5.3. A case utasítás

<case statement> ::= **case** *<expression>* **is**
    <case statement alternative> {*<case statement alternative>*}
    end case ;
<case statement alternative> ::= **when** *<choice>* { | *<choice>*} =>
    <sequence of statements>

A **case** utasításban vagy használni kell az **others** ágat, vagy pedig a kifejezés minden lehetséges értékét fel kell sorolni a **when** ágak között, és mindegyiket pontosan egyszer.

```
subtype UP_TO_20 is INTEGER range 1 .. 20;
VAL : UP_TO_20;
case VAL is
  when 1 .. 3 | 5 | 7 | 11 | 13 | 17 | 19 => DO_PRIME;
  when others => null;   -- Ha az others szót használjuk, az
end case;                -- utolsó ágban kell lennie.

case BIN_NUMBER(COUNT) is
  when 1 => UPDATE_BIN(1);
  when 2 => UPDATE_BIN(2);
  when 3 | 4 => EMPTY_BIN(1); EMPTY_BIN(2);
  when others => raise ERROR;
end case;

case TODAY is
  when MON => INITIAL_BALANCE;
  when FRI => CLOSING_BALANCE;
  when TUE .. THU => REPORT(TODAY);
  when SAT | SUN => null;
end case;
```

5.4. Ciklus utasítások

<loop statement> ::= [*<loop simple name>* :] [*<iteration scheme>*] **loop**
    <sequence of statements>
    end loop [*<loop simple name>*] ;
<iteration scheme> ::= **while** *<condition>*
    | **for** *<loop parameter specification>*
<loop parameter specification> ::= *<identifier>* **in** [**reverse**]
    <discrete range>

A ciklusnak három formája használatos. Az egyik a tradicionális iteráció ciklusváltozó segítségével (a **for** ciklus üres tartomány esetén egyszer sem hajtja végre a ciklusmagot):

```
for I in 1 .. MAX_NUM_ITEMS loop
  GET_NEW_ITEM(NEW_ITEM);
  MERGE_ITEM(NEW_ITEM.STORAGE_FILE);
  exit when NEW_ITEM = TERMINAL.ITEM;
end loop
```

```
for J in reverse BUFFER'RANGE loop
  if BUFFER(J) /= SPACE then
    PUT(BUFFER(J));
  end if;
end loop;
```

A ciklusváltozó a ciklus lokális konstansa, amit a ciklus első sorában deklarálunk.

A második forma a while típusú ciklus :

```
SUM := 0;
while NEXT /= null loop;
  SUM := SUM + NEXT.VALUE;
  NEXT := NEXT.SUCC;
end loop;
```

```
while BID(N).PRICE < CUT_OFF.PRICE loop
  RECORD_BID(BID(N).PRICE);
  N := N + 1;
end loop;
```

A harmadik forma egy feltétel nélküli ciklus, aminek a megszakítását az **exit** utasítással érhetjük el (ld. 5.6.).

```
loop
  --
  exit when FELTETEL;
  --
end loop;
```

A ciklusnak nevet is adhatunk. Ennek formája az, hogy a ciklusfej előtt, attól kettősponttal elválasztva leírunk egy azonosítót. A ciklusnévnek az **exit** utasításban, és egymásbaágyazott ciklusok esetén van szerepe.

```
A1 : for I in ... loop
     A2 : for I in ... loop
          ... B(A1.I) ...
     end loop A2;
end loop A1;
```

(Feladat. Az Ada nem tartalmaz a Modula-2 **REPEAT** utasításához hasonló, ún. posttest iteration ciklust. Tervezzünk egyet, ami jól illik az Ada többi ciklusához!)

5.5. A blokk utasítás

Blokkot képezni az Ada programban nemcsak alprogram kialakításával, hanem **declare** utasítással is lehet.

<block statement> ::= [<block simple name> :]
[<declare declarative part>]
begin *<sequence of statements>*
[**exception** *<exception handler {<exception handler>}*
end *[<block simple name>]* ;

A **declare** a következő helyzetekben használható előnyösen:

- Ha egy nagy adatstruktúrára van szükség a program egy rövid szakaszán. Ebben az esetben a struktúra deklarálható egy blokkban, így a memóriát a program csak arra az időre terheli meg, amíg a struktúra aktív.

```
declare
    TEMP : T;
begin
    TEMP := A; A := B; B := TEMP;
end;
```

- Az Ada megengedi olyan objektumok definiálását, amelyek méretét futási időben számítjuk ki. Van úgy, hogy a méretet nem lehet egy kifejezéssel megadni, egy kevés számítást is el kell végezni a változó deklarációja előtt. Egy lehetséges megoldás ilyenkor a **declare** utasítás használata.

```
A:VECTOR(N);
...
M := A'FIRST;
while M /= A'LAST and then A(M) /= 0 loop
  M := M + 1;
end loop;
declare
  B:VECTOR(M);
begin
  ...
end;
```

- Már egy deklarációs rész nélküli blokktörzs is lehetővé teszi, hogy a blokkban hibakezelő programrészt (exception handler, ld. 8.) helyezzünk el.

```
begin
  A := C / B;
exception
  when OVERFLOW => A := INTEGER'LAST;
  when UNDERFLOW => A := INTEGER'FIRST;
end;
```

A blokkban használhatjuk annak lokális azonosítóit, és a blokkot magába foglaló egységben látható azonosítókat, mint globális neveket. Ha névazonosság fordul elő, szelekciós jelöléssel (pl. `X.A`) hivatkozhatunk a kívánt objektumra.

```
KULSO: declare
  I,J:INTEGER;
begin
```

```
                -- I,J:INTEGER; közvetlenül elérhető
    BELSO: declare
      I:MY_INT;
    begin
                -- I:MY_INT; és J:INTEGER; közvetlenül elérhető, de
                -- I:INTEGER; csak KULSO.I alakban érhető el
    end BELSO;
      --
end KULSO;
```

5.6. Az exit utasítás

<exit statement> ::= **exit** [*<loop name>*] [**when** *<condition>*] ;

Az **exit** utasítás elhelyezhető bármely ciklusmagban. Hatására a vezérlés a ciklus utáni első utasításra adódik át.

```
loop
  text_io.get(CH);
  exit when CH = '.';
  if CH = '(' or CH = ')' then
    text_io.put(CH);
  end if;
end loop;
KULSO: loop
  --
  loop
    --
    exit KULSO when FELTETEL;
    --
  end loop;
  --
end loop KULSO;
```

5.7. A return utasítás

<return statement> ::= **return** [*<expression>*] ;

A **return** utasítás hatására az alpogram végrehajtása befejeződik. Függvény esetén a **return** utasításban adjuk meg a függvény eredményét is.

A függvény eredményének típusára az Ada nem tesz megkötést, így az nemcsak skalár, hanem összetett típus is lehet.

5.8. A goto utasítás

<goto statement> ::= **goto** *<label name>* ;

A **goto** utasításra a nyelvben valójában nincsen nagy szükség. Azért vették

be az utasításkészletbe, mert van olyan helyzet, amikor a kódot rövidebbé lehet tenni **goto** használatával.

```
<<COMPARE>>
if A(I) < ELEMENT then
   if LEFT(I) /= 0 then
      I := LEFT(I);
      goto COMPARE;
   end if;
   -- utasítások
end if;
```

6. Alprogramok

6.1. Alprogram definíciója

<subprogram declaration> ::= <subprogram specification> ;
<subprogram specification> ::= **procedure** *<identifier> [<formal part>]*
| **function** *<designator> [<formal part>]* **return** *<type mark>*
<designator> ::= <identifier>|<operator symbol>
<operator symbol> ::= <string literal>
<formal part> ::= (*parameter specification>* { ; *<parameter specification>*})
<subprogram body> ::= <subprogram specification> **is**
[<declarative part>]
begin *<sequence of statements>*
[**exception** *<exception handler>* {*<exception handler>*} *]*
end *[<designator>]* ;

Az Ada nyelv kétféle alprogramot ismer: az eljárást és függvényt. Az alprogramok két fő részből állnak: a specifikációs rész tartalmazza az alprogram paramétereinek (és függvény esetén az eredménynek) a leírását, a törzs tartalmazza az alprogram megvalósítását. A törzs maga is két részre osztható: a deklarációs rész tartalmazza az alprogram lokális deklarációit, a törzs utasításrésze pedig az alprogram hatását definiáló utasításokat.

A specifikációs rész valamely deklarációs részben önállóan is állhat, ekkor az a szerepe, hogy deklarálja az alprogram nevét, ami a programszöveg további részében így már felhasználható az alprogram aktivizálására. A törzs sohasem áll önmagában, mindig megelőzi a specifikációs rész.

```
procedure A; -- önmagában álló specifikációs rész
procedure B is
begin
   A;
end B;
procedure A is
begin
   -- utasítások
end A;
```

(Megjegyzés. Az Ada tervezői sem vállalkoztak rá, hogy az alprogramok specifikációjához olyan elemeket illesszenek, amellyekkel a programozó leírhatja az alprogram feladatát, vagy legalább az előfeltételét. Természetesen olyan formális leírásra gondolunk, amit a futtató rendszer ellenőrzésre fel tud használni. Ilyen megoldásokra más nyelvekben vannak kísérletek, azonban a kifejlesztett specifikációs nyelvek túlságosan primitívek. A probléma jó megoldásának haszna óriási lenne.)

Példa függvényre :

```
type VECTOR is
       array (INTEGER range <>) of REAL;
function INNER(A,B:VECTOR) return REAL is
  SUM : REAL := 0.0;
begin
  for I in A'RANGE loop
    SUM := SUM + A(I) * B(I);
  end loop;
  return SUM;
end INNER;
```

A függvény visszatérési értéke lehet akár tömb vagy rekord is, sőt, az összetett típusú értéket visszaadó függvények hívására alkalmazhatók a típus típusműveletei is. Tehát, ha pl. `F` egy vektor értékű függvény, akkor szabályos az `F(X)(1)` indexelés, ami az eredményvektor 1 -es indexű elemét adja meg, s ugyanígy, ha `F` egy rekord értékű függvény, akkor szabályos az `F(X).S` akalú szelekciós kifejezés (ha `S` a rekord egy szelektora).

Példa eljárásra :

```
procedure ADD_TO_LIST(L:in out LINK; V:in INTEGER) is
  NEXT : LINK;
begin
  NEXT := new CELL'(V,null,L);
  if L /= null then
    L.PRED := NEXT;
  end if;
  L := NEXT;
end ADD_TO_LIST;
```

6.2. Alprogram hívása

<procedure call statement> ::= <procedure name>
[<actual parameter part>] ;
<function call> ::= <function name> [<actual parameter part>]
<actual parameter part> ::= (*<parameter association>*
{ , <parameter association>})
<parameter association> ::= [<formal parameter> => *] <actual parameter>*
<formal parameter> ::= <parameter simple name>
<actual parameter> ::= <expression>|<variable name>
|<type mark> (*<variable name>*)

Az előző pontban definiált `INNER` függvényt így használhatjuk :

```
declare
  P,Q : VECTOR(1 .. 100);
  R:REAL;
begin
  ...
```

```
  R := INNER(P,Q);
  ...
end;
```

Paraméterek nélküli függvények esetében az üres paraméterlistát jelképező zárójelpárt is elhagyjuk.

Az Ada alprogramjai meghívhatják önmagukat is. Megengedett a közvetlen rekurzió, s az is, hogy egy alprogram egy hívási láncolaton keresztül hívja meg önmagát (pl. A hívja B -t, B hívja C, C hívja A -t).

```
type MATRIX is
      array (INTEGER range <>, INTEGER range <>) of REAL;
function DETERMINANT(X : MATRIX) return REAL is
  D : REAL := 0.0;
  B : array (X'RANGE(2)) of BOOLEAN := (others => FALSE);
  procedure THREAD(P:in REAL; E:in INTEGER; I:in INTEGER) is
    F : INTEGER;
  begin
    if I > X'LAST(1) then
      D := D + P * REAL((-1) ** E);
    elsif P /= 0.0 then
      F := 0;
      for J in reverse X'RANGE(2) loop
        if B(J) then
          F := F + 1;
        else
          B(J) := TRUE;
          THREAD(P * X(I,J), E + F, I + 1);
          B(J) := FALSE;
        end if;
      end loop;
    end if;
  end THREAD;
begin
  if X'LENGTH(1) /= X'LENGTH(2) then raise MATRIX_ERROR; end if;
  THREAD(1.0, 0, X'FIRST(1));
  return D;
end DETERMINANT;
```

Az eljáráshívás hagyományos formájában az aktális paraméterek *megfeleltetése* a formális paramétereknek pozíció szerint történik: az első aktuális paramétert megfeleltetjük az első formális paraméternek, a másodikat a másodiknak stb.

```
ADD_TO_LIST (HEAD,3);
```

Az Adában ezen kívül (az eljárás- és a függvényhívásokban egyaránt) használható még a névvel jelölt forma is, ahol az aktuális paramétert megelőzi a formális paraméter neve:

```
ADD_TO_LIST (L => HEAD, V => 3);
```

A névvel jelölt esetben az aktuális paraméterek sorrendje tetszőleges. A paraméterek megfeleltetésének kétféle jelölése keverhető is, de ekkor a pozicionálisan megadott paramétereknek kell elöl állniuk.

```
ADD_TO_LIST (HEAD, V => 3);
```

6.3. Paraméterátadás

Az aktuális-formális paramétermegfeleltetésre ugyanaz a szabály vonatkozik, mint az értékadásra: az aktuális és a formális objektum típusának meg kell egyezni.

A formális paraméter lehet egy határozatlan indexhatárú tömb, vagy egy diszkriminánssal rendelkező rekord típus is, ekkor az aktuális paraméter adja meg a megszorítást.

<parameter specification> ::= *<identifier list>* : *<mode><type mark>*
[:= *<expression>* *]*
<mode> ::= *[* **in** *]* | **in out** | **out**

A paraméterátadás módja szempontjából az alprogramok paraméterei háromfélék lehetnek:

- **in** — a formális paraméter lokális konstans, amelynek értékét az aktuális paraméter adja meg (az alprogramban nem módosítható az értéke);
- **out** — a formális paraméter az alprogram változója, ami az alprogram számításának eredményét viseli, és ezt az értéket kapja meg az aktuális paraméter (az alprogramban nem hivatkozhatunk az értékére);
- **in out** — a formális paraméter az alprogram változója, ami felveszi az aktuális paraméter értékét, és a számítás végén értéket ad vissza az aktuális paraméternek.

Ha a formális paraméter mellől a jelzést elhagyjuk, akkor a feltételtezés **in**. Függvénynek csak **in** fajtájú paramétere lehet.

Az aktuális paraméter egy típuskonverzió is lehet. Ekkor **in** és **in out** módú paraméter esetén a paraméterátadás előtt, **in out** és **out** módú paraméter esetén az értékek visszaadásakor történik meg a konvertálás.

Ha az alprogram végrehajtása egy **exception** kiváltásával fejeződik be, akkor a visszatéréskor az **in out** és az **out** módú paramétereknek megfelelő változók értéke definiálatlan.

Paraméterátadáskor ellenőrzésre kerül az is, hogy az értékek megfelelnek-e a megszorításoknak. Skalár típusok esetén

- a paraméterátadás előtt: az **in** és **in out** paramétereknél ellenőrzésre kerül, hogy az értékek eleget tesznek-e a formális típus megszorításának;
- a paraméterértékek visszaadása előtt: az **in out** és **out** paramétereknél ellenőrzésre kerül, hogy az értékek eleget tesznek-e az aktuális típus meg-

szorításának.

Minden más típus esetén az Ada csak a belépéskor ellenőrzi a paramétereket, s a skalárokra vonatkozó szabályt alkalmazza.

(Megjegyzés. Ha egy **in out** módú formális paraméter típusa, vagy annak egy komponensének típusa altípus, pl. `NATURAL`, s az aktuális paraméter még nem kapott értéket, az az eljárásba való belépéskor futási hibát okozhat. Ha meggondoljuk, hogy egy absztrakt típus vektorral való reprezentálásakor a vektor egyes elemei kezdetben nem rendelkeznek értékkel, akkor látjuk, hogy az Ada szigorú ellenőrzési mechanizmusa milyen könnyen lehet kellemetlen is.)

A pointerek és a skalár értékek átadásakor az Ada előírja, hogy minden nyelvi implementáció úgy köteles az paraméterátadást megvalósítani, hogy **in** és **in out** esetén a hívás előtt az értéket egy lokális változóba átmásolja, **in out** és **out** esetén visszatéréskor az értéket visszamásolja az aktuális paraméterbe. Az összetett típusú értékek átadására az Ada megengedi a fordítóprogram számára az érték szerinti és a cím szerinti paraméterátadás közötti választást.

(Feladat. Készítsünk olyan eljárást, amivel megállapíthatjuk, hogy a rendelkezésünkre álló Ada fordítóprogramban a tömbök átadása érték szerint, vagy cím szerint történik-e!)

6.4. A formális paraméterek default értéke

Az Ada érdekes lehetősége, hogy az **in** módú paramétereknek feltételezett bemenő értéke lehet. Az alprogram specifikációs része tartalmazza ezt a feltételezett értéket, amit akkor vesz fel a formális paraméter, ha neki megfelelő aktuális paramétert nem adtunk meg.

```
type SPIRIT is (GIN,VODKA);
type STYLE is (ON_THE_ROCKS,STRAIGHT_UP);
type TRIMMING is (OLIVE,TWIST);
```

Ha feltételezett értékeket használunk, akkor az alprogram specifikációját így adjuk meg:

```
procedure DRY_MARTINI
          (BASE :in SPIRIT := GIN;
           HOW :in STYLE := ON_THE_ROCKS;
           WITH :in TRIMMING := OLIVE);
```

Itt `BASE`, `HOW` és `WITH` a formális paraméterek.

(Megjegyzés. Az alprogramok paramétereit általában két osztályba sorolhatjuk: az egyikbe azok a paraméterek tartoznak, amelyeken az alprogram a fő operációt végzi, vagy ennek eredményét adja vissza, a másikba azok a paraméterek, amelyek ehhez a fő funkcióhoz kiegészítő adatként szolgálnak. Például egy egyenes szakaszt rajzoló eljárásnak alapvető paramétere a szakasz két végpontja, kiegészítő paramétere a vonal vastagsága. Az ilyen, kiegészítő paraméterekhez adunk meg általában default értéket. A default paraméterértékek beve-

zetése kényelmesebbé is teszi az alprogramok használatát, és általánosabb alprogramok írását is segíti.)

A fenti eljárást pl. a következő formában hívhatjuk:

```
DRY_MARTINI ( VODKA,WITH => TWIST );
DRY_MARTINI ( HOW => STRAIGHT_UP );
DRY_MARTINI;
```

6.5. Alprogramok átlapolása

Korábban már láttuk, hogy a felsorolási típusok konstansai átlapolhatók, azaz több típusnak is lehet ugyanolyan nevű értéke. Ekkor egy ilyen literál esetén az, hogy melyik típus értékéről van szó, a szövegkörnyezetből deríthető ki. Az alprogramok esetében hasonló dologra van lehetőség. Érvényben lehet egyszerre pl. a következő három alprogram :

```
procedure SET(EZ_A_NAP:in NAP);
procedure SET(S:in SZIN);
procedure SET(A:in ALLAPOT);
```

Ekkor különböző alprogramok aktivizálását jelentik a `SET(HETFO);` ill. a `SET(PIROS);` eljáráshívások. Ha pl. `PIROS` maga is átlapolt konstans, akkor minősítést kell alkalmaznunk, és az utasítást így kell írni :

```
SET(SZIN'(PIROS));
```

Ezekután meg kell határoznunk, hogy két alprogramspecifikáció mikor egyezik meg, ill. mikor különböznek. Két alprogramspecifikáció akkor egyezik meg, ha azok ugyanazokat a szintaktikus elemeket ugyanolyan sorrendben tartalmazzák, amennyiben azonban a specifikációk számokat is tartalmaznak, azok értékének (és nem az írásformájuknak) kell megegyezni. Tehát az alábbi két `P` eljárás specifikációja megegyezik, a két `Q` eljárásé különbözik:

```
procedure P --
        (A:in INTEGER := 32);
procedure P(A : in INTEGER := 16#20#);
function Q(A:in INTEGER);
function Q(A: INTEGER);
```

6.6. Az operátorok átlapolása

Az alprogramok neve lehet azonosító, mint korábban is láttuk, de felhasználhatjuk a nyelv operátorait is függvényeink neveként. Például definiálhatjuk a vektorok közötti szorzást a "*" névvel:

```
function "*"(A,B:VECTOR) return REAL is
   ...
end "*";
```

Ez a "*" művelet átlapolását jelenti. Ezután a vektorok közötti szorzást `R := P * Q;` alakban írhatjuk. (Az egészek, valósak, stb. közötti szorzás műveleti jele továbbra is a "*" marad.) Ha a "*" szerepel valahol a programban, és nem volna egyértelmű, hogy melyik "*" függvényről van szó, akkor a fenti értékadás minősítéssel írható:

```
R := VECTOR'(P) * VECTOR'(Q);
```

Egy másik, teljesebb példát is mutatunk. Tf. definiáltuk a következő típust :

```
type IRANY is (E,EK,K,DK,D,DNY,NY,ENY);
```

A relációs operátorok (=, /=, <, <=, >, >=, **in**, **not in**) és a diszkrét típus attributumai automatikusan biztosítottak a típus számára. Ha további műveletekre is szükségünk van, azoknak a programjait magunknak kell megírnunk. Tf. szükségünk van egy additív műveletre, ahol `D + M` azt az irányt jelenti, amit úgy kapunk, hogy `D` -hez hozzáadunk `M` nyolcadot az óra járásának megfelelő irányban. Ez a művelet így definiálható :

```
function "+"(D:IRANY; M:INTEGER) return IRANY is
begin
  return IRANY'VAL((IRANY'POS(D) + M) mod 8 );
end "+";
```

Igen veszélyes lehet a `D2 := D1 + 3 + 2;` formájú utasítás, aminek jelentését zárójelezéssel célszerű egyértelművé tenni.

(Megjegyzés. A programozónak illik ilyenkor a "+" korábbi tulajdonságait —pl. kommutativitás— megtartani, e célból el kell készíteni a

```
function "+"(M:INTEGER; D:IRANY) return IRANY is ...
```

függvényt is.)

Két megkötés érvényes az operátorok átlapolására. Először: az **in**, a **not in**, az **and then** és az **or else** relációk nem átlapolhatók. Másodszor: az `=` csak akkor átlapolható, ha az operandusok **limited private** típusúak. Még ebben az esetben is az `=` eredménye logikai kell hogy legyen. A `/=` sohasem átlapolható, ám mindig automatikusan az `=` eredményének ellentettjét adja.

6.7. Példaprogramok

i) Négyzetgyökvonás

Ez a példa azt mutatja be, hogy az Adában altípusok bevezetésével egyszerűsíthető az előfeltétel ellenőrzése.

```
subtype NON_NEGATIVE_REAL is FLOAT range 0.0 .. FLOAT'LAST;
subtype POSITIVE_REAL is
          FLOAT range FLOAT'SMALL .. FLOAT'LAST;

function SQRT(X : NON_NEGATIVE_REAL;
          EPS : POSITIVE_REAL) return FLOAT is
  OLD_VALUE : FLOAT := 0.0;
  NEW_VALUE : FLOAT := X / 2.0;
begin
  while abs(NEW_VALUE - OLD_VALUE) > EPS loop
    OLD_VALUE := NEW_VALUE;
    NEW_VALUE := 0.5 * (OLD_VALUE + X / OLD_VALUE);
  end loop;
  return NEW_VALUE;
end SQRT;
```

ii) Makrohelyettesítések

Egy olyan programot készítünk, ami egy text-file-t dolgoz fel: felismeri benne a makrodefiníciókat, s a makrohívásokat kicseréli a makro törzsére.

A makrodefiníció formája: **#define** *makronév makrotörzs*. A **#define** szöveg mindig a sor elején kezdődik. A makronév betűkből és számjegyekből álló azonosító, a makrotörzs egy szóközöket nem tartalmazó tetszőleges string. Egy makro hívása a definícióját követően bárhol előfordulhat, kivéve a makrodefiníciós sorokat.

A definiált neveket egy egyszerű táblázatban, egy vektorban (`TT, NT`) tartjuk nyilván. Az Ada vektorműveletek kihasználásával (`ANALYSE`) nagyon röviden kódolható a makroneveknek a makrotörzsekre való kicserélése egy-egy soron belül (gondolkozzunk el azon, hogyan oldanánk ezt meg mondjuk Modula-2 -ben !).

```
with TEXT_IO; use TEXT_IO;
procedure PREPROCESSZOR is
    -- A define neveket és értékeket nyilvántartó táblázat:
    type PNEV is access STRING;
    type NEVPAR is record
        N, E : PNEV;
      end record;
    type TABLA is array (INTEGER range <>) of NEVPAR;
    TT : TABLA(1 .. 128);
    NT : INTEGER := 0; -- a táblázat pillanatnyi hossza
    --
```

```
LINE : STRING(1 .. 128);
N : INTEGER;
F, G : FILE_TYPE;

procedure ENTER(STR:in STRING) is
-- Az STR string egy #define utasításnak a #define
-- utáni részét tartalmazza. ENTER feladata az STR
-- alapján egy új pár felvétele a TT táblázatba.
    I : INTEGER := STR'FIRST;
    I0 : INTEGER;

    procedure LEVALASZT is
    begin
        while I <= STR'LAST and then STR(I) = ' '
            loop I := I + 1;
        end loop;
        I0 := I;
        while I <= STR'LAST and then STR(I) /= ' '
            loop I := I + 1;
        end loop;
    end LEVALASZT;
begin
    NT := NT + 1;
    LEVALASZT;
    TT(NT).N := new STRING'(STR(I0 .. I - 1));
    LEVALASZT;
    TT(NT).E := new STRING'(STR(I0 .. I - 1));
end ENTER;

function FIND(STR : STRING) return INTEGER is
-- Feladata: az STR név megkeresése a TT táblázatban.
-- Visszaadja az elem indexét, ill. -1-et, ha a név
-- nem szerepel a táblázatban.
begin
    for I in 1 .. NT loop
        if TT(I).N.all = STR then return I; end if;
    end loop;
    return -1;
end FIND;

function ANALYSE(STR : STRING) return STRING is
-- Feladata: egy szövegsorban a makronevek kicserélése
-- a makrotörzsekre.
    I : INTEGER := STR'FIRST;
    I0 : INTEGER;
    K : INTEGER;
    function ALPHA(CH:CHARACTER) return BOOLEAN is
    begin
        return (CH >= 'A' and then CH <= 'Z') or else
               (CH >= 'a' and then CH <= 'z');
    end ALPHA;
    function ALNUM(CH:CHARACTER) return BOOLEAN is
    begin
        return (CH >= '0' and then CH <= '9') or else
               ALPHA(CH);
    end ALNUM;
begin
```

```
        if STR'LENGTH = 0 then return ""; end if;
        while I <= STR'LAST and then not ALPHA(STR(I))
            loop I := I + 1;
        end loop;
        I0 := I;
        while I <= STR'LAST and then ALNUM(STR(I))
            loop I := I + 1;
        end loop;
        -- I0 az STR stringben az első betű indexe,
        -- vagy I0 = STR'LAST + 1.
        -- I az I0 indexen kezdődő azonosító
        -- utáni pozíció indexe, vagy I = STR'LAST + 1.
        K := FIND(STR(I0 .. I - 1));
        if K in TT'RANGE then
            return STR(STR'FIRST .. I0 - 1) & TT(K).E.all &
                   ANALYSE(STR(I .. STR'LAST));
        else
            return STR(STR'FIRST .. I - 1) &
                   ANALYSE(STR(I .. STR'LAST));
        end if;
    end ANALYSE;
begin
    PUT("Input file = "); GET_LINE(LINE, N);
    OPEN(F, IN_FILE, LINE(1 .. N));
    PUT("Output file = "); GET_LINE(LINE, N);
    CREATE(G, OUT_FILE, LINE(1 .. N));
    while not END_OF_FILE(F) loop
        GET_LINE(F, LINE, N);
        if LINE(1 .. 7) = "#define" then
            ENTER(LINE(8 .. N));
        else
            PUT_LINE(G, ANALYSE(LINE(1 .. N)));
        end if;
    end loop;
    CLOSE(G);
    CLOSE(F);
end PREPROCESSZOR;
```

7. A package

A package fordítási egységként is szerepeltethető programegység, és mint ilyen a program feldarabolásának egyik fő eszköze.

7.1. A package specifikációja

<package declaration> ::= *<package specification>* ;
<package specification> ::= **package** *<identifier>* **is**
{*<basic declarative item>*}
[**private** {*<basic declarative item>*}]
end [*<package simple name>*]

A package két részből áll, a specifikációs részből és a törzsből. Az előző írja le az exportált információk jellemzőit, az utóbbi tartalmazza (és egyben elrejti) a megvalósítást. (A specifikációs rész látható részében megadott összes információ exportálásra kerül.) Speciálisan előfordulhat az is, hogy a package-nek csak specifikációs része van, törzse nincs (pl. ha a specifikációs rész csak változókat tartalmaz, és ezek mint közös adatmező használatosak). Ha a package specifikációja tartalmazza valamely programegység specifikációját, akkor a programegység törzsének a package-törzsben kell lennie.

```
package STACK is
   procedure PUSH(X:in REAL);
   function POP return REAL;
end STACK;
```

7.2. A package törzse

<package body> ::= **package body** *<package simple name>* **is**
[*<declarative part>*]
[**begin** *<sequence of statements>*
[**exception** *<exception handler>* {*<exception handler>*}]]
end [*<package simple name>*] ;

A package törzse tartalmazza a specifikációs részben felsorolt programegységek törzseit, tartalmazhat lokális deklarációkat, valamint tartalmazhat egy inicializáló részt. A közvetlenül a törzsben definiált objektumok a package törzsének kiértékelésekor jönnek létre, s a package-et tartalmazó egységből való kilépéskor szűnnek meg. Ugyancsak a package törzsének kiértékelésekor hajtódik végre a package inicializáló része. Könyvtári szintű package esetén a törzs kiértékelésére a főprogram végrehajtásának megkezdése előtt, beágyazott package esetén a tartalmazó deklarációs rész kiértékelésekor kerül sor.

A törzsben közvetlenül hivatkozhatunk a törzshöz tartozó specifikációs részben elhelyezett bármely deklarációra.

```
package body STACK is
  MAX : constant := 100;
  S : array (1 .. MAX) of REAL;
  PTR : INTEGER range 0 .. MAX;
  procedure PUSH(X:in REAL) is
  begin
    PTR := PTR + 1; S(PTR) := X;
  end PUSH;
  function POP return REAL is
  begin
    PTR := PTR - 1; return S(PTR + 1);
  end POP;
begin
  PTR := 0;
end STACK;
```

A `STACK` package két alprogramot definiál, amelyek egy (legfeljebb 100 elemű) vermen végeznek műveleteket. A `MAX, S, PTR` azonosítók lokálisak, a package-en kívülről nem érhetők el.

7.3. A private típus

A package-k egy fontos alkalmazási területe a privát típusok kialakítása.

<private type declaration> ::= **type** *<identifier>* [*<discriminant part>*] **is**
    [**limited**] **private** ;
<deferred constant declaration> ::= *<identifier list>* :
    constant *<type mark>* ;

A következő példában megmutatjuk, hogyan építhető fel a privát `COMPLEX` típus, hogyan definiálhatók ennek a műveletei, és ezután hogyan használhatja azt a programozó.

```
package COMPLEX_NUMBERS is
  type COMPLEX is private;
  function "+"(X,Y:COMPLEX) return COMPLEX;
  function "-"(X,Y:COMPLEX) return COMPLEX;
  function "*"(X,Y:COMPLEX) return COMPLEX;
  function "/"(X,Y:COMPLEX) return COMPLEX;
  function CONS(R,I:REAL) return COMPLEX;
  function RL_PART(X:COMPLEX) return REAL;
  function IM_PART(X:COMPLEX) return REAL;
private
  type COMPLEX is
    record RL,IM : REAL; end record;
end;
```

A specifikációs rész tehát két darabra válik, az ún. *látható rész*re és a *privát*

*rész*re. A privát rész csak a package törzsében látható, kívülről nem. A privát részban adjuk meg a privát típusok definícióját.

(Megjegyzés. A package-n kívüli utasításokban nem hivatkozhatunk a privát részban elrejtett információra. A fordítóprogram viszont természetesen felhasználja ezt a részt is a többi egység fordításakor, pl. egy privát típusú változó definiálásakor a privát részből olvassa el, hogy hány byte-t foglal el a változó. Í nincs szükség olyan jellegű megszorításra, mint a Module-2 -ben, ahol egy átlátsztlan típus reprezentációja csak annyi helyet foglalhat el, mint a `WORD` típus.)

A privát típus is paraméterezhető. Pl. a korábban is használt `TEXT` típus definíciója így írható fel :

```
type TEXT(MAX:NATURAL) is private;
```

Ekkor a privát részben a típus implementációjának is kell tartalmaznia ugyanezt a paramétert.

A következő példában dinamikus tömböt tartalmazó paraméterezett rekorddal reprezentáljuk az `X` típust:

```
package P is
   subtype ARRAY_RANGE is INTEGER range 0 .. 100;;
   type X(SIZE:ARRAY_RANGE) is private;
   -- az X típus műveletei következnek itt
private
   type X(SIZE:ARRAY_RANGE) is
      record
         AY:array (1 .. SIZE) of T;
      end record
end P;
```

Egy 100 hosszúságú vektor típust hoz létre a következő deklaráció:

```
F1 : X(SIZE => 100);
```

A látható részben szerepelhet privát konstans is, pl.

```
I : constant COMPLEX;
```

ekkor a privát részben meg kell adni ennek az értékét :

```
I : constant COMPLEX := (0.0,1.0);
```

A `COMPLEX_NUMBERS package` törzse pl. ez lehet :

```
package body COMPLEX_NUMBERS is
   function "+"(X,Y:COMPLEX) return COMPLEX is
   begin
      return (X.RL + Y.RL, X.IM + Y.IM);
   end "+";
   function "-"(X,Y:COMPLEX) return COMPLEX is
   begin
```

```
    return (X.RL - Y.RL, X.IM - Y.IM);
  end "-";
  function "*"(X,Y:COMPLEX) return COMPLEX is
  begin
    return (X.RL * Y.RL - X.IM * Y.IM,
            X.RL * Y.IM + X.IM * Y.RL);
  end "*";
  function "/"(X,Y:COMPLEX) return COMPLEX is
    B : REAL := Y.RL ** 2 + Y.IM ** 2;
  begin
    return ((X.RL * Y.RL + X.IM * Y.IM) / B,
            (X.IM * Y.RL - X.RL * Y.IM) / B);
  end "/";
  function CONS(R,I:REAL) return COMPLEX is
  begin
    return (R,I);
  end CONS;
  function RL_PART(X:COMPLEX) return REAL is
  begin
    return X.RL;
  end RL_PART;
  function IM_PART(X:COMPLEX) return REAL is
  begin
    return X.IM;
  end IM_PART;
end COMPLEX_NUMBERS;
```

Ha most pl. ezt a reprezentációt kicseréljük polárkoordinátás alakra, a package-re hivatkozó egységben semmit sem kell változtatni.

A `COMPLEX_NUMBERS` package a következő módon használható fel:

```
declare
  use COMPLEX_NUMBERS;
  Z : constant COMLEX := CONS(0.0,0.0);
  C,D:COMPLEX;
  R,S:REAL;
begin
  C := CONS(1.5, -0.6);
  D := C + I;           -- COMPLEX +
  R := RL_PART(D) + 6.0;-- REAL +
end;
```

(Figyeljük meg, hogy pl. `D = I` helyes feltétel, de `D > I` nem az!)

A privát típus műveletei az `=` és a `/=` relációk, s azok az alprogramok, amiket ugyanabban a specifikációs részben specifikáltunk, ahol a privát típus van, s amelyek paraméterlistáján a privát típus szerepel. Privát típusból való származtatáskor a típusműveletek ugyanúgy átszármaznak az új típusra, mint más típusok esetén.

A privát részben és a package törzsében természetesen használható a privát típus reprezentációjából adódó minden típusművelet.

7.4. A limited private típus

A privát típusokra az értékadás és az =, /= relációk ugyanúgy alkalmazhatók, mint más típusok esetében. Ez nem mindig előnyös. (Gondoljuk meg, mi lesz az összehasonlítás eredménye két verem esetén, vagy mi lesz az értékadás hatása egy pointerekkel megvalósított típus esetén.)

A **limited private** változókra csak a package-ben definiált műveletek alkalmazhatók. A **limited private** változókra vonatkozó szabályok:
- a **limited private** változó nem kaphat kezdőértéket (a package-en kívül),
- a **limited private** típus paramétereinek nem lehet kezdőértéke,
- a package-n kívül nem definiálható **limited private** konstans.

Példaként a **limited private** típusra tekintsük meg a következő package-t.

```
package I_O_PACKAGE is
   type FILE is limited private;
   procedure OPEN (F:out FILE; FILE_NAME:in STRING);
   procedure CLOSE(F:in out FILE);
   procedure READ (F:in out FILE; ITEM:out INTEGER);
   procedure WRITE(F:in out FILE; ITEM:in INTEGER);
private
   type FILE is
      record
         -- a file-descriptor leírása (ez gépfüggő)
      end record;
end I_O_PACKAGE;

package body I_O_PACKAGE is
   procedure OPEN (F:out FILE; FILE_NAME:in STRING) is ...
   procedure CLOSE(F:in out FILE) is ...
   ...
begin
   ...
end I_O_PACKAGE;
```

7.5. Példaprogramok

i) Véletlenszámok előállítása

A package a nyelv legalkalmasabb eszköze absztrakt fogalmak megvalósítására. Nem a típusmegvalósítás az egyetlen terület, ahol ilyen célra alkalmazhatjuk a package-t. Ebben a pontban egy másfajta absztrakt fogalom implementálására mutatunk példát.

```
package RNG is
   function RANDOM return FLOAT;
```

```
  SEED : INTEGER;
end RNG;

with CALENDAR; use CALENDAR;
package body RNG is
  MODULUS : INTEGER := 65536;
  MULT : INTEGER := 13849;
  ADDON : INTEGER := 56963;
  function RANDOM return FLOAT is
  begin
    SEED := (SEED * MULT + ADDON) mod MODULUS;
    return FLOAT(SEED) / FLOAT(MODULUS);
  end RANDOM;
begin
  SEED := INTEGER(SECONDS(CLOCK)) mod MODULUS;
end RNG;
```

ii) Halmaz megvalósítása

Az ORDERED_SET package egy olyan halmazt valósít meg, amelynek elemeit az ID diszkrét típus értékeivel, mint kulcsokkal azonosítjuk, s ezekhez a kulcsokhoz DURATION típusú értékekkel megadott időtartamok tartoznak. A SMALLEST alprogramot függvényként megvalósítani nem volna szerencsés, ennek ugyanis mellékhatása van: azon kívül, hogy a legkisebb elem indexét megadja, ezt az elemet ki is veszi a halmazból.

```
type ID is range 1 .. 128;

package ORDERED_SET is
  procedure INSERT(JOB:in ID; T:in DURATION);
  procedure SMALLEST(JOB:out ID);
  function EMPTY return BOOLEAN;
end ORDERED_SET;

package body ORDERED_SET is
  IN_SET : array (ID) of BOOLEAN := (ID => FALSE);
  RANK : array (ID) of DURATION;

  procedure INSERT(JOB:in ID; T:in DURATION) is
  begin
    IN_SET(JOB) := TRUE; RANK(JOB) := T;
  end INSERT;

  procedure SMALLEST(JOB:out ID) is
  -- Feltételezzük, hogy a halmaz nem üres.
    T : DURATION := DURATION'LAST;
    SMALL : ID;
  begin
    for I in ID loop
      if IN_SET(I) and then RANK(I) <= T then
        SMALL := I; T := RANK(I);
      end if;
    end loop;
    IN_SET(SMALL) := FALSE;
    JOB := SMALL;
```

```
    end SMALLEST;
    function EMPTY return BOOLEAN is
    begin
      for I in ID loop
        if IN_SET(I) then return FALSE; end if;
      end loop;
      return TRUE;
    end EMPTY;
  end ORDERED_SET;
```

Ez a package egyetlen halmaz objektumot valósít meg. Ennél általánosabb megoldás az, amit a következő példában mutatunk a heap típusra.

iii) A heap típus megvalósítása

A heap típus egyetlen komponens típust tartalmazó iterált típus. A heap alaptípusa csak olyan típus lehet, amelyen értelmezve van egy rendezési reláció. A típushoz öt műveletet definiálunk:

create : $\rightarrow$ *Heap*
push : *Heap* $\times$ *Basetype* $\rightarrow$ *Heap*
top : *Heap* $\rightarrow$ *Basetype*
pop : *Heap* $\rightarrow$ *Heap*
empty : *Heap* $\rightarrow$ *Boolean*

A típus sajátossága, hogy a *top* és *pop* művelet mindig a sorozat legnagyobb elemét választja le.

empty(create) = `true`
empty(push(h,x)) = `false`
top(create) = `error`
pop(create) = `error`
push(pop(h), top(h)) = *h*
pop(push(h,x)) = `if` *empty(h)* `or` *top(h)* $\leq$ *x* `then` *h* `else` *push(pop(h), x)*
top(push(h, x)) = `if` *empty(h)* `or` *top(h)* $\leq$ *x* `then` *x* `else` *top(h)*

Egy olyan reprezentációt választunk, amelyben a legnagyobb elem könnyen elérhető, de egy elem behelyezése vagy törlése is kevés műveletet igényel. A heap-et egy `h(1..SIZE)` vektorral és egy `n` egésszel ábrázoljuk. `n` a sorozat hossza és `n = 0` jelenti az üres heap-et, a sorozatot pedig a `h(1)`, ..., `h(n)` elemekkel fejezzük ki. A reprezentációhoz hozzárendelt invariáns: $\forall 1 \leq i \leq \frac{n}{2} : h_i \geq max(h_{2i}, h_{2i+1})$.

Ez a vektorral való ábrázolás megfelel a heap egy kiegyensúlyozott rendezett bináris fával való ábrázolásának. Az invariáns most azt jelenti, hogy minden részfában a gyökér a legnagyobb elem. Így a sorozat legnagyobb elemét mindig `h(1)` ábrázolja, annak kivételekor pedig `h(2)` vagy `h(3)`, ezek helyére pedig a "belőlük lelógó" elemek egyike "csúszik előre", sít. Egy elem behelyezésének vagy törlésének költsége egyaránt $log\ n$ -nel arányos.

```
package HEAP_TYPE is
   type HEAP(SIZE : NATURAL := 128) is limited private;
   HEAP_EMPTY : exception;
   HEAP_FULL : exception;
   procedure PUSH (H : in out HEAP; X : in INTEGER);
   function TOP (H : HEAP) return INTEGER;
   procedure POP (H : in out HEAP);
   function EMPTY (H : HEAP) return BOOLEAN;
private
   type VECTOR is array (INTEGER range) <>) of INTEGER;
   type HEAP(SIZE : NATURAL := 128) is
      record
         N : NATURAL := 0;
         H : VECTOR(1 .. SIZE);
      end record;
end HEAP_TYPE;

package body HEAP_TYPE is
   procedure PUSH (H : in out HEAP; X : in INTEGER) is
      I, J : NATURAL; SWAP : BOOLEAN;
   begin
      if H.N = H.SIZE then raise HEAP_FULL; end if;
      H.N := H.N + 1; J := H.N; SWAP := TRUE;
      while SWAP and then J >= 2 loop
         I := J; J := J / 2;
         if X > H.H(J)
            then H.H(I) := H.H(J);
            else SWAP := FALSE;
         end if;
      end loop;
      if SWAP then H.H(J) := X; else H.H(I) := X; end if;
   end PUSH;

   function TOP(H : HEAP) return INTEGER is
   begin
      if H.N = 0 then raise HEAP_EMPTY; end if;
      return H.H(1);
   end TOP;

   procedure POP(H : in out HEAP) is
      I, J : NATURAL; SWAP : BOOLEAN; TMP : INTEGER;
   begin
      if H.N = 0 then raise HEAP_EMPTY; end if;
      TMP := H.H(H.N); H.N := H.N - 1;
      J := 1; SWAP := TRUE;
      while SWAP and then 2*J <= H.N loop
         I := J; J := 2 * J;
         if J < H.N and then H.H(J+1) > H.H(J)
            then J := J + 1;
         end if;
         if TMP < H.H(J)
            then H.H(I) := H.H(J);
            else SWAP := FALSE;
         end if;
      end loop;
      if SWAP then H.H(J) := TMP; else H.H(I) := TMP; end if;
```

```
    end POP;
    function EMPTY (H : HEAP) return BOOLEAN is
    begin
      return H.N = 0;
    end EMPTY;
end HEAP_TYPE;
```

8. Hibakezelés, kivételkezelés

Az Ada megadja a programozó számára annak lehetőségét, hogy hibák és egyéb kivételes végrehajtási helyzetek esetén rendelkezzék ezek lekezeléséről.

<exception declaration> ::= <identifier list> **: exception ;**
<exception handler> ::= **when** *<exception choice>* { **|** *<exception choice>*}
=> *<sequence of statements>*
<exception choice> ::= <exception name>| **others**

A nyelv tartalmaz predefinit kivételeket is, és a felhasználó maga is definiálhat kivételeket.

```
ADAT_HIBA : exception;
```

A blokkok és alprogramok törzsének végén a programozó utasításokat (ún. exception handler) helyezhet el a kivételek lekezelésére.

```
begin
   -- utasítássorozat
exception
   when ADAT_HIBA =>
      -- a hibát lekezelő utasítások
end;
```

Ha a törzs egy utasításánál fellép valamilyen végrehajtási hiba, kivétel, akkor a vezérlés átadódik a blokk végén levő kivételkezelő résznek. Ez egy **case**-szerű szerkezet, aminek a kivételhez tartozó ágában levő utasítások végrehajtásával a blokk vagy programegység végrehajtása végetér. Az **exception** utasításban a **when**-ágak után szerepelhet egy **others**-ág is, ami összefogja a fel nem sorolt kivételeket.

Az Ada predefinit kivételei a következők:

- `CONSTRAINT_ERROR` — akkor lép fel, ha megszorítás megsértése (pl. egy változó olyan értéket kap, ami kívül esik a típusértékhalmazán), indexhiba, diszkrimináns megsértése, vagy **null** értékű pointer felhasználása következik be;
- `NUMERIC_ERROR` — akkor lép fel, ha egy predefinit numerikus operátor eredménye hibás eredményt ad;
- `PROGRAM_ERROR` — akkor lép fel, ha olyan alprogram, task vagy generic aktivizálására történik kísérlet, aminek a törzse még nincs kiértékelve, ha a függvény végrehajtása befejeződött, de nem hajtódott végre **return** utasítás, vagy ha egy szelektív várakoztatás (ld. 12. 7.1.) minden ága zárt és nincs az utasításnak **else** ága;
- `STORAGE_ERROR` — akkor lép fel, ha nem áll rendelkezésre elég memória az igényelt allokáláshoz;
- `TASKING_ERROR` — taskok közötti kommunikációban léphet fel.

<raise statement> ::= **raise** *[<exception name>]* ;

A kivétel mindig a **raise** utasítás hatására lép fel.

```
begin
   -- utasítások
   raise ADAT_HIBA;
   -- utasítások
exception
   when NUMERIC_ERROR | CONSTRAINT_ERROR =>
      PUT("Numeric or constrain error occured");
   when STORAGE_ERROR =>
      PUT("Ran out of space");
   when others =>
      PUT("Something else went wrong");
end;
```

Kivétel bekövetkezésekor a futtató rendszerben a kivételkezelő kiválasztása attól függ, hogy a kivétel a deklaráció kiértékelése, vagy az utasítások végrehajtása során lépett-e fel. Ha kivétel lép fel egy blokk deklarációs részének kiértékelésekor, akkor a blokk végrehajtása abbamarad, s az exception a blokk végrehajtását kezdeményező egységben lép fel. A végrehajtást kezdeményező egység package és blokk esetében a tartalmazó blokk, alprogram esetében a hívó blokk, task esetében az aktivizáló blokk. A programegység vagy blokk végrehajtható részében fellépő kivételt lekezelő programrész magában a blokkban van. Ha a blokk nem tartalmaz ehhez a kivételhez rendelkezést, akkor a blokk végrehajtása befejeződik, és a kivétel fellép a blokk végrehajtását kezdeményező egységben. Ha a kivételkezelő program végrehajtása közben következik be hiba, akkor a blokk végrehajtása befejeződik, és a kivétel a a blokk végrehajtását kezdeményező egységben lép fel. (Ha egy alprogram hibával tér vissza, akkor az nem ad vissza értéket az **out** és **in out** módú paramétereiben.)

A kivételt háromféle módon használhatjuk.

Először. Használni lehet egyszerű módon hibajelzések adására: ha a program adathibát észlel, végrehajt egy **raise** utasítást, és ezzel jelzi az őt aktivizáló blokknak a számítást sikertelenségét.

Másodszor. A kivételkezelő program kialakítható úgy, hogy felismerje a hibát, korrigálja azt, és a programegység vagy blokk helyett adjon egy többé-kevésbé használható eredményt, megmentve a programot a leállástól.

Harmadszor. A programegység a kivétel felismerése után még elvégez néhány befejező műveletet, és csak ezután jelzi a hibát a hívó szintjén.

```
with DATA_IO; use DATA_IO;
procedure ANALYSE
          (FILE_NAME:in STRING; RES:out ANSWER) is
   procedure INIT(RESULTS:in out ANSWER) is
   begin
```

```
    -- a RESULTS inicializálása
  end INIT;
  procedure UPDATE
        (CURRENT:in DATA; RES:in out ANSWER) is
  begin
    -- CURRENT feldolgozása és RESULTS javítása;
    -- a hibákat ezeken keresztül jelezzük
  end UPDATE;
  procedure PROCESS_FILE
        (FILE_NAME:in STRING; RES:out ANSWER) is
    F:IN_FILE;
    X:DATA;
  begin
    OPEN(F,FILE_NAME); INIT(RES);
    while not END_OF_FILE(F) loop
      READ(F,X); UPDATE(X,RES);
    end loop;
    CLOSE(F);
  exception
    when others => CLOSE(F); raise;
  end PROCESS_FILE;
begin
  --
  PROCESS_FILE(FILE_NAME,RESULTS);
  --
exception
  -- a fellépett kivételek lekezelése
end ANALYSE;
```

A PROCESS_FILE eljárásban le kell zárni a file-t, és a hibát tovább kell adni az ANALYSE eljárásnak. (A paraméter nélküli **raise** utasítás hatására a lekezelt kivétel újra fellép.)

A kivétel lekezelésének ez a fejlett rendszere biztosítja, hogy nagy megbízhatóságú software rendszereket készíthessünk Ada nyelven.

(Megjegyzés. Azért vannak az Ada hibakezelő mechanizmusának komoly hiányosságai is. Mivel egy hiba fellépésekor a **raise** utasítás nem ad vissza semmilyen más információt, csak a hiba tényét, ezért, ha ennél több információt akarunk a hívónak visszaadni, akkor globális változókat kell bevezetnünk. A másik kellemetlen eset akkor áll elő, ha valahol egy hívó nem kezel le egy hibát, így az továbbadódhat, s végül egy olyan programrészhez is eljuthat, ahol már nem kezelhető le értelmesen. A probléma az, hogy a hiba továbbterjedésének semmi határa nincs kijelölve. Hasonló probléma léphet fel a **when others => null;** szerkezet fegyelmezetlen használatakor, ami olyan hibákat kezelhet le, jobban mondva szüntethet meg, amiket "másnak szántak", s így nem jut el a címzetthez.)

8.1. Példaprogramok

i) Verem objektum megvalósítása

Absztrakt objektumok vagy típusok megvalósításakor a legtöbb korábbi nyelvben komoly gondot okoz az objektum kezelésével kapcsolatos hibák jelzése és kezelése. Általában az egyetlen használható megoldás az, hogy a típusműveleteket kiegészítjük egy kimenő logikai paraméterrel. Ez a megoldás azonban egyrészt lassúbbá teszi az alprogramhívást, másrészt visszatérés után mindig ellenőrizni kell a visszaadott értéket, hogy megtudjuk, sikerült-e a művelet végrehajtása. Mindez azért kellemetlen, mert csak nagyon ritkán, vagy soha elűem oforduló helyzetek kezelésére kell nem kevés energiát fordítanunk. Az Ada által bevezetett hibakezelés megfelelő megoldás ennek a problémának a kezelésére.

```
package STACK is
   ERROR : exception;
   procedure PUSH(X:in REAL);
   function POP return REAL;
end STACK;

package body STACK is
   MAX : constant := 100;
   S : array (1 .. MAX) of REAL;
   PTR : INTEGER range 0 .. MAX;
   procedure PUSH(X:in REAL) is
   begin
      if PTR = MAX then raise ERROR; end if;
      PTR := PTR + 1; S(PTR) := X;
   end PUSH;
   function POP return REAL is
   begin
      if PTR=0 then raise ERROR; end if;
      PTR := PTR - 1; return S(PTR - 1);
   end POP;
begin PTR := 0;
end STACK;
```

9. Láthatósági szabályok

A deklaráció egy azonosítónak és egy egyednek az összekapcsolása. A deklaráció *hatásköre* a programszövegnek az szakasza, ahol az azonosítóval a deklarált egyedre hivatkozhatunk. *Közvetlen hatáskör*nek hívjuk a szövegnek azt a szakaszát, amely a deklaráció pontjától a deklaráló egység végéig tart. Némely esetben a hatáskör nagyobb a közvetlen hatáskörnél, kiterjed a deklarációt tartalmazó deklarációs rész határáig. Ilyen esetek Adában

- a package specifikációs részében levő deklaráció,
- egy **entry** deklarációja,
- rekordkomponens deklarációja,
- diszkrimináns,
- alprogramparaméter specifikációja,
- generic paraméter specifikációja.

A szöveg egy adott helyén egy azonosító jelentését a *láthatóság*i és az átlapolási szabályok határozzák meg. Az Adában egy deklaráció a hatáskörén belül mindenütt látható, ez a láthatóság azonban lehet *közvetlen* vagy lehet *szelekciós láthatóság*. Pl. a package-ben definiált eljárásra a package-n kívül szelekciós formában (`PACKAGE.ELJARAS`) hivatkozhatunk, és ugyanez a helyzet a rekord komponenseivel is.

Az Adában, mint minden ALGOL-szerű nyelvben, a programegységek egymásbaágyazhatók. Ezt a szerkezetet *blokkstruktúrá*nak nevezzük. Egy beágyazott blokk szempontjából a külső blokk deklarációit *globális*nak, a blokk saját deklarációit pedig *lokális*nak mondjuk.

```
procedure P is
   A, B, C : INTEGER;
   procedure Q is
      A, C : INTEGER;
      procedure R is
         A : INTEGER;
      begin
         -- közvetlenül látható nevek: Q a P-ből, R a Q-ból,
         -- A az R-ből, B a P-ből, C a Q-ból;
         -- csak szelekciós hivatkozással elérhető nevek: P.A, P.C, Q.A;
         -- globális nevek: P, P.A, P.B, P.C, Q, Q.A, Q.C;
         -- lokális nevek: R, A;
      end R;
   begin
      -- közvetlenül látható nevek: Q a P-ből, R a Q-ból,
      -- A a Q-ból, B a P-ből, C a Q-ból;
      -- csak szelekciós hivatkozással elérhető nevek: P.A, P.C;
      -- globális nevek: P, P.A, P.B, P.C;
      -- lokális nevek: Q, A, C, R;
   end Q;
```

```
  procedure S is
    B : INTEGER;
  begin
    -- közvetlenül látható nevek: Q a P-ből, S a P-ből,
    -- A a P-ből, B az S-ből, C a P-ből;
    -- csak szelekciós hivatkozással elérhető nevek: P.B;
    -- globális nevek: P, P.A, P.B, P.C, Q;
    -- lokális nevek: S, B;
  end S;
begin
  -- közvetlenül látható nevek: Q a P-ből, S a P-ből,
  -- A a P-ből, B a P-ből, C a P-ből;
  -- lokális nevek: P, A, B, C, Q, S;
end P;
```

A programszöveg egy adott pontján egy azonosítóhoz több egyed is tartozhat, azaz ugyanaz az azonosító több közvetlenül látható deklarációra is hivatkozhat. Ezt az azonosítók *átlapolás*ának nevezzük. Az Adában átlapolhatók a felsorolási konstansok nevei és az alprogramok nevei. Ilyenkor az azonosító környezete (pl. a paraméterlista) alapján csak egyetlen olyan deklaráció lehet, ami minden szempontból megfelel a hivatkozásnak, különben a hivatkozás szintaktikus hibát jelent. Az alábbi példában a `P` eljárásban a `T` típus `/=` operátora két helyről is látszik, ezért a hivatkozásnál a package nevének megadásával egyértelművé kell tenni a hivatkozást:

```
package A is
  type T is new INTEGER;
end A;
package B is
  subtype T1 is T;
end B;
with A, B; use A, B;
procedure P(X, Y :in T) is
begin
  if B."/="(X, Y) then ... -- így helyes
end P;
```

```
type ACC_STRING is access STRING;
function F(X : INTEGER) return ACC_STRING;
function F return ACC_STRING;
-- Most F(3)'ADDRESS nem egyértelmű kifejezés.
```

A deklaráció *eltakarás*ának (elrejtésének) nevezzük azt az esetet, ha egy érvényes deklaráció hatáskörében egy belső blokkban ugyanahhoz az azonosítóhoz egy másik egyedet rendelünk. A kívül deklarált egyedre ilyenkor csak szelekciós jelöléssel hivatkozhatunk.

```
procedure P is
  A,B:BOOLEAN;
  procedure Q is
    C:BOOLEAN;
    B:BOOLEAN;
  begin
```

```
    B := A;   -- Q.B := P.A
    C := P.B; -- Q.C := P.B
  end Q;
begin
  A:=B;       -- P.A := P.B
end P;
```

Itt `Q` -ban a `P` -beli `B` -re csak szelektíven lehet hivatkozni, mert `B` `Q`-beli deklarációja eltakarja a `P`-belit.

9.1. A use utasítás

<use clause> ::= **use** *<package simple name>* { , *<package simple name>*} ;

A **use** utasítás a *szelekciós hivatkozás* lerövidítésére szolgál. Példa egymásbaágyazott blokkok és **use** esetén a hivatkozási szabályok alkalmazására:

```
package D is
  T,U,V : BOOLEAN;
end D;

package P is
  package E is
    B,W,V : INTEGER;
  end E;
  procedure Q is
    T,X : REAL;
    use D,E;
  begin
    -- Ezen a ponton
    -- T jelentése Q.T (és nem D.T),
    -- U jelentése D.U,
    -- B jelentése E.B,
    -- W jelentése E.W,
    -- X jelentése Q.X,
    -- V illegális, D.V vagy E.V alakban kell V-re hivatkozni
  end Q;
begin
  ...
end P;
```

9.2. Az átnevezés

Lehetséges az eddig felsoroltakon kívül még egy másik hivatkozási mód is: átnevezhetjük a látható objektumokat.

<renaming declaration> ::= <identifier> :
<type mark> **renames** *<object name>* ;
|*<identifier>* : **exception renames** *<exception name>* ;
| **package** *<identifier>* **renames** *<package name>* ;
|*<subprogram specification>* **renames** *<subprogram or entry name>* ;

Az alábbi átnevezések mind szabályosak.

```
function REAL_PLUS(LEFT, RIGHT : REAL) return REAL renames "+";
function NEXT(C : COLOR) return COLOR renames COLOR'SUCC;
declare
  procedure SPUSH(X:in REAL) renames STACK.PUSH;
  function SPOP return REAL renames STACK.POP;
begin
  SPUSH(X);
  ...
  X := SPOP;
end;
```

Az átnevezés egyaránt használható a szöveg rövidítésére, a nevek közötti konfliktusok feloldására és a végrehajtási idő csökkentésére.

```
L : INTEGER renames D.EV;
AI : REAL renames A(I);
```

Ezután az `AI := AI + 1;` forma használatával elkerüljük az indexkifejezés többszöri kiszámítását. (Ha `I` értéke a programban megváltozik, azért az `AI` továbbra is a korábbi változót jelenti.)

10. Programstruktúra

A fordítási egységek kétfélék: ún. könyvtári egységek (alprogram, package vagy generic specifikációja, generic példányosítása (ld. 11.2), alprogram törzse) vagy másodlagos egységek (könyvtári egység törzse vagy alegység, ld. 10.1.). A könyvtári egységek nevei különbözők kell legyenek. Ada főprogram olyan könyvtári eljárás lehet, amelynek nincs paramétere.

<compilation> ::= {*<compilation unit>*}
<compilation unit> ::= *<context clause><library unit>*
|*<context clause><secondary unit>*
<library unit> ::= *<subprogram declaration>*|*<package declaration>*
|*<generic declaration>*|*<generic instantiation>*|*<subprogram body>*
<secondary unit> ::= *<library unit body>*|*<subunit>*
<library unit body> ::= *<subprogram body>*|*<package body>*
<context clause> ::= {*<with clause>* {*<use clause>*}}
<with clause> ::= **with** *<library unit simple name>*
{ , *<library unit simple name>*} ;

Mivel a fordítási egységek között különféle kapcsolatok lehetnek (egy egység importálhat egy másikból, specifikáció-törzs kapcsolat, törzs-alegység kapcsolat), egy egység fordításakor a fordítóprogramnak szüksége van a hivatkozott egységekben definiált bizonyos információkra. Ezért a lefordított egységek az ún. Ada könyvtárba kerülnek, ahol az egységek object kódján kívül megőrzésre kerülnek a szükséges szimboltáblák is. A könyvtár nyilvántartja az egységek közötti kapcsolatokat is.

Ha egy egység fordításakor a fordítóprogram hibát észlel, akkor az egység nem kerül be a könyvtárba. Ha olyan egységet fordítunk, ami már szerepel a könyvtárban, akkor az új egység a régi helyére kerül, s *érvénytelen*né válik a könyvtárban minden olyan egység, ami az új egységtől függ. Az érvénytelenné vált egységek a könyvtáron belül *újrafordíthatók* (tehát anélkül, hogy egy fordítással bevinnénk a könyvtárba, azaz a fordítás és az újrafordítás két külön művelet).

Az egységek fordítási sorrendjére vonatkozó szabályok teljesen megegyeznek a Modula-2 szabályaival. Egy programegység akkor fordítható, ha azoknak az egységeknek a specifikációs része, amelyekre ez hivatkozik, már le vannak fordítva. A package specifikációja és törzse egy-egy önálló egység, és ezek közül a specifikációt kell előbb fordítani. Ha egy egység csak a specifikációtól függ, akkor a törzs megváltoztatása miatt ezt az egységet nem kell újra lefordítani. (Az alegységekre vonatkozó szabályokat ld. a 10.1. pontban.)

A könyvtárban vannak a predefinit Ada egységek moduljai. Ilyenek az input-output modulok (ld. 15.), a `CALENDAR` modul (ld. 12.6.) és a rendszerfüggő információkat tartalmazó modulok (`SYSTEM`, `UNCHECKED_CONVERSION`, `UNCHEC-`

KED_DEALLOCATION, MACHINE_CODE stb.).

A könyvtári rendszer biztosítja, hogy a függetlenül fordított egységek kapcsolata egymásnak megfelelő legyen (ezeket ellenőrzi).

(Megjegyzés. Elég sok modulból álló programoknál azt láthatjuk, hogy a modulok a feladat egy-egy logikai része köré csoportosulnak. Pl. egy fordítóprogram moduljainak egy csoportja a szemantikus elemzéshez, másik csoportja a kódgeneráláshoz sít. tartozik. Az ilyen csoportoknak általában csak egy-két olyan moduljuk van, amiből másik csoportbeli modul importál, a többi modul ezt az egy-két modult szolgálja ki. Sajnos sem az Ada, sem más programozási nyelv nem ad eszközt az ilyen "modulcsoportoknak" a kezelésére.)

Minden fordítási egység előtt **with** utasításban kell felsorolni azokat az egységeket, amelyekből importálni akarunk. A **with** a hivatkozott egység által exportált összes deklarációt láthatóvá teszi. A megnevezett package-ben definiált nevekre ezután *package.név* formában hivatkozhatunk. Hogy ezt a hosszadalmas formát elkerüljük, a **with** utasítás után szinte mindig alkalmazunk egy **use** utasítást is, ami közvetlenül láthatóvá teszi a package exportált neveit. (Ha ilyen módon több package-ből is importáltuk ugyanazt az azonosítót, az nem hiba, majd az azonosító használatakor kell egyértelművé tenni, hogy melyikre hivatkozunk.)

(Megjegyzés. A programkészítés szempontjából hasznos lenne, ha importáláskor valamiképpen el lehetne hagyni a programból az importált modulnak a felesleges alprogramjait. Pl. kellemetlen, hogy ha a programozó egy olyan fajta készletet használ, mint a matematikai függvények modulja, akkor annak minden alprogramja hozzászerkesztődik a programhoz, pedig talán csak az SQRT függvényre lenne szükség.)

Példa teljes Ada programra :

```
with TEXT_IO, REAL_OPERATIONS; use REAL_OPERATIONS;
procedure QUADRATIC_EQUATION is
  A,B,C,D : REAL;
  use REAL_IO ,
    TEXT_IO ,
    REAL_FUNCTIONS;
begin
  GET(A); GET(B); GET(C);
  D := B ** 2 - 4.0 * A * C;
  if D < 0.0 then
    PUT("Imaginary Roots.");
  else
    PUT("Real Roots : X1 = ");
    PUT((-B - SQRT(D)) / (2.0 * A)); PUT(" X2 = ");
    PUT((-B + SQRT(D)) / (2.0 * A));
  end if;
  NEW_LINE;
end QUADRATIC_EQUATION;
```

Tekintsük a következő programot.

```
procedure MAIN is
  type COLOR is (RED,BLUE,GREEN,YELLOW);
  package SETS is
    type CSET is private;
    procedure INSERT(C:in COLOR; CS:in out CSET);
    -- más műveletek
  private
    type CSET is array (COLOR) of BOOLEAN;
  end SETS;
  package body SETS is
    procedure INSERT(C:in COLOR; CS:in out CSET) is
      --
    end INSERT;
    -- más műveletek
  end SETS;
  use SETS;
  A,B:CSET;
begin
  --
  INSERT(RED,A);
  --
end MAIN;
```

Ezt a programot a következőképpen választhatjuk szét önálló programegységekre:

```
package GLOBALS is
  type COLOR is (RED,BLUE,GREEN,YELLOW);
end GLOBALS;
```

```
with GLOBALS; use GLOBALS;
package SETS is
  type CSET is private;
  procedure INSERT(C:in COLOR; CS:in out CSET);
  -- más műveletek
private
  type CSET is array (COLOR) of BOOLEAN;
end SETS;
```

```
package body SETS is
  procedure INSERT(C:in COLOR; CS:in out CSET) is
    --
  end INSERT;
  -- más műveletek
end SETS;
```

```
with GLOBALS, SETS; use GLOBALS, SETS;
procedure MAIN is
  A,B:CSET;
begin
  --
  INSERT(RED,A);
  --
end MAIN;
```

10.1. Alegységek

<body stub> ::= *<subprogram specification>* **is separate** ;
| **package body** *<package simple name>* **is separate** ;
| **task body** *<task simple name>* **is separate** ;
<subunit> ::= **separate** (*<parent unit name>*) *<body>*

Egy package-ben definiált alprogram törzsét kiemelhetjük a package-ből, és önálló fordítási egységet, ún. *alegység*et képezhetünk. A package-ben az alegység törzsének helyén egy *csonk* marad. Például a `STACK` package törzsét a következőképpen is írhatjuk :

```
package body STACK is
   MAX : constant := 100;
   S : array (1 .. MAX) of REAL;
   PTR : INTEGER range 0.MAX;
   procedure PUSH(X:in REAL) is separate;
   function POP return REAL is separate;
begin
   PTR := 0;
end STACK;
```

A két alegység ekkor külön fordítható, ezek formája a következő:

```
separate STACK
procedure PUSH(X:in REAL) is
begin
   PTR := PTR + 1;
   S(PTR) := X;
end PUSH;
```

```
separate STACK
procedure POP return REAL is
begin
   PTR := PTR - 1;
   return S(PTR + 1);
end POP;
```

Egy alegységben a hatáskör és a láthatóság pontosan olyan, mintha az alegység a csonk helyén lenne. Tehát pl. az alegységben minden külön jelölés nélkül hivatkozhatunk olyan típusokra és globális változókra, amelyek a szülőben vannak definiálva.

Egy alegységből szintén el lehet különíteni valamely beágyazott programegység törzsét, azaz alegységnek szintén lehet alegysége. Az alegységek fordítási sorrendjére az a szabály, hogy egy alegység csak akkor fordítható le, ha már lefordítottuk azt az egységet, amelyből az alegység származik. Ha egy törzset vagy alegységet ismételten lefordítunk, akkor újra kell fordítani ezek összes alegységét.

Azt az egységet, amiből az alegységet leválasztottuk, az alegység *szülő*jének

hívjuk. A szülők láncolatán visszafelé haladva egy olyan egységhez jutunk, ami már nem alegység, hanem valódi törzs. Ezt a törzset az alegység *ős*ének nevezzük.

Nem egyezhet meg két olyan alegységnek a neve, amelyeknek közös ősük van.

10.2. Példaprogramok

i) Prioritási sorok

Ebben a példában több package felhasználásával építünk fel egy absztrakt objektumot.

Tekintsünk egy olyan operációs rendszert, amelyben a jobok 1 és 10 közötti prioritásokkal vannak ellátva. A jobok kezelését a `PRIORITY_QUEUES` package-el képzeljük el.

```
package PRIORITY_QUEUES is
   MAX_SIZE : constant := 50;
   type PRIORITY is new INTEGER range 1 .. 10;
   procedure ADD(P:in PRIORITY; J:in JOB_ID);
      -- A várakozó jobok közé felveszi
      -- a J azonosítójú P prioritású jobot.
   procedure NEXT(J:out JOB_ID);
      -- Kiveszi a legmagasabb prioritású várakozó jobok
      -- közül azt, amelyik a legrégebben várakozik.
   function FULL(P : PRIORITY) return BOOLEAN;
      -- Egy adott prioritáshoz legfeljebb MAX_SIZE darab
      -- várakozó tartozhat. Igazat ad vissza, ha a P
      -- prioritáshoz tartozó várakozási sor tele van.
   function ANY_JOB return BOOLEAN;
      -- Igazat ad vissza, ha van várakozó job.
end PRIORITY_QUEUES;
```

Az egyes prioritásokhoz tartoźo várakozási sorok kezelésére elkészítjük a `FIFO` package-t.

```
package FIFO is
   type QUEUE(MAX: INTEGER) is limited private;
   QUEUE_OVERFLOW : exception;
   QUEUE_UNDERFLOW : exception;
   procedure ADD(Q:in out QUEUE; J:in JOB_ID);
      -- Hozzáveszi a Q sorhoz a J értéket.
   procedure FIRST(Q:in out QUEUE; J:out JOB_ID);
      -- Visszaadja és törli a Q sorból a legrégebbi elemet.
   function FULL(Q : QUEUE) return BOOLEAN;
   function EMPTY(Q : QUEUE) return BOOLEAN;
private
   type JOBS is array (INTEGER range <>) of JOB_ID;
   type QUEUE(MAX: INTEGER) is
      record
```

```
      X : JOBS(1 .. MAX);
      FIRST, LAST : INTEGER := 1;
      CUR_SIZE : INTEGER := 0;
    end record;
end FIFO;

package body FIFO is
  procedure ADD(Q:in out QUEUE; J:in JOB_ID) is
  begin
    if FULL(Q) then raise QUEUE_OVERFLOW; end if;
    Q.X(Q.LAST) := J;
    Q.LAST := Q.LAST mod Q.X'LENGTH + 1;
    Q.CUR_SIZE := Q.CUR_SIZE + 1;
  end ADD;

  procedure FIRST(Q:in out QUEUE; J:out JOB_ID) is
  begin
    if EMPTY(Q) then raise QUEUE_UNDERFLOW; end if;
    J := Q.X(Q.FIRST);
    Q.FIRST := Q.FIRST mod Q.X'LENGTH + 1;
    Q.CUR_SIZE := Q.CUR_SIZE - 1;
  end FIRST;

  function FULL(Q : QUEUE) return BOOLEAN is
  begin
    return Q.CUR_SIZE = Q.X'LENGTH;
  end FULL;

  function EMPTY(Q : QUEUE) return BOOLEAN is
  begin
    return Q.CUR_SIZE = 0;
  end EMPTY;

end FIFO;
```

A PRIORITY_QUEUES package törzse most már minden nehézség nélkül megírható a FIFO package felhasználásával.

```
with FIFO;
package body PRIORITY_QUEUES is
  PQ : array (PRIORITY) of FIFO.QUEUE(MAX_SIZE);
  procedure ADD(P:in PRIORITY; J:in JOB_ID) is
  begin
    FIFO.ADD(PQ(P), J);
  end ADD;

  procedure NEXT(J:out JOB_ID) is
  begin -- Feltételezi, hogy van várakozó job.
    for I in reverse PRIORITY loop
      if not FIFO.EMPTY(PQ(I)) then
        FIFO.FIRST(PQ(I), J);
        return;
      end if;
    end loop;
  end NEXT;

  function FULL(P : PRIORITY) return BOOLEAN is
  begin
    return FIFO.FULL(PQ(P));
```

```
    end FULL;
    function ANY_JOB return BOOLEAN is
    begin
      for I in PRIORITY loop
        if not FIFO.EMPTY(PQ(I)) then return TRUE; end if;
      end loop;
      return FALSE;
    end ANY_JOB;
  end PRIORITY_QUEUES;
```

ii) Keresztreferencia lista készítése

Ebben a pontban egy Ada nyelvű programokról keresztreferencia listát nyomtató programot készítünk. A program névsor szerint felsorolja az Ada programban szereplő azonosítókat, s mindegyikről megadja, hogy mely sorokban fordul elő. A megjegyzéseket, szövegliterálokat és az alapszavakat a program nem veszi figyelembe.

A program két részből áll: egy scanner-ből, ami az input szöveget "megválogatja", kihagyja belőle a feladat szempontjából érdektelen részeket, és egy táblakezelőből, ami nyilvántartja az azonosítókat és a sorszámokat, valamint nyomtatni tudja ezt a táblázatot. Az elemző részt a főmodul foglalja magába, először ezt adjuk meg. A főmodul hivatkozik a táblakezelő package-re, aminek a specifikációs része a következő :

```
with TEXT_IO; use TEXT_IO;
package TABLE_HANDLER is
    type TABLE is private;
    TABLE_FULL : exception;
    procedure INIT_TABLE(T:in out TABLE);
    procedure ENTER(T:in TABLE; NAME:in STRING; N:in INTEGER);
        -- Felvesz egy nevet a táblába, ha az még nem
        -- szerepelt benne, s hozzáírja a sorszámot.
    procedure TABULATE(F:in FILE_TYPE; T:in TABLE);
        -- Nyomtatja a táblázat teljes tartalmát.
private
    type ITEM;
    type LIST_PTR is access ITEM;
    type ITEM is record
        NUM : INTEGER := 0;
        NEXT : LIST_PTR;
    end record;
    type STRING_PTR is access STRING;
    type WORD;
    type TABLE is access WORD;
    type WORD is record
        KEY : STRING_PTR;
        FIRST : LIST_PTR;
        LEFT, RIGHT : TABLE;
    end record;
```

```
end TABLE_HANDLER;
```

A főmodul a táblázatkezelő modulon kívül csak a text input-output modult használja fel.

```
with TEXT_IO, TABLE_HANDLER;
use TEXT_IO, TABLE_HANDLER;
procedure XREF is
    LNO : NATURAL := 0; -- az aktuális sor sorszáma
    T : TABLE;
    F, G : FILE_TYPE;
    INFILE, OUTFILE : STRING(1 .. 64);
    ID : STRING(1 .. 80);
    CH : CHARACTER;
    K : NATURAL;
    type KEY_TYPE is access STRING;
    type KEY_TYPES is array (INTEGER range <>) of KEY_TYPE;
    KEYS : constant KEY_TYPES(1 .. 63) := (
        new STRING'("abort"), new STRING'("abs"),
        new STRING'("accept"), new STRING'("access"),
        new STRING'("all"), new STRING'("and"),
        new STRING'("array"), new STRING'("at"),
        new STRING'("begin"), new STRING'("body"),
        new STRING'("case"), new STRING'("constant"),
        new STRING'("declare"), new STRING'("delay"),
        new STRING'("delta"), new STRING'("digits"),
        new STRING'("do"), new STRING'("else"),
        new STRING'("elsif"), new STRING'("end"),
        new STRING'("entry"), new STRING'("exception"),
        new STRING'("exit"), new STRING'("for"),
        new STRING'("function"), new STRING'("generic"),
        new STRING'("goto"), new STRING'("if"),
        new STRING'("in"), new STRING'("is"),
        new STRING'("limited"), new STRING'("loop"),
        new STRING'("mod"), new STRING'("new"),
        new STRING'("not"), new STRING'("null"),
        new STRING'("of"), new STRING'("or"),
        new STRING'("others"), new STRING'("out"),
        new STRING'("package"), new STRING'("pragma"),
        new STRING'("private"), new STRING'("procedure"),
        new STRING'("raise"), new STRING'("range"),
        new STRING'("record"), new STRING'("rem"),
        new STRING'("renames"), new STRING'("return"),
        new STRING'("reverse"), new STRING'("select"),
        new STRING'("separate"), new STRING'("subtype"),
        new STRING'("task"), new STRING'("terminate"),
        new STRING'("then"), new STRING'("type"),
        new STRING'("use"), new STRING'("when"),
        new STRING'("while"), new STRING'("with"),
        new STRING'("xor")
      );
    EOFCH : constant CHARACTER) := ASCII.NUL;
    procedure READ(CH:out CHARACTER) is
```

```
        -- Egyszeres előreolvasás megvalósítása.
        -- A file végén egy EOFCH karaktert ad vissza.
        -- Ez az eljárás kezeli az LNO sorszámlálót is.
        FIRST_CH_IN_CUR_LINE : BOOLEAN := TRUE;
    begin
        if FIRST_CH_IN_CUR_LINE then
            FIRST_CH_IN_CUR_LINE := FALSE;
            LNO := LNO + 1;
        end if;
        if END_OF_FILE(F) then
            CH := EOFCH;
        elsif END_OF_LINE(F) then
            SKIP_LINE(F); CH := ASCII.CR;
            FIRST_CH_IN_CUR_LINE := TRUE;
        else
            GET(F, CH);
        end if;
    end READ;

    function IS_KEY(NAME : STRING) return BOOLEAN is
    begin
        for I in KEYS'RANGE loop
            if NAME = KEYS(I).all then return TRUE; end if;
        end loop;
        return FALSE;
    end IS_KEY;

    function IS_ALPHA(X : CHARACTER) return BOOLEAN is
    begin
        return (X >= 'a' and then X <= 'z') or else
               (X >= 'A' and then X <= 'Z');
    end IS_ALPHA;

    function IS_NUM(X : CHARACTER) return BOOLEAN is
    begin
        return X >= '0' and then X <= '9';
    end IS_NUM;

    function IS_ALNUM(X : CHARACTER) return BOOLEAN is
    begin
        return IS_ALPHA(X) or else IS_ALNUM(X);
    end IS_ALNUM;

begin
    INIT_TABLE(T);

    PUT("Input file = "); GET_LINE(INFILE, K); NEW_LINE;
    OPEN(F, IN_FILE, INFILE(1 .. K));
    PUT("Output file = "); GET_LINE(OUTFILE, K); NEW_LINE;
    CREATE(G, OUT_FILE, OUTFILE(1 .. K));

    READ(CH);
    while CH /= EOFCH loop
        if IS_ALPHA(CH) then
            K := 0;
            while CH = '_' or else IS_ALNUM(CH) loop
                K := K + 1; ID(K) := CH; READ(CH);
            end loop;
            if not IS_KEY(ID(1 .. K)) then
```

```
                ENTER(T,ID(1 .. K),LNO);
            end if;
        elsif IS_NUM(CH) then
            while IS_NUM(CH) loop READ(CH); end loop;
        elsif CH = '-' then
            READ(CH);
            if CH = '-' then -- comment
                while CH /= ASCII.CR loop READ(CH); end loop;
            end if;
        elsif CH = ''' then
            READ(CH);
            while CH /= ''' loop READ(CH); end loop;
        elsif CH = '"' then
            READ(CH);
            while CH /= '"' loop READ(CH); end loop;
        else
            READ(CH);
        end if;
    end loop;
    TABULATE(G, T);
    CLOSE(F); CLOSE(G);
end XREF;
```

A táblázatkezelő modul exportálja a TABLE típust és az ENTER és TABULATE eljárásokat. A tábla szerkezete, valamint a keresési és elérési algoritmusok rejtve maradnak a többi modul elöl. A táblázatot most igényesebben ábrázoljuk, mint az előző példában: rendezett bináris fát építünk. A SEARCH eljárás jó példát mutat egy ilyen fában való keresésre, a TRAVERSE_TREE eljárás pedig egy rekurzív alprogram a fa bejárására.

```
with TEXT_IO; use TEXT_IO;
package body TABLE_HANDLER is
    package INT_IO is new INTEGER_IO(INTEGER); use INT_IO;
    procedure INIT_TABLE(T :in out TABLE) is
    begin
        T := new WORD;
    end INIT_TABLE;
    function SEARCH(P : TABLE; NAME : STRING) return TABLE is
        -- Megkeresi a nevet a táblázatban, a ha még nincs
        -- benne, akkor felveszi. Visszaadja a nevet
        -- tartalmazó elem címét.
        PP, Q : TABLE; LESS : BOOLEAN;
    begin PP := P; Q := PP.RIGHT; LESS := FALSE;
        while Q /= null loop
            PP := Q; LESS := NAME < PP.KEY.all;
            if NAME = PP.KEY.all then return PP; end if;
            if LESS then Q := PP.LEFT;
            else Q := PP.RIGHT;
            end if;
        end loop;
        Q := new WORD'(new STRING'(NAME),null,null,null);
        if LESS then PP.LEFT := Q; else PP.RIGHT := Q; end if;
        return Q;
```

```
    end SEARCH;

    procedure ENTER(T:in TABLE; NAME:in STRING; N:in INTEGER) is
        P : TABLE;
        Q : LIST_PTR;
        i : NATURAL;
    begin I := 0;
        P := SEARCH(T, NAME);
        Q := new ITEM'(N, P.FIRST);
        P.FIRST := Q;
    end ENTER;

    procedure PRINT_ITEM(F:in FILE_TYPE; P:in TABLE) is
        -- Kinyomtat egy nevet, s a hozzá tartozó sorszámokat.
        L : constant := 20; H : constant := 6;
        I : INTEGER;
        Q : LIST_PTR;
    begin PUT(F, P.KEY.all);
        if P.KEY'LENGTH <= L then
            PUT(F, (1 .. L - P.KEY'LENGTH => ' '));
        else
            NEW_LINE(F);
        end if;
        I := L + H; Q := P.FIRST;
        while Q /= null loop
            if I + H > 80 then NEW_LINE(F); I := L + H; end if;
            PUT(Q.NUM, H); I := I + H;
            Q := Q.NEXT;
        end loop;
        NEW_LINE(F);
    end PRINT_ITEM;

    procedure TRAVERSE_TREE(F:in FILE_TYPE; P:in TABLE) is
    begin
        if P /= null then
            TRAVERSE_TREE(F, P.LEFT);
            PRINT_ITEM(F, P);
            TRAVERSE_TREE(F, P.RIGHT);
        end if;
    end TRAVERSE_TREE;

    procedure TABULATE(F:in FILE_TYPE; T:in TABLE) is
    begin
        NEW_LINE(F); TRAVERSE_TREE(F, T.RIGHT);
    end TABULATE;

end TABLE_HANDLER;
```

11. Generic

Lényeges, hogy a könyvtári eljárásokat elég általánosan meg lehessen írni, és azokat széles körben, sok feladatra alkalmazni lehessen. Bizonyos általánosság elérhető az alprogramokkal: ezeket paramétereik jó megválasztásával sok hasonló feladat megoldására felhasználhatjuk. A generic lehetővé teszi, hogy az alprogramokat és a package-ket típusokkal és alprogramokkal paraméterezzük, s ezzel az általánosság növelhető.

A generic nem közvetlenül végrehajtható programegység, hanem egy sablon, amivel végrehajtható egyedek hozhatók létre. A generic-ből egy végrehajtható egyed létrehozását *példányosítás*nak (instantiation) nevezzük. A példányosítás mindig fordítási időben hajtódik végre.

(Megjegyzés. A generic a makroutasítás fogalmának továbbfejlesztése. Példányosítással lényegében olyan kódot kapunk, mint ami úgy állna elő, ha a formális paramétereket kézzel behelyettesítve közvetlenül a programegységet írnánk le.)

11.1. Generic deklarációja

<generic declaration> ::= *<generic specification>* ;
<generic specification> ::= *<generic formal part><subprogram specification>*
|*<generic formal part><package specification>*
<generic formal part> ::= **generic** {*<generic parameter declaration>*}
<generic parameter declaration> ::= *<identifier list>* : [**in** [**out**]]
<type mark> [**:=** *<expression>*] ;
| **type** *<identifier>* **is** *<generic type definition>*
|*<private type declaration>*
| **with** *<subprogram specification>* [**is** *name>*] ;
| **with** *<subprogram specification>* [**is** <>] ;
<generic type definition> ::= (<>) | **range** <> | **digits** <>
| **delta** <> |*<array type definition>*|*<access type definition>*

A generic alprogramot vagy package-t ugyanúgy írjuk, mint az egyszerű alprogramot és package-t, csak a specifikáció elé kerül a generic alapszó a generic paramétereinek felsorolásával.

A generic szintaktikus elemzése a generic definiálásakor megtörténik, de a kiértékelésére csak példányosításkor kerül sor (ez fontos a könyvtárak esetében).

A generic programegységet mindenütt elhelyezhetjük, ahol egyszerű programegység állhat. (Így pl. a generic fordítási egység is lehet.) A generic könyvtári egységre ugyanúgy **with**-tel kell hivatkozni, mint más könyvtári egységre (a **use**-nak természetesen nincs értelme, s nem is használható generic-re).

Példaként egy olyan privát verem típust definiálunk, amiben a verem hossza és az elemek típusa a package paramétere.

```
generic
   MAX : INTEGER;
   type ELEM is private;
package STACK is
   procedure PUSH(X:in ELEM);
   function POP return ELEM;
end STACK;
package body STACK is
   S : array (1 .. MAX) of ELEM;
   ...
end STACK;
```

A generic háromféle paraméterrel paraméterezhető: egyszerű adatértékkel vagy objektummal, típussal, és alprogrammal.

A következő példában a generic paramétere objektum:

```
generic
   PERIOD : in NATURAL;
package RING_COUNTER is
   function IS_ZERO return BOOLEAN;
   procedure INCREMENT;
end RING_COUNTER;

package body RING_COUNTER is
   COUNT : INTEGER range 0 .. PERIOD - 1 := 0;
   function IS_ZERO return BOOLEAN is
   begin
      return COUNT=0;
   end IS_ZERO;
   procedure INCREMENT is
   begin
      COUNT := (COUNT + 1) mod PERIOD;
   end INCREMENT;
end RING_COUNTER;
```

A típussal való paraméterezhetőség teszi igazán erőteljessé az Adát.

```
generic
   type DISCRETE is (<>);
function NEXT_OPERATION(X:DISCRETE) return DISCRETE;

function NEXT_OPERATION(X:DISCRETE) return DISCRETE is
begin
   if X=X'LAST then
      return X'FIRST;
   else
      return X'SUCC;
   end if;
end NEXT_OPERATION;
```

A formális típus egy típusosztály is lehet. A formális típusok osztályainak

leírására a következő jeleket használjuk:

(<>)	—	tetszőleges	diszkrét
range <>	—	tetszőleges	INTEGER
delta <>	—	tetszőleges	fixpontos
digits <>	—	tetszőleges	lebegőpontos

Minden, az illető típusosztályhoz tartozó művelet és attributum használható a genericben, kivéve az IMAGE és VALUE attributumokat (ha ezekre mégis szükség volna, meg kell adni őket paraméterként). Megjegyezzük, hogy ha a (<>) típusnak valamely INTEGER típust feleltetünk meg aktuális paraméterként, az aritmetikai műveletek a generic-ben nem használhatók. Vektor típus esetén az index és az elem típusa (hacsak ezek nem standard típusok) is paraméter kell legyen.

A paraméterezés harmadik módja, amikor a paraméter egy alprogram. Ezek általában a formális generic típusok műveletei.

```
generic
  type ITEM is private;
  type INDEX is (<>);
  type VECTOR is array (INDEX range <>) of ITEM;
  with function "<"(X,Y:ITEM) return BOOLEAN;
procedure QUICK_SORT(V:in out VECTOR);

procedure QUICK_SORT(V:in out VECTOR) is
  procedure SORT(LEFT,RIGHT:in INDEX) is
    CENTRE_VALUE: ITEM :=
      V(INDEX'VAL((INDEX'POS(LEFT) + INDEX'POS(RIGHT)) / 2));
    LEFT_IDX : INDEX := LEFT;
    RIGHT_IDX: INDEX := RIGHT;
  begin
    loop
      while V(LEFT_IDX) < CENTRE_VALUE loop
        LEFT_IDX := INDEX'SUCC(LEFT_IDX);
      end loop;
      while CENTRE_VALUE < V(RIGHT_IDX) loop
        RIGHT_IDX := INDEX'PRED(RIGHT_IDX);
      end loop;
      if LEFT_IDX <= RIGHT_IDX then
        declare
          TEMP : ITEM := V(LEFT_IDX);
        begin
          V(LEFT_IDX) := V(RIGHT_IDX);
          V(RIGHT_IDX) := TEMP;
        end;
        LEFT_IDX := INDEX'SUCC(LEFT_IDX);
        RIGHT_IDX := INDEX'PRED(RIGHT_IDX);
      end if;
      exit when LEFT_IDX > RIGHT_IDX;
    end loop;
    if LEFT < RIGHT_IDX then
      SORT(LEFT,RIGHT_IDX);
    end if;
    if LEFT_IDX < RIGHT then
```

```
        SORT(LEFT_IDX,RIGHT);
      end if;
    end SORT;
  begin
    SORT(V'FIRST,V'LAST);
  end QUICK_SORT;
```

(Megjegyzés. Kár, hogy az Ada nem vállalta fel az "alprogram típus" bevezetését, mert így ahhoz, hogy alprogramot adjunk meg paraméterként, mindjárt generic-et kell készítenünk. Pedig az eljárás típust már a Modula-2 is tartalmazza, s a taskok esetében az Ada is bevezeti.)

11.2. Példányosítás

<generic instantiation> ::= **package** *<identifier>* **is**
new *<generic package name>* [*<generic actual part>*] ;
| **procedure** *<identifier>* **is new** *<generic procedure name>*
[*<generic actual part>*] ;
| **function** *<designator>* **is new** *<generic function name>*
[*<generic actual part>*] ;
<generic actual part> ::= (*<generic association>*
{ ; *<generic association>*})
<generic association> ::= [*<generic formal parameter>* =>]
<generic actual parameter>
<generic formal parameter> ::= *<parameter simple name>*|*<operator symbol>*
<generic actual parameter> ::= *<expression>*|*<variable name>*
|*<subprogram name>*|*<entry name>*|*<type mark>*

Eddig a generic definíciójával foglalkoztunk. Most példányosítunk egy 100 hosszúságú, `REAL` elemeket tartalmazó vermet :

```
declare
  package MY_STACK is new STACK(100,REAL);
  use MY_STACK;
begin
  PUSH(X);
  ...
  X := POP();
end;
```

Nézzük meg a `RING_COUNTER` generic felhasználásának módját is:

```
procedure TABULATE (X:in INTEGER_VECTOR) is
  FIELD_WIDTH : constant := 10;
  package COUNTER is new RING_COUNTER(PERIOD => 6);
  use COUNTER;
begin
  for I in X'RANGE loop
    PUT(X(I),FIELD_WIDTH);
    INCREMENT;
    if IS_ZERO then NEW_LINE; end if;
  end loop;
end TABULATE;
```

A QUICK_SORT genericből létrehozhatjuk pl. a következő eljárásokat:

```
procedure STRING_SORT is
    new QUICK_SORT(CHARACTER,POSITIVE,STRING,"<");
```

A **with**-ben itt default értéket lehet megadni:

```
with function "<"(X,Y:ITEM) return BOOLEAN is <>;
```

Ekkor

```
procedure STRING_SORT is
    new QUICK_SORT(CHARACTER,POSITIVE,STRING);
```

helyes, és ekvivalens az előzővel.

(Ellenőrző kérdések. Nézzük meg figyelmesen az alábbi programrészleteket!

```
generic
  type A is (<>);
  type B is private;
package G is
  function NEXT(X : A) return A;
  function NEXT(X : B) return B;
end G;
package P is new G(BOOLEAN, BOOLEAN);
```

Adhat-e hibajelzést a fordítóprogram a fenti részletek fordításakor? Adhat-e hibajelzést P felhasználásakor?

```
package P is
  type T is private;
  DC : constant T;
  generic package PP is end PP;
private
  type T is new INTEGER;
  DC : constant T := -1;
end P;
procedure Q(X:out P.T) is begin X := P.DC; end Q;
generic
  Y:in out P.T;
package CALL is end CALL;
package body CALL is begin Q(Y); end CALL;
package body P is
  Z : T range 0 .. 1 := 0;
  package body PP is
    package CALL_Q is new CALL(Z);
  end PP;
end P;
```

Mi lesz a hatása a **package** CALL_Q_NOW **is new** P.PP; példányosításnak?)

11.3. Példaprogramok

i) Listakezelés

Szinte nincs olyan programozási nyelv, ami tartalmazná a sorozat típust. Ennek az az oka, hogy a sorozat típusú objektumok nagyon komoly memóriagazdálkodást igényelnek, s a nyelvek tervezői attól tartanak, hogy a sorozatokat nem tudják elég hatékonyan implementálni. Ezért a programozóra

hagyják, hogyan akarja implementálni a sorozatot: vektorral, vagy pointerekkel felépített listával. Ebben a pontban egy listakezelő generic package-t mutatunk be.

```
generic
   type ELEM is private;
package LIST_PACKAGE is
   type CELL is private;
   type POINTER is access CELL;
   type ARR is array (INTEGER range <>) of ELEM;
   function MAKE(A : ARR) return POINTER;
   function FRONT(P : POINTER) return ELEM;
   function REST(P : POINTER) return POINTER;
   function ADD_ON(E : ELEM; P : POINTER) return POINTER;
private
   type CELL is
      record
         VALUE : ELEM;
         LINK : POINTER;
      end record;
end LIST_PACKAGE;

package body LIST_PACKAGE is
   function MAKE(A : ARR) return POINTER is
      P : POINTER;
   begin
      for I in reverse A'RANGE loop
         P := ADD_ON(A(I), P);
      end loop;
      return P;
   end MAKE;

   function FRONT(P : POINTER) return ELEM is
   begin
      return P.VALUE;
   end FRONT;

   function REST(P : POINTER) return POINTER is
   begin
      return P.LINK;
   end REST;

   function ADD_ON(E : ELEM; P : POINTER) return POINTER is
   begin
      return new CELL'(E, P);
   end ADD_ON;
end LIST_PACKAGE;

   with LIST_PACKAGE;
procedure LIST_DEMO is
   package INT_LIST is new LIST_PACKAGE(INTEGER);
   P : INT_LIST.POINTER;
begin
   P := INT_LIST.MAKE((1,2,3,4));
   P := INT_LIST.ADD_ON(5, P);
   while P /= null loop
      PUT(INT_LIST.FRONT(P), WIDTH => 2);
      P := INT_LIST.REST(P);
```

```
  end loop;
end LIST_DEMO;
```

ii) Matematikai könyvtári generic

A generic legfontosabb és legnagyobb felhasználási területe a software könyvtárak kialakítása. Az alábbi generic példa egy matematikai könyvtári egységet mutat be:

```
generic
  type REAL is digits <>;
  with function F(X : REAL) return REAL;
function INTEGRATE_G(A, B, EPS : in REAL) return REAL;

function INTEGRATE_G(A, B, EPS : in REAL) return REAL is
-- Függvény határozott integráljának kiszámítása
-- az [A, B] intervallumon a trapéz-módszer alapján.
  NEW_APPROX : REAL := 0.0;
  PREVIOUS_APPROX : REAL := 0.0;
  FINISHED : BOOLEAN := FALSE;
  N : INTEGER; -- az intervallumok száma
  H : REAL; -- az intervallumok mérete
  SUM : REAL;
begin
  N := INTEGER(10.0 * (B - A)) + 1;
  H := (B - A) / REAL(N);
  while not FINISHED loop
    PREVIOUS_APPROX := NEW_APPROX;
    SUM := F(A) / 2.0;
    for I in 1 .. N-1 loop
      SUM := SUM + F(A + REAL(I) * H);
    end loop;
    SUM := SUM + F(B) / 2.0;
    NEW_APPROX := SUM * H;
    FINISHED := abs(NEW_APPROX - PREVIOUS_APPROX) < EPS;
    N := N * 2;
    H := H / 2.0;
  end loop;
  return NEW_APPROX;
end INTEGRATE_G;
```

12. Taskok

A taskot nem tartalmazó program végrehajtása az előző pontokban leírt szabályok szerint, az utasításainak a szekvenciális végrehajtásával történik. Úgy képzelhetjük el, hogy az utasításokat egyetlen "logikai processzor" hajtja végre. A taskok olyan egységek, amelyek végrehajtása párhuzamosan történik. Úgy képzelhetjük el, hogy mindegyik taskot egy saját "logikai processzor" hajt végre. A szekvenciális programokban egyidőben egy utasítás áll végrehajtás alatt, míg a párhuzamos programokban a végrehajtás több szálon fut, egy időpillanatban a programnak több utasítása is lehet végrehajtás alatt. A különböző taskok a szinkronizációs pontoktól eltekintve egymástól függetlenül működnek. (Az, hogy a programot egy vagy több "fizikai processzor" hajtja végre, csak a nyelv implementációjához tartozó kérdés.)

A taskoknak lehetnek ún. **entry** pontjaik. A task egy **entry** pontja hívható egy másik taskból. A hívott taskban az **entry**-hez tartozik egy **accept** utasítás, és a hívás hatására ez hajtódik végre, ezzel oldhatók meg a szinkronizációs feladatok. Az **entry**-nek paraméterei is lehetnek, amelyek a taskok közötti információátadásra szolgálnak.

A taskok szinkronizálása a taskok randevúival történik, ami egy hívást kiadó ill. fogadó task között zajlik le.

A taskot egy task programegységgel definiáljuk, ami task specifikációból és task törzsből áll. Ebben a pontban a task egységre, az **entry**-kre a taskok közötti kapcsolatok leírásához definiált utasításokra (**entry** hívása, **accept**, **delay**, **select** és **abort** utasítás) vonatkozó szabályokat írjuk le.

12.1. A task specifikációja és törzse

<task declaration> ::= <task specification> ;
<task specification> ::= **task** [**type**] *<identifier>* [**is**
{*<entry declaration>*}{*<representation clause>*}
end [*<task simple name>*]]
<task body> ::= **task body** *<task simple name>* **is**
[*<declarative part>*]
begin *<sequence of statements>*
[**exception** *<exception handler>* {*<exception handler>*}]
end [*<task simple name>*] ;

A specifikáció és a törzs szerepe pontosan olyan, mint más programegységek esetében. Az a specifikáció, ami a **task type** alapszavakkal kezdődik, task típust definiál (azaz ezt a taskot objektumok definiálásával megtöbbszörözhetjük). A task típus értékei olyan taskok, amelyeknek az **entry** pontjai pontosan azok, amiket a

specifikációs rész felsorol. A **type** alapszó nélküli specifikáció egyedi taskot definiál. Az ilyen task deklaráció ekvivalens azzal, mintha definiáltunk volna egy task típust és deklaráltunk volna ehhez egy taskot. Példa task típusok és egyszerű taskok specifikálására:

```
task type RESOURCE is
  entry SEIZE;
  entry RELEASE;
end RESOURCE;

task type KEYBOARD_DRIVER is
  entry READ (C:out CHARACTER);
  entry WRITE(C:is CHARACTER);
end KEYBOARD_DRIVER;

task PRODUCER;

task CONSUMER;

task BUFFER is
  entry PUT(C:in CHARACTER);
  entry GET(C:out CHARACTER);
end BUFFER;
```

A task törzse definiálja a task végrehajtásának hatását. A task törzsének végrehajtása a task aktivizálásának hatására történik meg. Példák task törzsre:

```
task body PRODUCER is
  C : CHARACTER;
begin
  loop
    PRODUCE(C);
    BUFFER.PUT(C);
  end loop;
end PRODUCER;

task body CONSUMER is
  C : CHARACTER;
begin
  loop
    BUFFER.GET(C);
    CONSUME(C);
  end loop;
end CONSUMER;
```

12.2. Task típusok, task objektumok

A task típus **limited** típus, így sem az értékadás sem az összehasonlítás (=, /=) nem megengedett task típusú objektumokra, és ezek nem szerepelhetnek **out** módú formális paraméterként sem. Ha a task objektum egy olyan objektum, vagy egy olyan objektumnak a komponense, amit

a) egy objektumdeklarációban deklaráltunk, a task objektum értékét a deklaráció kiértékelése nyomán határozzuk meg;

b) egy allokátor kiértékelése nyomán hozunk létre, a task objektum értékét az

allokátor kiértékelésekor határozzuk meg.

Példák task változók deklarálására :

```
type KEYBOARD is access KEYBOARD_DRIVER;

CONTROL : RESOURCE;
TELETYPE : KEYBOARD_DRIVER;
POOL : array (1 .. 10) of KEYBOARD_DRIVER;
TERMINAL : KEYBOARD := new KEYBOARD_DRIVER;
```

12.3. Taskok aktivizálása, végrehajtása

A task végrehajtását a task törzse definiálja. A végrehajtás kezdeti lépése a task aktivizálása (ez a deklarációs rész kiértékelését is magában foglalja).

A task aktivizálása az objektumot deklaráló deklarációs rész kiértékelésének befejezése után, a törzs első utasításának végrehajtása előtt történik meg. Ha a deklaráció egy package specifikációs részében van, az aktivizálás a törzs deklarációs részének kiértékelése után és a törzs végrehajtása előtt történik meg. Az a task objektum vagy objektum komponens, amit allokátorral hozunk létre, az allokátor kiértékelésekor kerül aktivizálásra. (Ugyanezek érvényesek arra a taskra is, ami egy objektum komponense.)

Ha a task aktivizálása közben hiba történik, a task *komplett* állapotba kerül, az aktivizálás befejezése után fellép a `TASKING_ERROR` a taskokat tartalmazó programegység törzsében. Egy *komplett* állapotba került taskot már semmilyen módon nem lehet újra elindítani.

12.4. Taskok terminálása

Minden task függ legalább egy szülőtől. A szülő lehet szintén task, végrehajtás alatt álló blokk, alprogram vagy könyvtári package. A függés közvetlen az alábbi két esetben.

a) Ha az objektum vagy objektum komponense által jelölt taskot egy allokátor kiértékelése közben hoztuk létre, a task attól az egységtől (mint szülőtől) függ, amelyik az **access** típust kiértékelte.
b) Ha deklaráció kiértékelése közben hoztuk létre, akkor attól az egységtől függ, amelyik a deklaráció kiértékelését kezdeményezte.

Ha a task függ egy szülőtől, amit egy másik egység (szülő) hajt végre, akkor a task (közvetve) ettől a szülőtől is függ.

A task kompletté válik, ha végetér a task törzsében levő utasítások végrehajtása, vagy ha a végrehajtás közben hiba lép fel és nincs a taskban hibakezelő programrész, vagy ha van, és annak a végrehajtása befejeződik. A task *terminál*, ha komplett, és az összes tőle függő task már terminált. Különben a task

pontosan akkor terminál, ha a végrehajtása egy **terminate** utasításhoz ért, és teljesül az alábbi két feltétel:

a) a task olyan szülőtől függ, ami már terminált (a végrehajtása végetért, tehát pl. nem könyvtári package);
b) a szülőtől függő összes task terminált már, vagy szintén egy **terminate** utasításnál várakozik.

Példa :

```
declare
   type GLOBAL is access RESOURCE;
   A, B : RESOURCE;
   G : GLOBAL;
begin          -- A és B aktivizálódik
   declare
      type LOCAL is access RESOURCE;
      X : GLOBAL := new RESOURCE;
      L : LOCAL := new RESOURCE;
      C : RESOURCE;
   begin       -- C aktivizálódik
      G := X;  -- G és X ugyanazt az objektumot jelölik
   end;        -- várakozás C és L.all terminálására
end;           -- várakozás A, B és G.all terminálására
```

A taskokhoz is tartoznak attributumok:

- *TaskName*´CALLABLE — logikai érték, ami pontosan akkor hamis, ha a task már kompletté vált, terminált vagy abortált (vagyis már nem lesz többé hívható).
- *TaskName*´TERMINATED — logikai érték, ami pontosan akkor igaz, ha a task már terminált.
- *EntryName*´COUNT — a megnevezett **entry**-hez tartozó **accept**-re várakozó hívások száma.

(Ellenőrző kérdés. Nézzük meg figyelmesen az alábbi programrészletet!

```
with TEXT_IO; use TEXT_IO;
procedure MAIN is
   task type T;
   task body T is begin null; end T;
   function F return T is
      X : T;
   begin
      return X;
   end F;
begin
   if F´TERMINATED then
      PUT_LINE("terminated");
   else
      PUT_LINE("not terminated");
   end if;
end MAIN;
```

Mit kell kiírnia a programnak?)

12.5. entry, entry hívás, accept utasítás

<entry declaration> ::= **entry** *<identifier>* [(*<discrete range>*)]

[<formal part>] ;
<entry call statement> ::= <entry name> [<actual parameter part>] ;
<accept statement> ::= **accept** *<entry simple name> [(<entry index>)]*
[<formal part>]
[**do** *<sequence of statements>*
end *[<entry simple name>]]* ;
<entry index> ::= <expression>

Az **entry** hívás és az **accept** utasítás szolgálnak elősorban a taskok szinkronizálására és adatcseréjére. Az **entry** deklaráció hasonló az alprogramdeklarációhoz, de csak taskban megengedett. Az **entry** formális és aktuális paraméterrészére ugyanaz érvényes, mint az alprogramok paramétereire.

Az **entry** deklaráció deklarálhatja **entry**-k családját is; a családot diszkrét intervallummal jelöljük ki. A család egy **entry**-jére indexeléssel hivatkozhatunk. A törzsön kívül az **entry** nevekre szelekciós formában kell hivatkozni, ahol az előtag a task objektum neve.

Az **accept** utasítás definiálja azokat a műveleteket (a számítást), amiket akkor kell elvégezni, ha az **entry** hívása megtörtént. Minden **entry**-hez külön **accept** tartozik.

Ha az **entry**-t az adott pillanatban csak egy task hívja, két eset lehetséges.

a) Ha a task végrehajtása még nem jutott el a megfelelő **accept** utasításhoz, a hívó felfüggesztődik.
b) Ha a task végrehajtása egy **accept** utasításhoz ért és nincs a megfelelő **entry**-re hívás, a task felfüggesztődik.

Ha az **entry** hívás pillanatában a hívott **entry**-hez tartozó **accept** utasításnál tart a végrehajtás, végrehajtódik az **accept** utasítás törzse (a hívó task felfüggesztődik). Ez a *randevú*. Az **accept** törzs végrehajtása után mindkét task párhuzamosan fut tovább.

Ha több task hívja ugyanazt az **entry**-t, miközben a végrehajtás nem érte el az **accept** utasítást, a hívások egy várakozási sorba kerülnek. Minden **entry**-hez külön várakozási sor tartozik. Az **accept** utasítás minden egyes végrehajtása kivesz egy várakozót a sorból.

Példa az **accept** utasítás használatára :

```
task body BUFFER is
  BUF : CHARACTER;
begin
  loop
    accept PUT(C:in CHARACTER) do
      BUF := C;
    end PUT;
    accept GET(C:out CHARACTER) do
      C := BUF;
    end GET;
```

```
   end loop;
end BUFFER;
```

12.6. A delay és az abort utasítás

<delay statement> ::= **delay** *<simple expression>* ;

A **delay** t ; utasítás felfüggeszti a taskot t másodpercre. Az Ada a főprogramot magát is egy tasknak fogja fel, ezért a **delay** utasítás használható taskon kívül is, ekkor a főprogramot függeszti fel az adott időre. Az idő megadására az Ada bevezeti a predefinit valós DURATION típust. A DURATION típus gépi reprezentációjára az Ada előírja, hogy DURATION´SMALL nem lehet nagyobb 20 ezredmásodpercnél, a típus legnagyobb értéke nem lehet kisebb, mint 86400.

Az idő kezeléséhez az Ada könyvtár predefinit egységként tartalmazza a CALENDAR package-t.

```
with SYSTEM;
package CALENDAR is
   type TIME is private;

   subtype YEAR_NUMBER is INTEGER range 1901 .. 2099;
   subtype MONTH_NUMBER is INTEGER range 1 .. 12;
   subtype DAY_NUMBER is INTEGER range 1 .. 31;
   subtype DAY_DURATION is DURATION range 0.0 .. 86_400.0;

   function CLOCK return TIME;

   function YEAR (DATE: TIME) return YEAR_NUMBER;
   function MONTH (DATE: TIME) return MONTH_NUMBER;
   function DAY (DATE: TIME) return DAY_NUMBER;
   function SECONDS (DATE: TIME) return DAY_DURATION;

   procedure SPLIT(DATE : in TIME;
                   YEAR : out YEAR_NUMBER;
                   MONTH : out MONTH_NUMBER;
                   DAY : out DAY_NUMBER;
                   SECONDS : out DAY_DURATION);

   function TIME_OF(YEAR : YEAR_NUMBER;
                   MONTH : MONTH_NUMBER;
                   DAY : DAY_NUMBER;
                   SECONDS : DAY_DURATION := 0.0) return TIME;

   function "+" (LEFT : TIME; RIGHT : DURATION) return TIME;
   function "+" (LEFT : DURATION; RIGHT : TIME) return TIME;
   function "-" (LEFT : TIME; RIGHT : DURATION) return TIME;
   function "-" (LEFT : TIME; RIGHT : TIME) return DURATION;

   function "<" (LEFT, RIGHT : TIME) return BOOLEAN;
   function "<=" (LEFT, RIGHT : TIME) return BOOLEAN;
   function ">" (LEFT, RIGHT : TIME) return BOOLEAN;
   function ">=" (LEFT, RIGHT : TIME) return BOOLEAN;

   TIME_ERROR : exception;

private
```

```
   -- implementációfüggő
end CALENDAR;
```

<abort statement> ::= **abort** *<task name>* { , *<task name>*} ;

Az **abort** `T1,T2;` utasítás *abnormal* állapotba hozza a megnevezett taskokat. Ebben az állapotban a task randevúban már nem vehet részt. Ha az abortált task **accept**, **select**, **delay** utasításnál, **entry** hívásnál (ahol a randevú még nem jött létre) tart, akkor a task komplett állapotba kerül.

12.7. A select utasítás

<select statement> ::= *<selective wait>*|*<conditional entry call>*
|*<timed entry call>*

A **select** utasítás kombinálva az **accept**, **delay** és **terminate** utasításokkal, különféle helyezetek megoldására alkalmazható. Három ilyen a szelektív várakoztatás, a feltételes **entry** utasítás és az időhöz kötött **entry** hívás.

12.7.1. A szelektív várakoztatás

<selective wait> ::= **select** *<select alternative>*
{ **or** *<select alternative>*}
[**else** *<sequence of statements>*]
end select ;
<select alternative> ::= [**when** *<condition>* =>] *<selective wait alternative>*
<selective wait alternative> ::= *<accept alternative>*
|*<delay alternative>*|*<terminate alternative>*
<accept alternative> ::= *<accept statement>* [*<sequence of statements>*]
<delay alternative> ::= *<delay statement>* [*<sequence of statements>*]
<terminate alternative> ::= **terminate** ;

A **select** utasításnak ez a formája lehetővé teszi valamely feltételek egyikének teljesüléséig a várakozást, majd egy feltétel teljesülésekor a hozzá tartozó utasítások végrehajtását.

A szelektív várakoztatásnak legalább egy olyan ága kell legyen, ami **accept** utasítást tartalmaz, ezen kívül lehet olyan ága, ami **delay** utasítást, és olyan, ami **terminate** utasítást tartalmaz.

Egy ágat *nyitott*nak nevezünk, ha nincs benne feltétel (**when**), vagy ha a feltétel értéke igaz, különben *zárt*nak nevezzük.

A **select** utasítás végrehajtásakor a feltételek kiértékelésre kerülnek, és meghatározásra kerül, mely ágak nyitottak. Ezután végrehajtódik egy nyitott ág, vagy ha ilyen nincs, az **else** - ág. Ezzel az utasítás végrehajtása befe-

jeződik. Ha több nyitott ág is van, a választás a következő szabályok szerint történik :

a) ha van nyitott **accept** ág, amelynél randevú jön létre, az hajtódik végre;
b) ha nincs nyitott **accept** ág, akkor egy nyitott **delay** ág választódhat ki;
c) az **else** - ág csak akkor választódhat ki, ha mindegyik ág zárt;
d) a **terminate** ág nem választódhat ki addig, amíg van a tasknak nem üres **entry**-sora.

Példa a szelektív várakoztatásra:

```
task body RESOURCE is
   BUSY : BOOLEAN := FALSE;
begin
   loop
      select
         when not BUSY =>
            accept SEIZE do BUSY := TRUE; end;
      or
         accept RELEASE do BUSY := FALSE; end;
      or
         terminate;
      end select;
   end loop;
end RESOURCE;
```

12.7.2. A feltételes entry hívás

<conditional entry call> ::= **select** *<entry call statement>*
[<sequence of statements>]
else *<sequence of statements>*
end select ;

A feltételes **entry** hívás csak akkor hoz létre **entry** hívást, ha a randevú azonnal létre is jön. Nem történik meg az **entry** hívás, ha a randevú nem jöhet azonnal létre (pl. mert a task végrehajtása még nem tart az **accept** utasításnál, vagy az **accept** ága nem nyitott, vagy az illető **entry**-hez tartozó várakozási sor nem üres).

```
procedure SPIN(R:in RESOURCE) is
begin
   loop
      select
         R.SEIZE;
         return;
      else
         null; -- várakozás BUSY = TRUE -ra
      end select;
   end loop;
end SPIN;
```

Ha a hívás nem történik meg, helyette az **else** ág hajtódik végre.

12.7.3. Időhöz kötött entry hívás

<timed entry call> ::= **select** *<entry call statement>*
[<sequence of statements>]
or *<delay alternative>*
end select ;

Az időhöz kötött **entry** hívásnál a hívás akkor következik be, ha a randevú még az utasításban megadott idő lejárta előtt létrejön. Ha az adott idő alatt a randevú nem jön létre, a **delay** - ág hajtódik végre.

```
select
   CONTROLLER.REQUEST(MEDIUM)(SOME_ITEM);
or
   delay 45.0;
end select;
```

12.8. Példaprogramok

i) Kölcsönös kizárás

Párhuzamos rendszerekben szükség van olyan eszközre, ami biztosítja, hogy amikor egy közösen használt objektumhoz egy task "hozzányúl", arra az időre a többi task ne férhessen hozzá. Az Adában a kölcsönös kizárást többféleképpen, és nagyon egyszerűen meg lehet valósítani.

Először az osztott objektumot magában a taskban helyezzük el. Itt `T` egy korábban definiált típus, `COMMON` pedig az osztott objektum neve.

```
task SHARED_DATA is
   entry UPDATE(X:in T);
   entry READ(X:out T);
end SHARED_DATA;

task body SHARED_DATA is
   COMMON : T;
begin
   loop
      select
         accept UPDATE(X:in T) do
            COMMON := X;
         end UPDATE;
      or
         accept READ(X:out T) do
            X := COMMON;
         end READ;
      or
         terminate;
      end select;
   end loop;
end SHARED_DATA;
```

Az osztott objektum lehet egy globális objektum is, ilyenkor a kölcsönös kizárást szemafor használatával valósíthatjuk meg. Egy szemaforhoz egy P és egy V művelet tartozik. A P operáció csak az első őt meghívó taskot endegi tovább, a többit megvárakoztatja, a V operáció pedig mindig továbbindít egy a P által felfüggesztett taskot.

```
task type SEMAPHORE is
  entry P;
  entry V;
end SEMAPHORE;

task body SEMAPHORE is
begin
  loop
    accept P;
    accept V;
  end loop;
end SEMAPHORE;
```

Az osztott objektumhoz való fordulás előtt tehát mindig meg kell hívni a P operációt, amikor pedig befejeződtek az osztott objektumon végzett műveletek, akkor meg kell hívni a V operációt, ezzel az objektum használatára vonatkozóan a kölcsönös kizárást biztosítottuk.

```
COMMON : T;
S : SEMAPHORE;
begin
...
S.P;
COMMON := F(COMMON);
S.V;
...
```

A szemafor egy nem túlságosan megbízható szinkronizációs technika, ha ugyanis valahol kifelejtjük a V operáció meghívását, ez teljesen összezavarja a program működését.

ii) Termelő-fogyasztó feladat

A termelő-fogyasztó feladatokban kétfajta task szerepel: az egyik, ami bizonyos típusú adatokat előállít, a másik pedig az, amelyik feldolgozza ezeket.

```
with SHARED_QUEUE;
package SHARED_INTEGER_QUEUE is
  new SHARED_QUEUE(INTEGER, 128);

task body PRODUCER is
  C : INTEGER;
begin
  loop
    -- C előállítása
```

```
      SHARED_INTEGER_QUEUE.QUEUE.PUT(C);
    end loop;
end PRODUCER;

task body CONSUMER is
    C : INTEGER;
begin
    loop
      SHARED_INTEGER_QUEUE.QUEUE.GET(C);
      -- C felhasználása
    end loop;
end CONSUMER;
```

A még fel nem dolgozott adatokat egy sorban kell tárolni, s ehhez a sorhoz mind a két fajta tasknak hozzá kell tudni férni, sőt, a hozzáférésre biztosítani kell a kölcsönös kizárást. Egy ilyen sor kezelését mutatjuk meg az alábbi generic-ben.

```
generic
    type ITEM is private;
    QUEUE_LENGTH : NATURAL;
package SHARED_QUEUE is
    task QUEUE is
      entry GET (D:out ITEM);
      entry PUT (D:in ITEM);
    end QUEUE;
end SHARED_QUEUE;

package body SHARED_QUEUE is
    task body QUEUE is
      type BUFFER_POSSIBLE is
        new NATURAL range 0 .. QUEUE_LENGTH;
      subtype BUFFER_ACTUAL is
        BUFFER_POSSIBLE range 1 .. BUFFER_POSSIBLE'LAST;
      BUFFER : array (BUFFER_ACTUAL) of ITEM;
      CONTAINS : BUFFER_POSSIBLE := 0;
      INX, OUTX : BUFFER_ACTUAL := 1;
    begin
      loop
        select
          when CONTAINS > 0 =>
            accept GET (D:out ITEM) do
              D := BUFFER(OUTX);
            end GET;
            OUTX := OUTX mod BUFFER_POSSIBLE'LAST + 1;
            CONTAINS := CONTAINS - 1;
        or
          when CONTAINS < BUFFER_POSSIBLE'LAST =>
            accept PUT (D:in ITEM) do
              BUFFER(INX) := D;
            end PUT;
            INX := INX mod BUFFER_POSSIBLE'LAST + 1;
            CONTAINS := CONTAINS + 1;
        or
          terminate;
        end select;
```

```
    end loop;
  end QUEUE;
end SHARED_QUEUE;
```

iii) Osztott halmaz megvalósítása

A pacakege-kről szóló fejezet végén közöltünk egy példát halmaz megvalósítására. Ez volt az ORDERED_SET package. Ebben a pontban elkészítjük ugyanazt a package-t, de úgy, hogy a halmaz osztott, több task által is használható legyen.

```
type ID is range 1 .. 128;

package ORDERED_SET is
  EMPTY_SET : exception;
  procedure INSERT(JOB:in ID; T:in DURATION);
  procedure SMALLEST(JOB:out ID);
end ORDERED_SET;

package body ORDERED_SET is
  task SET_MANAGER is
    entry INSERT (JOB:in ID; T:in DURATION);
    entry SMALLEST (JOB:out ID; PRESENT: out BOOLEAN);
  end SET_MANAGER;

  task body SET_MANAGER is
    IN_SET : array (ID) of BOOLEAN := (ID => FALSE);
    RANK : array (ID) of DURATION;
  begin
    loop
      select
        accept INSERT (JOB:in ID; T:in DURATION) do
          RANK(JOB) := T;
          IN_SET(JOB) := TRUE;
        end INSERT;
      or
        accept SMALLEST (JOB:out ID; PRESENT: out BOOLEAN) do
          declare
            T : DURATION := DURATION'LAST;
            SMALL : ID;
            FOUND : BOOLEAN := FALSE;
          begin
            for I in ID loop
              if IN_SET(I) and then RANK(I) <= T then
                SMALL := I; T := RANK(I); FOUND := TRUE;
              end if;
            end loop;
            if FOUND then
              IN_SET(SMALL) := FALSE;
              JOB := SMALL;
              PRESENT := TRUE;
            else
              PRESENT := FALSE;
            end if;
          end;
```

```
            end SMALLEST;
          or
            terminate;
          end select;
        end loop;
      end SET_MANAGER;
      procedure INSERT(JOB:in ID; T:in DURATION) is
      begin
        SET_MANAGER.INSERT(JOB, T);
      end INSERT;
      procedure SMALLEST(JOB:out ID) is
        PRESENT : BOOLEAN;
      begin
        SET_MANAGER.SMALLEST(JOB, PRESENT);
        if not PRESENT then raise EMPTY_SET; end if;
      end SMALLEST;
    end ORDERED_SET;
```

iv) Egy nagyon egyszerű terminál emulátor

Ebben a példában a taskok használatával, pontosabban egy-egy task interrupt rutinként való alkalmazásával foglalkozunk. Ha ez a program pl. egy IBM PC -n futna, s annak a soros vonalára (RS232) egy másik gép (pl. VAX) terminálvonala volna csatlakoztatva, akkor a program a klaviatúrán begépelt parancsokat továbbítaná a másik gépnek, s megjelenítené az arról jövő információt; ezáltal a PC eljátszhatná egy terminál szerepét. (Az, hogy ez milyen típusú terminál, tehát pl. VT52, VT100 vagy VT320, attól függ, hogy a programunk milyen escape-szekvenciákat ismer fel és hajt végre. A feladatnak ezzel a részével nem foglalkozunk.)

```
with TEXT_IO, KEYBOARD, RS232;
procedure TE is
    C : CHARACTER;
begin
    while TRUE loop
        if KEYBOARD.PRESSED then
            RS232.PUT(KEYBOARD.GET);
        end if;
        if RS232.RECEIVED then
            C := RS232.GET;
            case C is
            when ASCII.ETX =>
                exit;
            when ASCII.ESC =>
                ... az escape szekvenciák feldolgozása
            when others =>
                TEXT_IO.PUT(RS232.GET);
            end case;
        end if;
    end loop;
end TE;
```

```
package KEYBOARD is
    function PRESSED return BOOLEAN;
    function GET return CHARACTER;
end KEYBOARD;

with TEXT_IO;
package body KEYBOARD is
    DATA : CHARACTER;
    for DATA use at 16#60#;

    BUFFER_LEN : constant := 256;
    BUFFER : array (1 .. BUFFER_LEN) of CHARACTER;
    INX, OUTX : INTEGER := 1;
    COUNT : INTEGER := 0;

    task KEYBOARD_INPUT is
        entry INPUT;
        for INPUT use at 16#09#;
    end KEYBOARD_INPUT;
    task body KEYBOARD_INPUT is
    begin
        loop
            accept INPUT;
            if COUNT < BUFFER_LEN then
                BUFFER(INX) := DATA;
                INX := INX mod BUFFER_LEN + 1;
                COUNT := COUNT + 1;
            else
                TEXT_IO.PUT(ASCII.BEL);
            end if;
        end loop;
    end KEYBOARD_INPUT;

    function PRESSED return BOOLEAN is
    begin
        return COUNT > 0;
    end PRESSED;

    function GET return CHARACTER is
        CH : CHARACTER;
    begin -- precondition: COUNT > 0
        CH := BUFFER(OUTX);
        OUTX := OUTX mod BUFFER_LEN + 1;
        COUNT := COUNT - 1;
        return CH;
    end GET;

end KEYBOARD;

package RS232 is
    function RECEIVED return BOOLEAN;
    procedure PUT(CH:in CHARACTER);
    function GET return CHARACTER;
end RS232;

with TEXT_IO;
package body RS232 is
    type STATUS_REGISTER is array (0 .. 15) of BOOLEAN;

    LINE_STATUS : STATUS_REGISTER;
```

```
        for LINE_STATUS use at 16#3FD#;
        DATA : CHARACTER;
        for DATA use at 16#3F8#;
        BUFFER_LEN : constant := 256;
        BUFFER : array (1 .. BUFFER_LEN) of CHARACTER;
        INX, OUTX : INTEGER := 1;
        COUNT : INTEGER := 0;
        ENABLED : BOOLEAN := TRUE;
        task RS232_INPUT is
            entry INPUT;
            for INPUT use at 16#30#;
        end RS232_INPUT;
        task body RS232_INPUT is
        begin
            loop
                accept INPUT;
                if COUNT < BUFFER_LEN then
                    BUFFER(INX) := DATA;
                    INX := INX mod BUFFER_LEN + 1;
                    COUNT := COUNT + 1;
                end if;
            end loop;
        end RS232_INPUT;
        function RECEIVED return BOOLEAN is
        begin
            return COUNT > 0;
        end RECEIVED;
        procedure PUT(CH:in CHARACTER) is
        begin
            DATA := CH;
            loop
                exit when LINE_STATUS(9);
            end loop;
        end PUT;
        function GET return CHARACTER is
            CH : CHARACTER;
        begin -- precondition: COUNT > 0
            if not ENABLED and then COUNT < 30 then
                ENABLED := TRUE; TEXT_IO.PUT(ASCII.DC1);
            end if;
            CH := BUFFER(OUTX);
            OUTX := OUTX mod BUFFER_LEN + 1;
            COUNT := COUNT - 1;
            if ENABLED and then COUNT > 200 then
                ENABLED := FALSE; TEXT_IO.PUT(ASCII.DC3);
            end if;
            return CH;
        end GET;
    end RS232;
```

v) Forgalmi lámpák irányítása

Egy kereszteződés forgalmi lámpáinak vezérlését végző programot készítünk. A kereszteződésben egy főút és egy mellékút metszi egymást. Általában a főúton szabad a jelzés, s a főút forgalmát csak akkor állítjuk meg, ha a mellékútba beépített érzékelő azt jelzi, hogy a mellékúton a kereszteződéshez gépkocsi érkezett, vagy ha egy gyalogos azt jelzi, hogy a főúton levő zebrán át akar haladni. Az érzékelő és a gyalogos nyomógombja egyaránt a 8#2000# címen okoznak interruptot. A főút forgalmának megállításakor be kell tartani a következőket:

- A főút forgalmát csak akkor lehet megállítani, ha legalább 3 percen keresztül a zöld jelzés volt érvényben.
- A főút forgalmát legfeljebb fél percre szabad leállítani.

A forgalmi lámpák kapcsolására a TRAFFIC_LIGHT package-t használhatjuk:

```
package TRAFFIC_LIGHT is
  procedure START_LIGHT;
      -- a lámpák bekapcsolása; a főút lámpája zöld,
      -- a mellékúté piros
  procedure CHANGE_LIGHT;
      -- ellenkezőre állítja a kereszteződésben
      -- a lámpákat
end TRAFFIC_LIGHT;
```

A főprogram:

```
with TRAFFIC_LIGHT; use TRAFFIC_LIGHT;
procedure CONTROL_TRAFFIC_LIGHT is
  task LIGHT is
    entry STOP_MAIN;
    for STOP_MAIN use at 8#2000#;
  end LIGHT;
  task body LIGHT is
    CUT_OFF : constant DURATION := 180.0;
    SIDE_ROAD_OPEN : constant DURATION := 30.0;
  begin
    START_LIGHT;
    loop
      delay CUT_OFF;
      accept STOP_MAIN; CHANGE_LIGHT;
      delay SIDE_ROAD_OPEN;
      for I in 1 .. STOP_MAIN'COUNT loop
        accept STOP_MAIN;
      end loop;
      CHANGE_LIGHT;
    end loop;
  end LIGHT;
begin
  null;
end CONTROL_TRAFFIC_LIGHT;
```

vi) Prímszámok előállítása

Prímszámok előállításának egy jól ismert módszere az Eratoszthenesz-féle szitamódszer. A taskok felhasználásával egy érdekes program készíthető az alábbi alapgondolat alapján: a természetes számokat sorba állítjuk, majd az első prímszámmal elindítunk egy szűrő taskot. A szűrő task az első kapott számot kiírja, hiszen ez biztosan prím, elindít egy újabb szűrő taskot, s a számokat — kivéve az első szám többszöröseit — ennek átadja. [7]

```
with TEXT_IO; use TEXT_IO;
procedure SIEVE is
  package INT_IO is new INTEGER_IO(INTEGER); use INT_IO;
  -- First define the filter type.
  task type FILTER is
    entry NUMBER_STREAM(NUMBER:in INTEGER);
  end FILTER;
  type FILTER_TASK is access FILTER; -- pointer to a filter task
  FIRST_FILTER : FILTER_TASK;
  NUMBERS : INTEGER;
  function NEW_FILTER return FILTER_TASK is
  begin
    return new FILTER;
  end NEW_FILTER;
  task body FILTER is
    NEXT_FILTER : FILTER_TASK;
    PRIME, NUM : INTEGER;
  begin
    accept NUMBER_STREAM(NUMBER:in INTEGER) do
      PRIME := NUMBER;
    end NUMBER_STREAM;
    if PRIME > -1 then -- "-1" is an end-of-numbers flag.
      PUT(PRIME); NEW_LINE;
      NEXT_FILTER := NEW_FILTER; -- Spawn a new filter.
      loop
        accept NUMBER_STREAM(NUMBER:in INTEGER) do
          NUM := NUMBER;
        end NUMBER_STREAM;
        if NUM = -1 then
          NEXT_FILTER.NUMBER_STREAM(NUM);
          exit;
        end if;
        if NUM mod PRIME /= 0 then
          -- Pass the number on.
          NEXT_FILTER.NUMBER_STREAM(NUM);
        end if;
      end loop;
    end if;
  end FILTER;
begin -- Main program. Generate the initial list of integers.
  PUT("Enter end of integer range: ");
  GET(NUMBERS);
  FIRST_FILTER := new FILTER;
  PUT_LINE("All primes in the range are: ");
  for I in 2 .. NUMBERS loop
    FIRST_FILTER.NUMBER_STREAM(I);
  end loop;
```

```
   -- Flag the end-of-numbers.
   FIRST_FILTER.NUMBER_STREAM(-1);
end SIEVE;
```

13. Reprezentációs specifikációk

Egy software-készítésre is alkalmas magasszintű nyelvnek különböző szintű eszközökre van szüksége az adatmanipulációk terén (ráadásul ezeknek az eszközöknek élesen el kell határolódniuk):
- rendelkeznie kell eszközökkel az absztrakt típusok kezelésére, ahol az adatoknak csak a logikai tulajdonságai fontosak, és
- rendelkeznie kell olyan eszközökkel is, amelyek az adatok fizikai ábrázolásának szintjén használhatók fel.

Az eddigiekben az adatkezelésnek a logikai szintjével foglalkoztunk, ebben a pontban az alacsonyszintű, gépközeli eszközöket tárgyaljuk.

<representation clause> ::= <type representation clause>|<address clause>
<type representation clause> ::= <length clause>
|<enumeration representation clause>
|<record representation clause>
<length clause> ::= **for** *<attribute>* **use** *<simple expression>* ;

A típusokhoz megadhatjuk, hogy a típus egy értékének ábrázolásához a fordítóprogram mennyi helyet (hány bitet) használhat fel (ez nyilvánvalóan azt is meghatározza, hogy az értékek eléréséhez milyen kódot kell generálnia). Példák:

```
type MEDIUM is range 0 .. 65000;
for MEDIUM'SIZE use 16;
type COLOR is (WHITE, RED, YELLOW, GREEN, BLUE, BROWN, BLACK);
for COLOR'SIZE use 8;
```

<enumeration representation clause> ::= **for** *<type simple name>*
use *<aggregate>* ;

A felsorolási típusok értékeit alapértelmezésben a fordítóprogram a 0, 1, 2, ... számokkal kódolja. Előírható, hogy ez ne így történjen:

```
type MIX_CODE is (ADD, SUB, MUL, LDA);
for MIX_CODE use (ADD => 1, SUB => 2, MUL => 3, LDA => 8);
```

<record representation clause> ::= **for** *<type simple name>* **use record**
[<alignment clause>]
{<component clause>}
end record ;
<alignment clause> ::= **at mod** *<static simple expression>* ;
<component clause> ::= <component name> **at** *<static simple expression>*
range *<static range>* ;

Rekord típus reprezentációjára előírhatjuk, hogy az objektumok milyen paritású, esetleg 4-gyel vagy 8-cal osztható címen kezdődjenek, ill. hogy a komponensek a rekordon belül hányadik szóban és mely biteken helyezkedjenek el.

```
WORD : constant := 4;
type STATE is (A, M, W, P);
type MODE is (FIX, DEC, EXP, SIGNIF);
type BYTE_MASK is array (0 .. 7) of BOOLEAN;
type STATE_MASK is array (STATE) of BOOLEAN;
type MODE_MASK is array (MODE) of BOOLEAN;
type PROGRAM_STATUS_WORD is
  record
    SYSTEM_MASK      : BYTE_MASK;
    PROTECTION_KEY   : INTEGER range 0 .. 3;
    MACHINE_STATE    : STATE_ MASK;
    INTERRUPT_CAUSE  : INTERRUPTION_CODE;
    ILC              : INTEGER range 0 .. 3;
    CC               : INTEGER range 0 .. 3;
    PROGRAM_MASK     : MODE_MASK;
    INST_ADDRESS     : ADDRESS;
  end record;
for PROGRAM_STATUS_WORD use
  record at mod 8;
    SYSTEM_MASK      at 0*WORD range 0 .. 7;
    PROTECTION_KEY   at 0*WORD range 10 .. 11;
    MACHINE_STATE    at 0*WORD range 12 .. 15;
    INTERRUPT_CAUSE  at 0*WORD range 16 .. 31;
    ILC              at 1*WORD range 0 .. 1;
    CC               at 1*WORD range 2 .. 3;
    PROGRAM_MASK     at 1*WORD range 4 .. 7;
    INST_ADDRESS     at 1*WORD range 8 .. 31;
  end record;
```

<address clause> ::= **for** *<simple name>* **use at** *<simple expression>* ;

Alapértelmezésben a fordító- és a szerkesztőprogramok maguk határozzák meg a program objektumainak a címeit, az Ada esetében azonban lehetőségünk van ennek felülbírálására, s a programban előírhatjuk, hogy egy változó vagy egy alprogram a memória mely címére képződjön le.

```
CONTROL : INTEGER;
for CONTROL use at 16#0020#;   -- a CONTROL változó a memória
                               -- 16#0020#-as címének szimbolikus
                               -- neve lesz
task INTERRUPT_HANDLER is
  entry DONE;
  for DONE use at 16#0040#;    -- a DONE entry meghívása
                               -- azt a runtint hívja meg, amelynek címe
                               -- a 16#0040# címen található
end INTERRUPT_HANDLER;
```

14. Fordítási direktívák

A reprezentációs specifikációkon kívül a fordítást befolyásolhatjuk még fordítási direktívákkal, ún. pragmákkal is. A pragmák száma az egyes Ada implementációkban eltérhet. Itt csak néhány általánosan használt pragmát sorolunk fel.

<pragma> ::= **pragma** *<identifier>*
    [(*<argument association>* { , *<argument association>*})] ;
<argument association> ::= [*<argument identifier>* => *<name>*
    | [*<argument identifier>* =>] *<expression>*

pragma ELABORATE(*azonosító, ...*); — A pragma paraméterlistáján valamely, az adott egységben használt könyvtári egységek nevei lehetnek. A pragmát a **with** és **use** utasítások és az egység specifikációs része között kell elhelyezni. Az ELABORATE pragma azt írja elő, hogy a megnevezett package-eket a program végrehajtásának kezdetén előbb kell kiértékelni (a törzs inicializáló részét előbb kell végrehajtani), mint a pragmát tartalmazó egységet.

pragma INLINE(*azonosító, ...*); — A pragma paraméterlistáján alprogramnevek szerepelnek. A pragmát abban a deklarációs részben kell elhelyezni, ahol a listán szereplő alprogramok specifikációs részei vannak. Az INLINE pragma azt írja elő, hogy az alprogramtörzseket a hívás helyére makroszerűen be kell másolni. Az INLINE pragmának a fordítási sorrendre is van hatása. Ha egy egységben INLINE alprogramok vannak, akkor az egységre hivatkozó egységeket csak azután lehet fordítani, hogy az alprogramok törzsét tartalmazó egységet is lefordítottuk.

pragma INTERFACE(*nyelv megnevezése, alprogram neve*); — A pragmát a megnevezett alprogram specifikációja után kell elhelyezni. Az INTERFACE pragma arra figyelmezteti a fordítóprogramot, hogy az alprogramtörzset nem Adában készítették el, s a hívásokat a jelzett nyelvnek megfelelően kell generálni.

pragma OPTIMIZE(*azonosító*); — A pragma argumentume a TIME vagy a SPACE szó lehet. A pragmát deklarációs részben kell elhelyezni, s a deklarációs részhez tartozó blokkra vagy törzsre vonatkozik. Előírja, hogy az idő vagy a hely legyen-e az elsődleges optimalizálási kritérium.

pragma SUPPRESS(*azonosító*); — A pragma paraméterlistáján egy objektum, egy típus vagy egy **exception** neve lehet, ill. a pragma használható üres paraméterlistával is. A pragmát egy deklarációs részben kell elhelyezni, s az egész törzsre (vagy blokkra) érvényes. Azt jelenti, hogy a fordítóprogramnak a megnevezett objektumra, típusra (pontosabban az ilyen típusú összes objektumra) és **exception**-ra vonatkozó ellenőrzéseket a törzsből ki kell hagynia. A paraméter nélküli SUPPRESS azt jelenti, hogy a törzs fordításakor az összes ellenőrzést el kell hagyni.

15. Input - output

Az Adában, a modern gyakorlatnak megfelelően, a file-típusok műveletei nem a nyelv utasításai, hanem predefinit package-k alprogramjai. Ezek a `SEQUENTIAL_IO`, `DIRECT_IO`, `TEXT_IO`, `IO_EXCEPTIONS` és a `LOW_LEVEL_IO` package. Az Ada definíciója tartalmazza ezeknek a package-knek a specifikációs részét, s leírja az alprogramok szemantikáját. Ez a megoldás biztosítja az I/O egységességét és módosíthatóságát, mert így az I/O implementációját könnyű hozzáigazítani az illető operációs rendszerhez.

Minden I/O egy file-lal kapcsolatos, az I/O utasítások a file típusok típusműveletei. A file-ok maguk perifériákon helyezkednek el. Egy file kezeléséhez definiálnunk kell egy *file-változót*, ami a feldolgozás pillanatnyi állapotát írja le. A file-változó bármely —megfelelő típusú— külső file-lal összekapcsolható. A file kezelése tehát nem függ a berendezéstől, csak a file típusától. (Egy text-filet ugyanúgy kell kezelni, függetlenül attól, hogy az sornyomtatón, vagy lemezen van.)

Az Adában használható file-típusokat az említett package-ek **limited private** típusként definiálják. A file kezelése a következő lépésekben történik :
- file-változó deklarálása,
- a file-változó összekapcsolása egy külső file-lal (file-nyitás),
- az olvasó ill. író műveletek végrehajtása,
- a file lezárása.

Az I/O package-k egységesek az alprogramok neveinek megválasztásában. Egy file létrehozása (`CREATE`), egy létező külső file megnyitása (`OPEN`), egy file lezárása (`CLOSE`) ugyanolyan eljárással történik, bármelyik package-t használjuk is.

A file-ok kezelésekor bekövetkező hibákat az Ada egy külön package-ben, az `IO_EXCEPTIONS` package-ben definiálja.

```
package IO_EXCEPTIONS is
   STATUS_ERROR  : exception;
   MODE_ERROR    : exception;
   NAME_ERROR    : exception;
   USE_ERROR     : exception;
   DEVICE_ERROR  : exception;
   END_ERROR     : exception;
   DATA_ERROR    : exception;
   LAYOUT_ERROR  : exception;
end IO_EXCEPTIONS;
```

15.1. Szekvenciális file-ok kezelése

A szekvenciális file azonos típusú rekordok sorozata. Az egyes rekordokhoz ún. file-pozíciót rendelünk: a file első rekordjához az 1 -et, a másodikhoz a 2 -t, ..., az utolsó rekord utáni pozíció neve *END_OF_FILE*. A file-nyitás a file-pozíciót 1 -re állítja. A SEQUENTIAL_IO package generic package, amiből a rekordtípus megadásával lehet példányosítani.

```
with IO_EXCEPTIONS;
generic
   type ELEMENT_TYPE is private;
package SEQUENTIAL_IO is

   type FILE_TYPE is limited private;

   type FILE_MODE is (IN_FILE, OUT_FILE);

   -- File management

   procedure CREATE(FILE : in out FILE_TYPE;
                    MODE : in FILE_MODE := OUT_FILE;
                    NAME : in STRING := "";
                    FORM : in STRING := "");

   procedure OPEN(FILE : in out FILE_TYPE;
                  MODE : in FILE_MODE;
                  NAME : in STRING;
                  FORM : in STRING := "");

   procedure CLOSE(FILE : in out FILE_TYPE);
   procedure DELETE(FILE : in out FILE_TYPE);
   procedure RESET(FILE : in out FILE_TYPE; MODE : in FILE_MODE);
   procedure RESET(FILE : in out FILE_TYPE);

   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;

   -- Input and output operations

   procedure READ(FILE : in FILE_TYPE; ITEM : out ELEMENT_TYPE);
   procedure WRITE(FILE : in FILE_TYPE; ITEM : in ELEMENT_TYPE);

   function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

   -- Exceptions

   STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
   MODE_ERROR : exception renames IO_EXCEPTIONS.MODE_ERROR;
   NAME_ERROR : exception renames IO_EXCEPTIONS.NAME_ERROR;
   USE_ERROR : exception renames IO_EXCEPTIONS.USE_ERROR;
   DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
   END_ERROR : exception renames IO_EXCEPTIONS.END_ERROR;
   DATA_ERROR : exception renames IO_EXCEPTIONS.DATA_ERROR;

private

   -- implementációfüggő
```

```
end SEQUENTIAL_IO;
```

A rekordok olvasása a READ eljárással történik. Az *END_OF_FILE* pozíció után való olvasás END_ERROR -t vált ki. Az *END_OF_FILE* pozíció felismerhető az END_OF_FILE függvénnyel (a függvény akkor ad TRUE értéket, ha az aktuális file-pozíció az *END_OF_FILE* pozíció).

A READ és WRITE eljárások értelemszerűen változtatják (1-el növelik) a file-pozíciót.

15.2. Direkt file-ok kezelése

A direkt file-kezelés csak abban különbözik a szekvenciálistól, hogy a file-pozíció állítására az Ada külön bevezet egy eljárást. Az írás és az olvasás előtt így most pozicionálhatunk a kívánt rekordra. Az írás és az olvasás 1-el növeli a file-pozíciót.

```
with IO_EXCEPTIONS;
generic
   type ELEMENT_TYPE is private;
package DIRECT_IO is

   type FILE_TYPE is limited private;

   type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);
   type COUNT is range 0 .. implementációfüggő;
   subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;

   -- File management

   procedure CREATE(FILE : in out FILE_TYPE;
                    MODE : in FILE_MODE := INOUT_FILE;
                    NAME : in STRING := "";
                    FORM : in STRING := "");

   procedure OPEN(FILE : in out FILE_TYPE;
                  MODE : in FILE_MODE;
                  NAME : in STRING;
                  FORM : in STRING := "");

   procedure CLOSE(FILE : in out FILE_TYPE);
   procedure DELETE(FILE : in out FILE_TYPE);
   procedure RESET(FILE : in out FILE_TYPE; MODE : in FILE_MODE);
   procedure RESET(FILE : in out FILE_TYPE);

   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;

   -- Input and output operations

   procedure READ(FILE : in FILE_TYPE;
                  ITEM : out ELEMENT_TYPE;
                  FROM : in POSITIVE_COUNT);
   procedure READ(FILE : in FILE_TYPE;
                  ITEM : out ELEMENT_TYPE);
```

```
   procedure WRITE(FILE : in FILE_TYPE;
                   ITEM : in ELEMENT_TYPE;
                   TO : in POSITIVE_COUNT);
   procedure WRITE(FILE : in FILE_TYPE;
                   ITEM : in ELEMENT_TYPE);
   procedure SET_INDEX(FILE : in FILE_TYPE;
                   TO : in POSITIVE_COUNT);
   function INDEX(FILE : in FILE_TYPE) return POSITIVE_COUNT;
   function SIZE(FILE : in FILE_TYPE) return COUNT;
   function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;
   -- Exceptions
   STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
   MODE_ERROR : exception renames IO_EXCEPTIONS.MODE_ERROR;
   NAME_ERROR : exception renames IO_EXCEPTIONS.NAME_ERROR;
   USE_ERROR : exception renames IO_EXCEPTIONS.USE_ERROR;
   DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
   END_ERROR : exception renames IO_EXCEPTIONS.END_ERROR;
   DATA_ERROR : exception renames IO_EXCEPTIONS.DATA_ERROR;
private
   -- implementációfüggő
end DIRECT_IO;
```

15.3. Textfile-ok

A text-file-ok kezelésére való eljárások a TEXT_IO package-ben vannak definiálva. A TEXT_IO package nem generic, de generic package-ket tartalmaz az egész, valós és felsorolási típusokra, valamint alprogramokat a CHARACTER, STRING és BOOLEAN típusra.

```
with IO_EXCEPTIONS;
package TEXT_IO is
   type FILE_TYPE is limited private;
   type FILE_MODE is (IN_FILE, OUT_FILE);
   type COUNT is range 0 .. implementációfüggő;
   subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;
   UNBOUNDED : constant COUNT := 0; -- line and page length
   subtype FIELD is INTEGER range 0 .. implementációfüggő;
   subtype NUMBER_BASE is INTEGER range 2 .. 16;
   type TYPE_SET is (LOWER_CASE, UPPER_CASE);
   -- File Management
   procedure CREATE(FILE : in out FILE_TYPE;
                   MODE : in FILE_MODE := OUT_FILE;
                   NAME : in STRING := "";
                   FORM : in STRING := "");
   procedure OPEN(FILE : in out FILE_TYPE;
                   MODE : in FILE_MODE;
```

```
                    NAME : in STRING;
                    FORM : in STRING := "");
procedure CLOSE(FILE : in out FILE_TYPE);
procedure DELETE(FILE : in out FILE_TYPE);
procedure RESET(FILE : in out FILE_TYPE; MODE : in FILE_MODE);
procedure RESET(FILE : in out FILE_TYPE);

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;

-- Control of default input and output files

procedure SET_INPUT(FILE : in FILE_TYPE);
procedure SET_OUTPUT(FILE : in FILE_TYPE);

function STANDARD_INPUT return FILE_TYPE;
function STANDARD_OUTPUT return FILE_TYPE;

function CURRENT_INPUT return FILE_TYPE;
function CURRENT_OUTPUT return FILE_TYPE;

-- Specification of line and page lengths

procedure SET_LINE_LENGTH(FILE : in FILE_TYPE; TO : in COUNT);
procedure SET_LINE_LENGTH(TO : in COUNT);

procedure SET_PAGE_LENGTH(FILE : in FILE_TYPE; TO : in COUNT);
procedure SET_PAGE_LENGTH(TO : in COUNT);

function LINE_LENGTH(FILE : in FILE_TYPE) return COUNT;
function LINE_LENGTH return COUNT;

function PAGE_LENGTH(FILE : in FILE_TYPE) return COUNT;
function PAGE_LENGTH return COUNT;

-- Column, Line, and Page Control

procedure NEW_LINE(FILE : in FILE_TYPE;
                   SPACING : in POSITIVE_COUNT := 1);
procedure NEW_LINE(SPACING : in POSITIVE_COUNT := 1);

procedure SKIP_LINE(FILE : in FILE_TYPE;
                   SPACING : in POSITIVE_COUNT := 1);
procedure SKIP_LINE(SPACING : in POSITIVE_COUNT := 1);

function END_OF_LINE(FILE : in FILE_TYPE) return BOOLEAN;
function END_OF_LINE return BOOLEAN;

procedure NEW_PAGE(FILE : in FILE_TYPE);
procedure NEW_PAGE;

procedure SKIP_PAGE(FILE: in FILE_TYPE);
procedure SKIP_PAGE;

function END_OF_PAGE(FILE : in FILE_TYPE) return BOOLEAN;
function END_OF_PAGE return BOOLEAN;

function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;
function END_OF_FILE return BOOLEAN;

procedure SET_COL(FILE : in FILE_TYPE; TO : in POSITIVE_COUNT);
procedure SET_COL(TO : in POSITIVE_COUNT);

procedure SET_LINE(FILE : in FILE_TYPE;
```

```
                    TO : in POSITIVE_COUNT);
  procedure SET_LINE(TO : in POSITIVE_COUNT);

  function COL(FILE : in FILE_TYPE) return POSITIVE_COUNT;
  function COL return POSITIVE_COUNT;

  function LINE(FILE : in FILE_TYPE) return POSITIVE_COUNT;
  function LINE return POSITIVE_COUNT;

  function PAGE(FILE : in FILE_TYPE) return POSITIVE_COUNT;
  function PAGE return POSITIVE_COUNT;

  -- Character Input-Output
  -- ld. 15.3.1.

  -- String Input-Output
  -- ld. 15.3.1.

  -- Generic package for Input-Output of Integer Types
  generic
    type NUM is range <>;
  package INTEGER_IO is
    -- ld. 15.3.2.
  end INTEGER_IO;

  -- Generic package for Input-Output of Real Types
  generic
    type NUM is digits <>;
  package FLOAT_IO is
    -- ld. 15.3.3.
  end FLOAT_IO;

  generic
    type NUM is delta <>;
  package FIXED_IO is
    -- ld. 15.3.3.
  end FIXED_IO;

  -- Generic package for Input-Output of Enumeration types
  generic
    type ENUM is (<>);
  package ENUMERATION_IO is
    -- ld. 15.3.4.
  end ENUMERATION_IO;

  -- Exceptions

  STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
  MODE_ERROR : exception renames IO_EXCEPTIONS.MODE_ERROR;
  NAME_ERROR : exception renames IO_EXCEPTIONS.NAME_ERROR;
  USE_ERROR : exception renames IO_EXCEPTIONS.USE_ERROR;
  DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
  END_ERROR : exception renames IO_EXCEPTIONS.END_ERROR;
  DATA_ERROR : exception renames IO_EXCEPTIONS.DATA_ERROR;
  LAYOUT_ERROR : exception renames IO_EXCEPTIONS.LAYOUT_ERROR;

private

  -- implementációfüggő

end TEXT_IO;
```

Az Adában léteznek feltételezett (*standard*) text-file-ok is, amelyeket a fut-

tató rendszer a program indulásának pillanatában automatikusan megnyit. Interaktív rendszerekben ezek a file-ok a terminálhoz vannak rendelve: a klaviatúra a standard input, a képernyő a standard output file. A standard file-ra vonatkozó GET és PUT eljárásokban file-változót nem kell megadni. A standard file átirányítható más text-file-ra is a SET_INPUT és SET_OUTPUT eljárásokkal.

Az összes olvasást, ami a text-file-okra vonatkozik, az átlapolt GET eljárással hajthatjuk végre.

A GET eljárás a BOOLEAN, egész, valós és felsorolási típusok esetében is ugyanúgy működik: az aktuális sorpozíciótól egy szintaktikus egységet leválaszt és belső formára konvertál, miközben a bevezető szóközöket, tabulátorkaraktereket és sorvégeket átlépi. Ha az adat formátuma nem felel meg a várt szintaxisnak, az DATA_ERROR hibát, ha az érték nem esik a várt értékhatárok közé, az CONSTRAINT_ERROR hibát okoz.

Minden text-outputot az átlapolt PUT eljárással lehet elvégezni. A text-file-ok sormérete lehet rögzített, vagy változó. Ha rögzített sorméret mellett a kiírandó string nem fér el az aktuális sorban, akkor a következő sor első pozíciójára kerül (ha a string nem fér el egy sorban, az LAYOUT_ERROR hibát okoz). Változó sorhossz esetén a string kiírása folytatólagosan az aktuális sorba történik.

A változó sorhosszúságú file-okban az aktuális sor befejezésére a NEW_LINE eljárás szolgál. A NEW_LINE(N); eljáráshívás a standard file-ban befejezi az aktuális sort, és még N-1 db üres sort készít.

Karakteres és string inputnál rögzített sorhossz esetén — a sor teljes beolvasása után — a GET eljárás automatikusan átlép a következő sorra, változó sorhossz esetén erre külön művelet van, a SKIP_LINE.

A SET_COL eljárás az aktuális sorban való pozicionálásra szolgál.

15.3.1. Karakteres és string I/O

```
with IO_EXCEPTIONS;
package TEXT_IO is
   ...
   -- Character Input-Output
   procedure GET(FILE : in FILE_TYPE; ITEM : out CHARACTER);
   procedure GET(ITEM : out CHARACTER);
   procedure PUT(FILE : in FILE_TYPE; ITEM : in CHARACTER);
   procedure PUT(ITEM : in CHARACTER);
   -- String Input-Output
   procedure GET(FILE : in FILE_TYPE; ITEM : out STRING);
   procedure GET(ITEM : out STRING);
   procedure PUT(FILE : in FILE_TYPE; ITEM : in STRING);
   procedure PUT(ITEM : in STRING);
```

```
  procedure GET_LINE(FILE : in FILE_TYPE;
                     ITEM : out STRING;
                     LAST : out NATURAL);
  procedure GET_LINE(ITEM : out STRING; LAST : out NATURAL);
  procedure PUT_LINE(FILE : in FILE_TYPE; ITEM : in STRING);
  procedure PUT_LINE(ITEM : in STRING);
  ...
end TEXT_IO;
```

A CHARACTER és STRING típusú értékek konvertálás nélkül kerülnek a file-ba. Pl. PUT('B'); vagy PUT("Ez egy string");. Példa egy file karakterenként való átmásolására:

```
with TEXT_IO; use TEXT_IO;
procedure COPY_TEXT is
  CH : CHARACTER;
begin
  while not END_OF_FILE loop
    if END_OF_LINE then
      SKIP_LINE; NEW_LINE;
    else
      GET(CH); PUT(CH);
    end if;
  end loop;
end COPY_TEXT;
```

A GET_LINE függvény az aktuális sorpozíciótól a sor végéig tartó rész összes karakterét megadja a stringváltozóban, és ezután áttér az új sorba. A PUT_LINE(S); eljáráshívás hatása megegyezik a következő program hatásával :

```
PUT(S); NEW_LINE;
```

15.3.2. Integer I/O

```
with IO_EXCEPTIONS;
package TEXT_IO is
  ...
  -- Generic package for Input-Output of Integer Types
  generic
    type NUM is range <>;
  package INTEGER_IO is
    DEFAULT_WIDTH : FIELD := NUM'WIDTH;
    DEFAULT_BASE : NUMBER_BASE := 10;
    procedure GET(FILE : in FILE_TYPE;
                  ITEM : out NUM;
                  WIDTH : in FIELD := 0);
    procedure GET(ITEM : out NUM; WIDTH : in FIELD := 0);
    procedure PUT(FILE : in FILE_TYPE;
                  ITEM : in NUM;
                  WIDTH : in FIELD := DEFAULT_WIDTH;
                  BASE : in NUMBER_BASE := DEFAULT_BASE);
```

```
    procedure PUT(ITEM : in NUM;
                  WIDTH : in FIELD := DEFAULT_WIDTH;
                  BASE : in NUMBER_BASE := DEFAULT_BASE);
    procedure GET(FROM : in STRING;
                  ITEM : out NUM;
                  LAST : out POSITIVE);
    procedure PUT(TO : out STRING;
                  ITEM : in NUM;
                  BASE : in NUMBER_BASE := DEFAULT_BASE);
  end INTEGER_IO;
  ...
end TEXT_IO;
```

Egész számok kiírására az `INTEGER_IO` generic package használható. Paramétere a `NUM` típus. Például a

```
package INT_IO is new TEXT_IO.INTEGER_IO(INTEGER);
```

példányosítás után a

```
PUT(128,6); PUT(-3,10); PUT(255,10,16);
```

program eredménye a következő:

```
␣␣␣128␣␣␣␣␣␣␣␣␣-3␣␣␣␣16#FF#
```

15.3.3. Float és fixed I/O

```
with IO_EXCEPTIONS;
package TEXT_IO is
  ...
  -- Generic package for Input-Output of Real Types
  generic
    type NUM is digits <>;
  package FLOAT_IO is
    DEFAULT_FORE : FIELD := 2;
    DEFAULT_AFT : FIELD := NUM'DIGITS-1;
    DEFAULT_EXP : FIELD := 3;
    procedure GET(FILE : in FILE_TYPE;
                  ITEM : out NUM;
                  WIDTH : in FIELD := 0);
    procedure GET(ITEM : out NUM; WIDTH : in FIELD := 0);
    procedure PUT(FILE : in FILE_TYPE;
                  ITEM : in NUM;
                  FORE : in FIELD := DEFAULT_FORE;
                  AFT : in FIELD := DEFAULT_AFT;
                  EXP : in FIELD := DEFAULT_EXP);
    procedure PUT(ITEM : in NUM;
                  FORE : in FIELD := DEFAULT_FORE;
                  AFT : in FIELD := DEFAULT_AFT;
                  EXP : in FIELD := DEFAULT_EXP);
```

```
    procedure GET(FROM : in STRING;
                  ITEM : out NUM;
                  LAST : out POSITIVE);
    procedure PUT(TO : out STRING;
                  ITEM : in NUM;
                  AFT : in FIELD := DEFAULT_AFT;
                  EXP : in INTEGER := DEFAULT_EXP);
  end FLOAT_IO;
  generic
    type NUM is delta <>;
  package FIXED_IO is
    DEFAULT_FORE : FIELD := NUM'FORE;
    DEFAULT_AFT : FIELD := NUM'AFT;
    DEFAULT_EXP : FIELD := 0;
    procedure GET(FILE : in FILE_TYPE;
                  ITEM : out NUM;
                  WIDTH : in FIELD := 0);
    procedure GET(ITEM : out NUM; WIDTH : in FIELD := 0);
    procedure PUT(FILE : in FILE_TYPE;
                  ITEM : in NUM;
                  FORE : in FIELD := DEFAULT_FORE;
                  AFT : in FIELD := DEFAULT_AFT;
                  EXP : in FIELD := DEFAULT_EXP);
    procedure PUT(ITEM : in NUM;
                  FORE : in FIELD := DEFAULT_FORE;
                  AFT : in FIELD := DEFAULT_AFT;
                  EXP : in FIELD := DEFAULT_EXP);
    procedure GET(FROM : in STRING;
                  ITEM : out NUM;
                  LAST : out POSITIVE);
    procedure PUT(TO : out STRING;
                  ITEM : in NUM;
                  AFT : in FIELD := DEFAULT_AFT;
                  EXP : in INTEGER := DEFAULT_EXP);
  end FIXED_IO;
  ...
end TEXT_IO;
```

A valós számok kiírására szolgáló eljárások a `FIXED_IO` és a `FLOAT_IO` generic package-kben találhatók. Mindkét generic paramétere a `NUM` típus.

Például a

```
type REAL is digits 8;
package REAL_IO is new TEXT_IO.FLOAT_IO(REAL);
X : REAL := 0.003974;
PUT(X); PUT(X,WIDTH => 15, MANTISSA => 4);
PUT(X,WIDTH => 8, MANTISSA => 2, EXPONENT => 1);
```

program eredménye a következő :

```
3.9740000E-3␣␣␣␣␣␣3.974E-03␣␣4.0E-3
```

Hasonlóan a

```
type FIX is delta 0.05 range -5 .. 5;
package FI_IO is new TEXT_IO.FIXED_IO(FIX);
X : FIX := 1.75;
PUT(X); PUT(X,WIDTH => 10); PUT(X,FRACT => 4);
```

program eredménye :

```
1.75␣␣␣␣␣␣1.751.7500
```

15.3.4. A felsorolási típusokra vonatkozó I/O

```
with IO_EXCEPTIONS;
package TEXT_IO is
  ...
  -- Generic package for Input-Output of Enumeration types
  generic
    type ENUM is (<>);
  package ENUMERATION_IO is
    DEFAULT_WIDTH : FIELD := 0;
    DEFAULT_SETTING : TYPE_SET := UPPER_CASE;
    procedure GET(FILE : in FILE_TYPE; ITEM : out ENUM);
    procedure GET(ITEM : out ENUM);
    procedure PUT(FILE : in FILE_TYPE;
                  ITEM : in ENUM;
                  WIDTH : in FIELD := DEFAULT_WIDTH;
                  SET : in TYPE_SET := DEFAULT_SETTING);
    procedure PUT(ITEM : in ENUM;
                  WIDTH : in FIELD := DEFAULT_WIDTH;
                  SET : in TYPE_SET := DEFAULT_SETTING);
    procedure GET(FROM : in STRING;
                  ITEM : out ENUM;
                  LAST : out POSITIVE);
    procedure PUT(TO : out STRING;
                  ITEM : in ENUM;
                  SET : in TYPE_SET := DEFAULT_SETTING);
  end ENUMERATION_IO;
  ...
end TEXT_IO;
```

A felsorolási típusok kiírása az `ENUMERATION_IO` generic package eljárásaival végezhető el. Ennek paramétere az `ENUM` típus.

15.4. Példaprogramok

i) A felsorolási típus input-outputja

Az alábbi program az TEXT_IO package beágyazott package-inek interaktív használatát mutatja be.

```
with TEXT_IO; use TEXT_IO;
procedure DIALOGUE is
  type COLOR is (WHITE, RED, ORANGE, YELLOW,
                 GREEN, BLUE, BROWN);
  package COLOR_IO is new ENUMERATION_IO(ENUM => COLOR);
  package INT_IO is new INTEGER_IO(INTEGER);
  use COLOR_IO, INT_IO;
  INVENTORY : array (COLOR) of INTEGER :=
                 (20, 17, 43, 10, 28, 173, 87);
  CHOICE : COLOR;
  procedure ENTER_COLOR(SELECTION:out COLOR) is
  begin
    loop
      begin
        PUT("Color selected: ");
        GET(SELECTION);
        exit;
      exception
        when DATA_ERROR =>
          PUT("Invalid color, try again. ");
          NEW_LINE(2);
      end;
    end loop;
  end ENTER_COLOR;
begin -- DIALOGUE
  INT_IO.DEFAULT_WIDTH := 5;
  loop
    ENTER_COLOR(CHOICE);
    SET_COL(5); PUT(CHOICE); PUT(" items available:");
    SET_COL(40); PUT(INVENTORY(CHOICE));
    NEW_LINE;
  end loop;
end DIALOGUE;
```

16. Függelék

A) A STANDARD package

Az Adában a STANDARD package definiálja az összes predefinit típust, objektumot és alprogramot. A package törzse implementációfüggő.

Ez a package nem írható le teljes egészében Ada nyelven: egyes implementációfüggő definíciókat csak jelezni tudunk az alábbi kódban, a predefinit operátorokat pedig csak megjegyzésben adjuk meg, mert ezek bár logikailag ebbe a package-be tartoznak, az Ada fordítóprogram közvetlenül "ismeri" őket. (Néhány helyen a hosszú felsorolást három ponttal jelöltük, de ez csak helytakarékosság miatt történt.)

```
package STANDARD is
  type BOOLEAN is (FALSE, TRUE);
  -- The predefined relational operators for this type are
  -- follows:
  -- function "="   (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
  -- function "/="  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
  -- function "<"   (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
  -- function "<="  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
  -- function ">"   (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
  -- function ">="  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
  -- The predefined logical operators and the predefined logical
  -- negation operator are as follows:
  -- function "and" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
  -- function "or"  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
  -- function "xor" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
  -- function "not" (RIGHT : BOOLEAN) return BOOLEAN;
  -- The universal type universal_integer is predefined
  type INTEGER is implementation_defined;
  -- The predefined operators for this type are follows:
  -- function "="   (LEFT, RIGHT : INTEGER) return BOOLEAN;
  -- function "/="  (LEFT, RIGHT : INTEGER) return BOOLEAN;
  -- function "<"   (LEFT, RIGHT : INTEGER) return BOOLEAN;
  -- function "<="  (LEFT, RIGHT : INTEGER) return BOOLEAN;
  -- function ">"   (LEFT, RIGHT : INTEGER) return BOOLEAN;
  -- function ">="  (LEFT, RIGHT : INTEGER) return BOOLEAN;
  -- function "+"   (RIGHT : INTEGER) return INTEGER;
  -- function "-"   (RIGHT : INTEGER) return INTEGER;
  -- function "abs" (RIGHT : INTEGER) return INTEGER;
  -- function "+"   (LEFT, RIGHT : INTEGER) return INTEGER;
  -- function "-"   (LEFT, RIGHT : INTEGER) return INTEGER;
  -- function "*"   (LEFT, RIGHT : INTEGER) return INTEGER;
```

```
-- function "/"   (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "rem" (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "mod" (LEFT, RIGHT : INTEGER) return INTEGER;

-- function "**"  (LEFT : INTEGER; RIGHT : INTEGER) return INTEGER;

-- An implementation may provide additional predefined integer
-- types. It is recommended that the names of such additional
-- types end with INTEGER as in SHORT_INTEGER or LONG_INTEGER.
-- The specification of each operator for the type
-- universal_integer, or for any additional predefined
-- integer type, is obtaoned by replacing INTEGER by the name of
-- the type in the specification of the corresponding operator
-- of the type INTEGER, except for the roght operand of the
-- exponentiating operator.

-- The universal type universal_real is predefined.

type FLOAT is implementation_defined;

-- The predefined operators for this type are as follows:
-- function "="   (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "/="  (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "<"   (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "<="  (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function ">"   (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function ">="  (LEFT, RIGHT : FLOAT) return BOOLEAN;

-- function "+"   (RIGHT : FLOAT) return FLOAT;
-- function "-"   (RIGHT : FLOAT) return FLOAT;
-- function "abs" (RIGHT : FLOAT) return FLOAT;

-- function "+"   (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "-"   (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "*"   (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "/"   (LEFT, RIGHT : FLOAT) return FLOAT;

-- function "**"  (LEFT : FLOAT; RIGHT : INTEGER) return FLOAT;

-- An implementation may provide additional predefined floating
-- point types. It is recommended that the names of such
-- additional types end with FLOAT as in SHORT_FLOAT or
-- LONG_FLOAT. The specification of each operator for the type
-- universal_real, or for any additional predefined
-- floating point type, is obtaoned by replacing FLOAT by the
-- name of the type in the specification of the corresponding
-- operator of the type FLOAT;

-- In addition, the following operators are predefined for
-- universal types:
-- function "*"  (LEFT : universal_integer; RIGHT : universal_real)
--                return universal_real;
-- function "*"  (LEFT : universal_real;    RIGHT : universal_integer)
--                return universal_real;
-- function "/"  (LEFT : universal_real;    RIGHT : universal_integer)
--                return universal_real;

-- The type universal_fixed is predefined. The only operators
-- declared for this type are
-- function "*"  (LEFT : any_fixed_point_type;
--                RIGHT : any_fixed_point_type)
--                return universal_fixed;
```

```
-- function "/"   (LEFT : any_fixed_point_type;
--                 RIGHT : any_fixed_point_type)
--                 return universal_fixed;

-- The following characters from the standard ASCII character
-- set. Character literals corresponding to control characters
-- are not identifiers; they are indicated in italics in this
-- definition.

type CHARACTER is
  (nul, soh, stx, etx,    eot, enq, ack, bel,
   bs, ht, lf, vt,        ff, cr, so, si,
   dle, dc1, dc2, dc3,    dc4, nak, syn, etb,
   cna, em, sub, esc,     fs, gs, rs, us,
   ' ', '!', '"', '#',    '$', '%', '&', ''',
   '(', ')', '*', '+',    ',', '-', '.', '/',
   '0', '1', '2', '3',    '4', '5', '6', '7',
   '8', '9', ':', ';',    '<', '=', '>', '?',
   '@', 'A', 'B', 'C',    'D', 'E', 'F', 'G',
   'H', 'I', 'J', 'K',    'L', 'M', 'N', 'O',
   'P', 'Q', 'R', 'S',    'T', 'U', 'V', 'W',
   'X', 'Y', 'Z', '[',    '\', ']', '^', '_',
   '^', 'a', 'b', 'c',    'd', 'e', 'f', 'g',
   'h', 'i', 'j', 'k',    'l', 'm', 'n', 'o',
   'p', 'q', 'r', 's',    't', 'u', 'v', 'w',
   'x', 'y', 'z', '{',    '|', '}', '~', del );

for CHARACTER use -- 128 ASCII character set without holes
  (0, 1, 2, 3, ..., 125, 126, 127);

-- The predefined operators for the type CHARACTER are the same
-- as for any enumeration type.

package ASCII is

-- Control characters
  NUL                 : constant CHARACTER := nul;
  SOH                 : constant CHARACTER := soh;
  STX                 : constant CHARACTER := stx;
  ETX                 : constant CHARACTER := etx;
  EOT                 : constant CHARACTER := eot;
  ENQ                 : constant CHARACTER := enq;
  ACK                 : constant CHARACTER := ack;
  BEL                 : constant CHARACTER := bel;
  BS                  : constant CHARACTER := bs;
  HT                  : constant CHARACTER := ht;
  LF                  : constant CHARACTER := lf;
  VT                  : constant CHARACTER := vt;
  FF                  : constant CHARACTER := ff;
  CR                  : constant CHARACTER := cr;
  SO                  : constant CHARACTER := so;
  SI                  : constant CHARACTER := si;
  DLE                 : constant CHARACTER := dle;
  DC1                 : constant CHARACTER := dc1;
  DC2                 : constant CHARACTER := dc2;
  DC3                 : constant CHARACTER := dc3;
  DC4                 : constant CHARACTER := dc4;
  NAK                 : constant CHARACTER := nak;
  SYN                 : constant CHARACTER := syn;
```

```
  ETB                : constant CHARACTER := etb;
  CNA                : constant CHARACTER := cna;
  EM                 : constant CHARACTER := em;
  SUB                : constant CHARACTER := sub;
  ESC                : constant CHARACTER := esc;
  FS                 : constant CHARACTER := fs;
  GS                 : constant CHARACTER := gs;
  RS                 : constant CHARACTER := rs;
  US                 : constant CHARACTER := us;
-- Other characters
  EXCLAM             : constant CHARACTER := '!';
  QUOTATION          : constant CHARACTER := '"';
  SHARP              : constant CHARACTER := '#';
  DOLLAR             : constant CHARACTER := '$';
  PERCENT            : constant CHARACTER := '%';
  AMPERSAND          : constant CHARACTER := '&';
  COLON              : constant CHARACTER := ':';
  SEMICOLON          : constant CHARACTER := ';';
  QUERY              : constant CHARACTER := '?';
  AT_SIGN            : constant CHARACTER := '@';
  L_BRACKET          : constant CHARACTER := '[';
  BACK_SLASH         : constant CHARACTER := '\';
  R_BRACKET          : constant CHARACTER := ']';
  CIRCUMFLEX         : constant CHARACTER := '^';
  UNDERLINE          : constant CHARACTER := '_';
  GRAVE              : constant CHARACTER := '`';
  L_BRACE            : constant CHARACTER := '{';
  BAR                : constant CHARACTER := '|';
  R_BRACE            : constant CHARACTER := '}';
  TILDE              : constant CHARACTER := '~';
-- Lower case letters
  LC_A : constant CHARACTER := 'a';
  ...
  LC_Z : constant CHARACTER := 'z';
end ASCII;
-- Predefined subtypes
subtype NATURAL  is INTEGER range 0 .. INTEGER'LAST;
subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST;
-- Predefined string type
type STRING is array ( POSITIVE range <> ) of CHARACTER;
pragma PACK(STRING);
-- The predefined operators for this types are as follows:
-- function "="   (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "/="  (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "<"   (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "<="  (LEFT, RIGHT : STRING) return BOOLEAN;
-- function ">"   (LEFT, RIGHT : STRING) return BOOLEAN;
-- function ">="  (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "&"   (LEFT : STRING;    RIGHT : STRING)
--                 return STRING;
-- function "&"   (LEFT : CHARACTER; RIGHT : STRING)
--                 return STRING;
```

```
  -- function "&"   (LEFT : STRING;    RIGHT : CHARACTER)
  --                 return STRING;
  -- function "&"   (LEFT : CHARACTER; RIGHT : CHARACTER)
  --                 return STRING;

  -- type DURATION is delta implementation_defined
  --                   range implementation_defined;

  -- The predefined operators for the type DURATION are the same
  -- as for any fixed point type.

  -- The predefined exceptions:
  CONSTRAINT_ERROR : exception;
  NUMERIC_ERROR    : exception;
  PROGRAM_ERROR    : exception;
  STORAGE_ERROR    : exception;
  TASKING_ERROR    : exception;

end STANDARD;
```

B) Az Ada fogalmainak összefoglalása

Accept utasítás (Accept statement). Ld. *entry.*

Access érték (Access value). Ld. *access típus.*

Access típus (Access type). Egy access típus értéke — azaz egy *access érték* — a null érték, és azok az *objektum*ot *kijelölő* értékek, amiket *allokátor*ral hoztunk létre. Az ilyen objektum értékére hivatkozni, ill. az objektum értékét módosítani az access értéken keresztül lehet. Egy access típus definíciója specifikálja azoknak az objektumoknak a típusát, amelyek az access típus értékeivel kijelölhetők. Ld. még *kollekció.*

Aktuális paraméter (Actual parameter). Ld. *paraméter.*

Aggregátum (Aggregate). Egy aggregátum *kiértékelés*ének eredménye egy *összetett típus*ú érték. Az összetett értéket a komponensek értékei határozzák meg. A komponensek megadásánál a *pozicionális megfeleltetés*, vagy a *névvel jelölt megfeleltetés* használható.

Alegység (Subunit). Ld. *törzs.*

Alegység szülője (Parent unit). Egy alegység szülőjének nevezzük azt fordítási egységet (törzset vagy alegységet), amelyben az alegységre vonatkozó *csonk* van.

Allokátor (Allocator). Egy allokátor kiértékelése létrehoz egy *objektum*ot, s egy új *access érték*et, ami *kijelöli* ezt az objektumot.

Alprogram (Subprogram). Az *eljárás*t és a *függvény*t együttvéve alprogramnak mondjuk. Az eljárás tevékenységek egy sorozatát írja elő, aminek *végrehajtás*át egy *eljáráshívás*sal kezdeményezhetjük. A függvény szintén specifikálja tevékenységek egy sorozatát, valamint specifikál egy visszatérési értéket, a *függvény eredményét.* A *függvényhívás* egy kifejezés. Az alprogramot *alprogramspecifikáció* és *alprogramtörzs* formájában adjuk meg. Az alprogramspecifikációban rögzítjük a *nevét*, a *formális paraméter*eit, és (függvény esetén) az eredményét. Az alprogramtörzs tartalmazza a tevékenységek felsorolását. Az alprogramhívás *aktuális paraméter*eket tartalmaz, amiket a hívás megfeleltet a formális paramétereknek. Az alprogram a *programegység*ek egy fajtája.

Alprogramspecifikáció (Subprogram specification). Ld. *alprogram.*

Alprogramtörzs (Subprogram body). Ld. *alprogram.*

Altípus (Subtype). Egy *típus* altípusa a típusértékhalmaz egy részhalmazával jellemezhető. A részhalmazt a *típus*ra (*bázistípus*ra) megadott *megszorítás*sal

határozhatjuk meg. A részhalmazban levő értékekre azt mondjuk, hogy az *altípus*hoz tartoznak, és *eleget tesznek a megszorításnak*. Az altípus *művelet*ei ugyanazok, mint a bázistípus műveletei.

Attributum (Attribute). Egy attributum kiértékelésének eredménye valamely egyed (pl. típus, objektum, task) predefinit jellemzője. Az attributumok általában *függvény*ek.

Átlapolás (Overloading). A programszöveg egy pontján egy azonosítónak több jelentése is lehet: ezt a tulajdonságot nevezzük átlapolhatóságnak. Például: átlapolt felsorolási típusú literál lehet egy olyan azonosító, ami több *felsorolási típus* definíciójában is szerepel. Egy átlapolt azonosító valódi jelentését mindig a szövegkörnyezet határozza meg. Az *alprogram*ok, az *aggregátum*ok, az *allokátor*ok és a string *literál*ok szintén átlapolhatók.

Átnevezés (Renaming declaration). Az átnevezés újabb névvel lát el egy már létező egyedet.

Bázistípus (Base type). Azt a típust, amelynek *megszorítás*ával az *altípus*t előállítottuk, az altípus bázistípusának nevezzük. A megszorítatlan típus bázistípusa önmaga.

Blokk utasítás (Block statement). A blokk utasítás egy egyszerű utasítás, ami tartalmazhat egy utasítás-sorozatot. Szintén tartalmazhat *deklarációs rész*t és *kivételkezelő*t, amelyeknek hatása csak a blokk utasításaira terjed ki.

Csonk (Stub). Egy beágyazott programegység *törzs*ét leválaszthatjuk a tartalmazó fordítási egységről, s belőle *alegység*et képezhetünk. Az alegység törzsének helyén maradó részt (**separate**) csonknak nevezzük.

Deklaráció (Declaration). A deklaráció összekapcsol egy azonosítót (vagy más elnevezésre alkalmas jelölést) egy egyeddel. Ez az összekapcsolás a programszöveg egy szakaszán, a deklaráció *hatáskör*én belül érvényes. A deklaráció hatáskörén belül használható az azonosító az egyedre való hivatkozásra, másszóval, a *név* a hozzákapcsolt egyedet *jelöli*.

Deklarációs rész (Declarative part). A deklarációs rész *deklaráció*k sorozatából áll. Tartalmazhat ezen felül *alprogramtörzs*eket és *reprezentációs előírás*okat is.

Diszkrét típus (Discrete Type). A diszkrét típus egy olyan *típus*, aminek értékkészlete meghatározott (skalár) értékek rendezett halmaza. Diszkrét típusok a *felsorolási típus*ok és az *egész típus*ok. Diszkrét típusú értékek használhatók indexelésre, iterációs ciklus megszervezésére, az esetek leírására a **case** utasításban, és *variáns rész* képzésére a rekord típuson belül.

Diszkrimináns (Discriminant). A diszkrimináns egy kitüntetett *komponens*e a *rekord típus*ú *objektum*oknak és értékeknek. A többi komponens *altípus*a, sőt jelenléte függhet a diszkrimináns értékétől.

Diszkrimináns megszorítása (Discriminant constraint). Egy *rekord típus* vagy egy *privát típus* diszkriminánsának megszorítása a *típus* mindegyik diszkriminánsának értékét rögzíti.

Egész típus (Integer type). Egy egész típus olyan *diszkrét típus*, amelynek értékkészletét egy meghatározott *intervallum*ba eső egész számok halmaza reprezentálja.

Egyszerű név (Simple name). Ld. *deklaráció*, *név*.

Elaboráció (Elaboration). Egy *deklaráció* elaborációja az a folyamat, amelynek során a deklaráció kifejti hatását (azaz létrehozza az *objektum*ot, hozzárendeli a nevét, a kezdőértékét stb.). Az elaboráció a program futása során zajlik le.

Eleget tesz a megszorításnak (Satisfy). Ld. *megszorítás*, *altípus*.

Eljárás (Procedure). Ld. *alprogram*.

Eljáráshívás (Procedure call). Ld. *alprogram*.

Entry. A entry-ket a *task*ok közötti kommunikációban használjuk. Az entry hívása a *eljáráshívás*hoz hasonlóan történik. Az entry "belső tulajdonságait" egy vagy több *accept utasítás*sal definiáljuk, amelyek leírják azt a tevékenységsorozatot, amit az entry meghívása esetén el kell végezni.

Eltakarás (Hidden). Egy *deklarációs rész*en belül elhelyezkedhet egy másik deklarációs rész is. Ha ekkor a belső deklarációs részben előfordul a külső deklarációs részben levő *deklaráció* egy *homográf*ja, akkor azt mondjuk, hogy a belső deklarációs részben a deklaráció eltakarja a külső deklarációt.

Értékadás (Assignment). Az értékadás egy olyan *művelet*, ami kicseréli egy *változó* értékét egy új értékre. Az értékadó utasítás a baloldallal specifikálja a változót, s a jobboldallal annak új értékét.

Felsorolási típus (Enumeration type). Egy felsorolási típus olyan *diszkrét típus*, amelynek értékkészletét a *típus* definíciójában megadott literálokkal reprezentáljuk. A felsorolási típusú literálként azonosítókat és karakteres literálok használhatunk.

Fixpontos típus (Fixed point type). Ld. *valós típus*.

Fordítási egység (Compilation unit). Fordítási egység lehet egy *programegység* specifikációja vagy *törzs*e, amit önálló szövegként fordítunk le, vagy egy *alegység*. A fordítási egységet megelőzheti egy *környezeti előírás*, ami megnevezi azokat a fordítási egységeket, amiktől a fordítási egység függ.

Formális paraméter (Formal parameter). Ld. *paraméter*.

Függvény (Function). Ld. *alprogram.*

Függvény eredménye (Function result). Ld. *alprogram.*

Függvényhívás (Function call). Ld. *alprogram.*

Generic. A generic *alprogram*ok vagy *package*-k előállítására szolgáló minta. A minta segítségével előállított alprogramot vagy package-t a generic egy példányának hívjuk. A *példányosítás* a *deklaráció*k egy fajtája, amivel előállítható egy példány. A generic formailag egy alprogram vagy egy package, amelynek a specifikációs részét megelőzi a generic formális paramétereinek felsorolása. A generic formális paramétere lehet *típus*, *alprogram* vagy *objektum*. A generic a *programegység*ek egy fajtája.

Hatáskör (Scope). Ld. *deklaráció.*

Homográf deklaráció (Homograph declaration). Két *deklaráció* egymás homográfja, ha ugyanazt az azonosítót, operátort vagy karakter literált deklarálják, és — ha egyáltalán van, akkor — megegyezik a paramétereik és az eredményük *profile*-ja is.

Index. Ld. *tömb típus.*

Indexelt komponens (Indexed component). Az indexelt komponens egy *tömb* egy elemét *jelöli* meg. Ez a *név* egy formája, amely olyan *kifejezés*eket tartalmaz, amelyek meghatározzák az *index*ek értékét. Indexelt komponens jelölhet egy *entry*-t az entry-k családjában.

Indexmegszorítás (Index constraint). Egy *tömb típus* indexmegszorítása a típus mindegyik indexének definiálja az alsó és a felső határát.

Intervallum (Range). Egy *skalár típus* értékkészletének egy összefüggő halmazát intervallumnak hívjuk. Az intervallumot az értékek alsó és felső határának megadásával specifikáljuk.

Intervallum megszorítása (Range constraint). Egy *skalár típus* megszorítása az értékkészletnek egy részintervallumra való leszűkítését jelenti.

Jelöl (Denote). Ld. *deklaráció.*

Kiértékelés (Evaluation). Egy *kifejezés* kiértékelése az a folyamat, amelynek során kiszámításra kerül a kifejezés értéke. Ez a folyamat a program futása során zajlik le.

Kifejezés (Expression). A kifejezés egy érték kiszámításának módját definiálja.

Kijelölés (Designate). Ld. *access típus*, *task.*

Kiterjesztett név (Expanded name). A kiterjesztett név egy olyan egyedet *jelöl*, amit közvetlenül egy konstrukció belsejében *deklarált*unk. A kiterjesztett nevet *szelekciós formá*ban írjuk: a *prefix* jelöli a konstrukciót (ez lehet pl. *programegység*, *blokk utasítás*, ciklus vagy *accept utasítás*), az utótag (a *szelektor*) pedig az egyed *egyszerű neve*.

Kivétel (Exception). A kivétel egy olyan esemény, ami a program végrehajtása során léphet fel. A *kivétel kiváltás*a (fellépése) megszakítja a program végrehajtását. A *kivételkezelő* a programszövegnek az a része, ami specifikálja a kivétel fellépésekor elvégzendö tevékenységet. Egy kivétel fellépése után kiválasztásra kerül a hozzá tartozó *kivételkezelő*, majd ennek végrehajtása után a program végrehajtása normál módon folytatódik.

Kivétel kiváltása (Raising an exception). Ld. *kivétel*.

Kivételkezelő (Exception handler). Ld. *kivétel*.

Kollekció (Collection). A kollekció azoknak az *objektum*oknak a halmaza, amik ugyanarra az *access típus*ra vonatkozó *allokátor*ok kiértékelése eredményeképpen álltak elő.

Komponens (Component). A komponens egy olyan érték, ami egy összetett érték része, vagy olyan *objektum*, ami egy összetett objektumnak a része.

Konstans (Constant). Ld. *objektum*.

Környezeti előírás (Context clause). Ld. *fordítási egység*.

Közvetlen láthatóság (Direct visibility). Ld. *láthatóság*.

Látható rész (Visible part). Ld. *package*.

Láthatóság (Visibility). A programszöveg egy adott pontján egy *deklaráció* (tehát egy egyed és egy azonosító összekapcsolása) látható, ha az azonosítónak az illető egyed az egyik lehetséges jelentése.

Lebegőpontos típus (Floating point type). Ld. *valós típus*.

Lexikális elem (Lexical element). Lexikális elemek az azonosítók, a *literál*ok, az elhatároló jelek és a megjegyzések.

Limited típus (Limited type). A limited típus egy olyan típus, aminek semmilyen predefinit művelete nincs (tehát sem az értékadás, sem az implicit "=" művelet nem megengedett). A *task* típus limited típus, ezen kívül egy *privát típus*t is limited típusnak definiálhatunk. A limited típushoz explicit módon definiálható az "=" művelet.

Literál (Literal). A literálokat típusértékek reprezentálására használjuk. Vannak

numerikus, felsorolási típusú, karakteres és string literálok.

Megszorítás (Constraint). Egy megszorítás egy típus értékeinek részhalmazát definiálja. Azt mondjuk, hogy ennek a részhalmaznak egy eleme *eleget tesz a megszorításnak.*

Minősített kifejezés (Qualified expression). A minősített kifejezés egy olyan *kifejezés*, ami tartalmazza a *típus*ának vagy *altípus*ának a megjelölését is. A minősítést akkor használjuk, ha a kifejezés többértelmű, a típusa nem állapítható meg egyértelműen (pl. egy *átlapolás* következtében).

Modellszám (Model number). A modellszámoknak nevezzük a *valós típus*ok pontosan ábrázolható értékeit. A valós típusok *művelet*eit (pl. a kerekítést) a típus modellszámainak segítségével definiáljuk. A modellszámok és azok műveleteinek tulajdonságai adják meg a valós típusokra azt a minimális előírást, amit a nyelv minden implementációjának meg kell valósítania.

Mód (Mode). Ld. *paraméter.*

Művelet (Operation). Egy típus *művelet*ei a típusértékeken végrehajtható elemi tevékenységek. A műveletek lehetnek a típus definíciója által implicit módon meghatározottak, de megadhatók mint *alprogram*ok is, amelyek *paramétere* vagy a *függvény eredménye* az adott típus egy értéke.

Névvel jelölt megfeleltetés (Named association). A *névvel jelölt megfeleltetés* egy értéknek egy lista egy vagy több pozíciójához való hozzárendelését e pozíciók megnevezésével adja meg.

Név (Name). A név az egyedek kezelésére létrehozott konstrukció. Azt mondjuk, hogy egy név egy egyedet *jelöl*, ill. egy egyed a név jelentése. Ld. még *deklaráció*, *prefix.*

Objektum (Object). Az objektumok értékkel rendelkeznek. A program vagy egy objektum *deklaráció*jának *elaborációja*, vagy egy *allokátor kiértékelése* eredményeképpen hoz létre új objektumot. A deklaráció vagy az allokátor meghatározza az objektum *típus*át is: az objektum csak ilyen típusú értékeket vehet fel.

Operátor (Operator). Az operátor egy müvelet, aminek egy vagy két operandusa lehet. Az unáris operátort az operandus elé, a bináris operátort a két operandus közé írjuk. Ez a jelölés a *függvényhívás* egy speciális formája. Egy operátor jelentését függvény formájában definiálhatjuk. A típusokhoz általában léteznek implicit módon definiált operátorok is (pl. "=", "<" stb.).

Összetett típus (Compound type). Az Ósszetett típus értékei komponensekből állnak. Összetett típusok a *tömb típus* és a *rekord típus.*

Ősegység (Ancestor). Egy *alegység szülő*i láncán visszafele haladva egy valódi

*törzs*höz jutunk, amelynek már nincs szülője. Ezt a fordítási egységet az alegység ősegységének nevezzük.

Őstípus (Parent type). Ld. *származtatott típus.*

Package. A package logikailag összetartozó egyedek, *típus*ok, *objektum*ok, és *alprogram*ok gyűjteménye. A package-t *package specifikáció* és *package törzs* formájában adjuk meg. A packade specifikációjának van egy *látható része*, ami azoknak az egyedeknek a *deklaráció*it tartalmazza, amelyek a package-n kívül is láthatóak. Emellett tartalmazhat egy *privát rész*t is, amelynek a deklarációi kiegészítik, teljessé teszik a látható részben levő definíciókat, a package-n kívül azonban nem láthatóak. A package törzse tartalmazza a package specifikációs részében felsorolt *alprogram*ok törzseit. A package a *programegység*ek egy fajtája.

Package spacifikáció (Package specification). Ld. *package.*

Package törzs (Package body). Ld. *package.*

Paraméter (Parameter). Az *alprogram*, az *entry* és a *generic* paramétere egy névvel rendelkező egyed, amit a hozzá tartozó *alprogramtörzs*sel, *accept utasítás*sal ill. generic törzzsel való kommunikálásra használunk. A *formális paraméter* egy azonosító, ami a törzsben az egyedre való hivatkozásra szolgál. Az *aktuális paramétert* a *alprogramhívás*, az *entry hívás* ill. a *példányosítás* köti össze a megfelelő formális paraméterrel. A formális paraméter *mód*ja határozza meg, hogy a megfelelő aktuális paraméter adja át az értékét a formális paraméternek, vagy a formális paraméter adja át az értékét az aktuális paraméternek, vagy pedig mindkettő megtörténik. Az aktuális paraméterek és formális paraméterek egymáshoz rendelése történhet *névvel jelölt megfeleltetés*, vagy *pozicionális megfeleltetés* útján.

Példányosítás (Instantiation). Ld. *generic.*

Pozicionális megfeleltetés (Positional assotiation). A *pozicionális megfeleltetés* egy értéknek egy lista egy pozíciójához való hozzárendelését írja le úgy, hogy az értéket annyiadikként sorolja fel, ahányadik pozícióban a listában szerepelnie kell.

Pragma. A pragma a fordítóprogramnak szóló előírásokat foglal magába.

Prefix. A prefix a *nev*ek egyes formáiban a konstrukció első tagjaként állhat. Prefix lehet a *függvénhívás* és a *név.*

Privát rész (Private part). Ld. *package.*

Privát típus (Private type). A privát típus egy olyan *típus*, amelynek a felépítése és az értékhalmaza pontosan definiált, de a típust használók számára közvetlenül nem hozzáférhető. A privát típust az azt használó csak a *diszkriminánsa*in (ha vannak) és a *művelet*ein keresztül ismerheti. A privát típus és annak műveletei

egy *package látható részé*ben, vagy egy *generic formális paramétere*i között definiálhatók. Az *értékadás*, az "=" és a "/=" műveletek a privát típusra implicit módon definiáltak, kivéve, ha a típus *limited típus*.

Profile. Két formális paraméter rész profile-ja pontosan akkor egyezik meg, ha ugyanannyi paramétert tartalmaznak, s a paraméterek *bázistípus*a rendre megegyezik. Két *függvény* eredményének profile-ja pontosan akkor egyezik meg, ha az eredmények *bázistípus*a megegyezik.

Program. A program valahány *fordítási egység*ből állítható össze, amelyek között az egyik *alprogram* szerepe kitüntetett, ez a főprogram. A program végrehajtása a főprogram végrehajtását jelenti, ami meghívhat olyan alprogramokat, amelyek a program más fordítási egységeiben helyezkednek el.

Programegység (Program unit). A programegységnek négy fajtája van: a *generic*, a *package*, az *alprogram* és a *task*.

Randevú (Randezvous). A randevú két *task* közötti bizonyos fajta együttműködés: az egyik task hívja a másik egy *entry* pontját, s a fogadó task végrehajtja a megfelelő *accept utasítás*t.

Reprezentációs előírás (Representation clause). A reprezentációs előírás meghatározza a fordítóprogram számára egy *típus*, *objektum* vagy *task* egyes tulajdonságainak leképezését a tárgygép szolgáltatásaira. Egyes esetekben teljesen specifikálja a leképezést, máskor feltételeket definiál a leképezés megválasztásához.

Rekord típus (Record type). Egy rekord típusú érték *komponens*ekből áll, amelyek általában különböző *típus*úak ill. *altípus*úak. A rekord típus mindegyik komponenshez hozzárendel egy azonosítót, amellyel egyértelműen kiválasztható az érték vagy az *objektum* komponense.

Résztömb (Slice). Résztömbnek nevezünk egy egydimenziós *tömb típus*ú érték vagy *objektum index*einek egy *intervallum*ával meghatározott egydimenziós tömböt.

Statikus kifejezés (Static expression). Statikusnak nevezzük azokat a kifejezéseket, amelyek értéke fordítási időben kiszámítható.

Skalár típus (Scalar type). A skalár típusú értékeknek és *objektum*oknak nincsenek *komponens*ei. Skalár típusok a *diszkrét típus*ok és a *valós típus*ok. A skalár típus értékhalmaza rendezett halmaz.

Származtatott típus (Derived Type). A származtatott típus egy olyan *típus*, amelynek értékei és műveletei egy már létező típus másolatai. Ezt a már létező típust a származtatott típus *őstípus*ának nevezzük.

Szelekciós jelölés (Selected komponent). A szelekciós jelölés egy *név*, ami tartalmaz egy *prefix*et, s egy *szelektor*nak nevezett azonosítót. Szelekciós jelölést

használunk a rekord komponenseinek, az *entry*-knek, és az *access értékk*el elérhető *objektum*oknak a jelölésére. A szelekciós jelölést használjuk a *kiterjesztett nev*ek leírásához is.

Szelektor (Selector). Ld. *szelekciós jelölés.*

Task. A task a program többi részével párhuzamosan hajtódik végre. A taskot *task specifikáció* (ami specifikálja a task *nev*ét, s az entry-k *formális paramétere*it), és *task törzs* (ami definiálja a task hatását) formájában adjuk meg. A task egy fajtája a *programegység*eknek. A *task típus* egy *típus*, ami lehetővé teszi több hasonló task egyszerű *deklarál*ását. A task típus egy értéke *kijelöl* egy taskot.

Task specifikáció (Task specification). Ld. *task.*

Task típus (Task type). Ld. *task.*

Task törzs (Task body). Ld. *task.*

Típus (Type). Egy típus az értékhalmazával, s az ezen értékekre alkalmazható *művelet*ekkel jellemezhető. Az *access típus*, a *tömb típus*, a *rekord típus*, a *skalár típus*, és a *task típus* mind a t´ pusok egy-egy osztályát jelentik.

Tömb típus (Array type). Egy tömb típusú érték olyan *komponens*ekből áll, emelyek ugyanabba az *altípus*ba (*típus*ba) tartoznak. Mindegyik komponens egyértelműen meghatározható az *index*e (egydimenziós tömb esetén), vagy az indexeinek egy sorozata (többdimenziós tömb esetén) segítségével. Mindegyik indexnek valamely *diszkrét típus*hoz kell tartoznia, s a megengedett *intervallum*ba kell esnie.

Törzs (Body). A törzs definiálja egy *alprogram*, *package* vagy egy *task* hatását. A *csonk* a törzs egy sajátos formája, ami azt jelzi, hogy a törzset egy önállóan fordítható *alegység*ben adjuk meg.

Use utasítás (Use clause). A use utasítás egy *package látható rész*ében levő *deklaráció*kat *közvetlenül láthatóvá* tesz.

Utasítás (Statement). Egy utasítás egy vagy több olyan tevékenységet definiál, amik a *program* futása során hajtódnak végre.

Valós típus (Real type). A valós típus egy *típus*, amelynek értékei a valós számok közelítései. A valós típusoknak két fajtája van: A *fixpontos típus*okat a közelítés megengedett abszolút hibájával, a *lebegőpontos típus*okat a közelítés relatív hibájával (az ábrázoláshoz felhasználható decimális jegyek rögzítésével) definiáljuk.

Variáns rész (Variant part). A *rekord típus* variáns része a *diszkriminánst*ól függően definiálja a rekord *komponens*eit. A diszkrimináns minden egyes értékéhez hozzátartozik a komponensek egy pontosan definiált halmaza.

Változó (Variable). Ld. *objektum.*

Végrehajtás (Execution). Azt a folyamatot, amelynek során egy utasítás kifejti a hatását, az utasítás végrehajtásának mondjuk.

With utasítás (With clause). Ld. *fordítási egység.*

C) Az Ada szintaktikus szabályai névsorba szedve

<*abort statement*> ::= **abort** <*task name*> { **,** <*task name*>} **;**
<*accept alternative*> ::= <*accept statement*> [<*sequence of statements*>]
<*accept statement*> ::= **accept** <*entry simple name*> [**(** <*entry index*> **)**]
[<*formal part*>]
[**do** <*sequence of statements*>
end [<*entry simple name*>]] **;**
<*access type definition*> ::= **access** <*subtype indication*>
<*actual parameter part*> ::= **(** <*parameter association*>
{ **,** <*parameter association*>} **)**
<*actual parameter*> ::= <*expression*>|<*variable name*>
|<*type mark*> **(** <*variable name*> **)**
<*address clause*> ::= **for** <*simple name*> **use at** <*simple expression*> **;**
<*aggregate*> ::= **(** <*component association*> { **,** <*component association*>} **)**
<*alignment clause*> ::= **at mod** <*static simple expression*> **;**
<*allocator*> ::= **new** <*subtype indication*>| **new** <*qualified expression*>
<*argument association*> ::= [<*argument identifier*> **=>** <*name*>
| [<*argument identifier*> **=>**] <*expression*>
<*array type definition*> ::= <*unconstrained array definition*>
|<*constrained array definition*>
<*assignment statement*> ::= <*variable name*> **:=** <*expression*> **;**
<*attribute designator*> ::= <*simple name*>
[**(** <*universal static expression*> **)**]
<*attribute*> ::= <*prefix*> **´** <*attribute designator*>
<*base*> ::= <*integer*>
<*based integer*> ::= <*extended digit*> { [<*underline*>] <*extended digit*>
<*based literal*> ::= <*base*> **#** <*based integer*>
[**.** <*based integer*>] **#** <*exponent*>
<*basic declaration*> ::= <*object declaration*>|<*number declaration*>
|<*type declaration*>|<*subtype declaration*>|<*subprogram declaration*>
|<*package declaration*>|<*task declaration*>|<*generic declaration*>
|<*exception declaration*>|<*generic instantiation*>|<*renaming declaration*>
|<*deferred constant declaration*>
<*basic declarative item*> ::= <*basic declaration*>|<*represetation clause*>
|<*use clause*>
<*basic graphic character*> ::= <*upper case letter*>
|<*digit*>|<*special character*>|<*space character*>
<*binary adding operator*> ::= **+** | **–** | **&**
<*block statement*> ::= [<*block simple name*> **:**]
[<*declare declarative part*>]
begin <*sequence of statements*>
[**exception** <*exception handler* {<*exception handler*>}
end [<*block simple name*>] **;**
<*body stub*> ::= <*subprogram specification*> **is separate ;**
| **package body** <*package simple name*> **is separate ;**
| **task body** <*task simple name*> **is separate ;**
<*body*> ::= <*proper body*>|<*body stub*>

<case statement alternative> ::= **when** <choice> { | <choice>} **=>**
<sequence of statements>
<case statement> ::= **case** <expression> **is**
<case statement alternative> {<case statement alternative>}
end case ;
<character literal> ::= ´ <graphic character> ´
<choice> ::= <simple expression>|<discrete range>| **others**
|<component simple name>
<compilation unit> ::= <context clause><library unit>
|<context clause><secondary unit>
<compilation> ::= {<compilation unit>}
<component association> ::= [<choice> { | <choice>} **=>**] <expression>
<component clause> ::= <component name> **at** <static simple expression>
range <static range> ;
<component declaration> ::= <identifier list> :
<component subtype definition> [**:=** <expression>] ;
<component list> ::= <component declaration> {<component declaration>}
| {<component declaration>} <variant part>| **null ;**
<component subtype definition> ::= <subtype indication>
<compound statement> ::= <if statement>|<case statement>
|<loop statement>|<block statement>|<accept statement>
|<select statement>
<condition> ::= <boolean expression>
<conditional entry call> ::= **select** <entry call statement>
[<sequence of statements>]
else <sequence of statements>
end select ;
<constrained array definition> ::= **array** <index constraint> **of**
<component subtype indication>
<constraint> ::= <range constraint>|<floating point constraint>
|<fixed point constraint>|<index constraint>|<discriminant constraint>
<context clause> ::= {<with clause> {<use clause>}}
<decimal literal> ::= <integer> [. <integer>] [<exponent>]
<declarative part> ::= {<basic declarative item>}{<later declarative item>}
<deferred constant declaration> ::= <identifier list> :
constant <type mark> ;
<delay alternative> ::= <delay statement> [<sequence of statements>]
<delay statement> ::= **delay** <simple expression> ;
<derived type definition> ::= **new** <subtype indication>
<designator> ::= <identifier>|<operator symbol>
<discrete range> ::= <discrete subtype indication>|<range>
<discriminant association> ::= [<discriminant simple name>
{ | <discriminant simple name>} **=>**] <expression>
<discriminant constraint> ::= (<discriminant association>
{ , <discriminant association>})
<discriminant part> ::= (<discriminant specification>
{ ; <discriminant specification>})
<discriminant specification> ::= <identifier list> :
<type mark> [**:=** <expression>]
<entry call statement> ::= <entry name> [<actual parameter part>] ;

<*entry declaration*> ::= **entry** <*identifier*> [(<*discrete range*>)]
[<*formal part*>] ;
<*entry index*> ::= <*expression*>
<*enumaration literal specification*> ::= <*enumeration literal*>
<*enumeration literal*> ::= <*identifier*>|<*character literal*>
<*enumeration representation clause*> ::= **for** <*type simple name*>
use <*aggregate*> ;
<*enumeration type definition*> ::= (<*enumaration literal specification*>
{ , <*enumaration literal specification*>})
<*exception choice*> ::= <*exception name*>| **others**
<*exception declaration*> ::= <*identifier list*> : **exception** ;
<*exception handler*> ::= **when** <*exception choice*> { | <*exception choice*>}
=> <*sequence of statements*>
<*exit statement*> ::= **exit** [<*loop name*>] [**when** <*condition*>] ;
<*exponent*> ::= E [+] <*integer*>| E - <*integer*>
<*expression*> ::= <*relation*> { **and** <*relation*>}
|<*relation*> { **and then** <*relation*>}
|<*relation*> { **or** <*relation*>}
|<*relation*> { **or else** <*relation*>}
|<*relation*> { **xor** <*relation*>}
<*extended digit*> ::= <*digit*>|<*letter*>
<*factor*> ::= <*primary*> [** <*primary*>] | **abs** <*primary*>| **not** <*primary*>
<*fixed accuracy definition*> ::= **delta** <*static simple expression*>
<*fixed point constraint*> ::= <*fixed accuracy definition*>
[<*range constraint*>]
<*floating accuracy definition*> ::= **digits** <*static simple expression*>
<*floating point constraint*> ::= <*floating accuracy definition*>
[<*range constraint*>]
<*formal parameter*> ::= <*parameter simple name*>
<*formal part*> ::= (*parameter specification*> { ; <*parameter specification*>})
<*full type declaration*> ::= **type** <*identifier*> [<*discriminant part*>] **is**
<*type definition*> ;
<*function call*> ::= <*function name*> [<*actual parameter part*>]
<*generic actual parameter*> ::= <*expression*>|<*variable name*>
|<*subprogram name*>|<*entry name*>|<*type mark*>
<*generic actual part*> ::= (<*generic association*>
{ ; <*generic association*>})
<*generic association*> ::= [<*generic formal parameter*> =>]
<*generic actual parameter*>
<*generic declaration*> ::= <*generic specification*> ;
<*generic formal parameter*> ::= <*parameter simple name*>|<*operator symbol*>
<*generic formal part*> ::= **generic** {<*generic parameter declaration*>}
<*generic instantiation*> ::= **package** <*identifier*> **is**
new <*generic package name*> [<*generic actual part*>] ;
| **procedure** <*identifier*> **is new** <*generic procedure name*>
[<*generic actual part*>] ;
| **function** <*designator*> **is new** <*generic function name*>
[<*generic actual part*>] ;
<*generic parameter declaration*> ::= <*identifier list*> : [**in** [**out**]]
<*type mark*> [:= <*expression*>] ;

```
    | type <identifier> is <generic type definition>
    |<private type declaration>
    | with <subprogram specification> [ is name> ] ;
    | with <subprogram specification> [ is <> ] ;
<generic specification> ::= <generic formal part><subprogram specification>
    |<generic formal part><package specification>
<generic type definition> ::= ( <> ) | range <> | digits <>
    | delta <> |<array type definition>|<access type definition>
<goto statement> ::= goto <label name> ;
<graphic character> ::= <basic graphic character>
    |<lower case letter>|<other special character>
<highest precedence operator> ::= ** | abs | not
<identifier list> ::= <identifier> { , <identifier>}
<identifier> ::= <letter> { [ <underline> ] <letter or digit>}
<if statement> ::= if <condition> then <sequence of statements>
    { elsif <condition> then <sequence of statements>}
    [ else <sequence of statements> ]
    end if ;
<incomplete type definition> ::= type <identifier> [ <discriminant part> ] ;
<index constraint> ::= ( <discrete range> { , <discrete range>} )
<index subtype definition> ::= <type mark> range <>
<indexed component> ::= <prefix> ( <expression> { , <expression>} )
<integer type definition> ::= <range constraint>
<integer> ::= <digit> { [ <underline> ] <digit>}
<iteration scheme> ::= while <condition>
    | for <loop parameter specification>
<label> ::= << <label simple name> >>
<later declarative item> ::= <body>|<subprogram declaration>
    |<package declaration>|<task declaration>|<generic declaration>
    |<use clause>|<generic instantiation>
<length clause> ::= for <attribute> use <simple expression> ;
<library unit body> ::= <subprogram body>|<package body>
<library unit> ::= <subprogram declaration>|<package declaration>
    |<generic declaration>|<generic instantiation>|<subprogram body>
<logical operator> ::= and | or | xor
<loop parameter specification> ::= <identifier> in [ reverse ]
    <discrete range>
<loop statement> ::= [ <loop simple name> : ] [ <iteration scheme> ] loop
    <sequence of statements>
    end loop [ <loop simple name> ] ;
<mode> ::= [ in ] | in out | out
<multiplying operator> ::= * | / | mod | rem
<name> ::= <simple name>|<character literal>|<operator symbol>
    |<indexed component>|<slice>|<selected component>|<attribute>
<null statement> ::= null ;
<number declaration> ::= <identifier list> : constant :=
    <universal static expression> ;
<numeric literal> ::= <decimal literal>|<based literal>
<object declaration> ::= <identifier list> : [ constant ]
    <subtype indication> [ := <expression> ] ;
```

```
    |<identifier list> : [ constant ]
    <constrained array definition> [ := <expression> ] ;
<operator symbol> ::= <string literal>
<package body> ::= package body <package simple name> is
    [ <declarative part> ]
    [ begin <sequence of statements>
    [ exception <exception handler> {<exception handler>} ] ]
    end [ <package simple name> ] ;
<package declaration> ::= <package specification> ;
<package specification> ::= package <identifier> is
    {<basic declarative item>}
    [ private {<basic declarative item>} ]
    end [ <package simple name> ]
<parameter association> ::= [ <formal parameter> => ] <actual parameter>
<parameter specification> ::= <identifier list> : <mode><type mark>
    [ := <expression> ]
<pragma> ::= pragma <identifier>
    [ ( <argument association> { , <argument association>} ) ] ;
<prefix> ::= <name>|<function call>
<primary> ::= <numeric literal>| null |<aggregate>|<string literal>
    |<name>|<allocator>|<function call>|<type conversion>
    |<qualified expression>| ( <expression> )
<private type declaration> ::= type <identifier> [ <discriminant part> ] is
    [ limited ] private ;
<procedure call statement> ::= <procedure name>
    [ <actual parameter part> ] ;
<proper body> ::= <subprogram body>|<package body>|<task body>
<qualified expression> ::= <type mark> ' ( <expression> )
    |<type mark> ' <aggregate>
<raise statement> ::= raise [ <exception name> ] ;
<range constraint> ::= range <range>
<range> ::= <range attribute>|<simple expression> .. <simple expression>
<real type definition> ::= <floating point constraint>
    |<fixed point constraint>
<record representation clause> ::= for <type simple name> use record
    [ <alignment clause> ]
    {<component clause>}
    end record ;
<record type definition> ::= record <component list> end record
<relation> ::= <simple expression>
    [ <relational operator><simple expression> ]
    |<simple expression> [ not ] in <range>
    |<simple expression> [ not ] in <type mark>
<relational operator> ::= = | /= | < | <= | > | >=
<renaming declaration> ::= <identifier> :
    <type mark> renames <object name> ;
    |<identifier> : exception renames <exception name> ;
    | package <identifier> renames <package name> ;
    |<subprogram specification> renames <subprogram or entry name> ;
<representation clause> ::= <type representation clause>|<address clause>
```

```
<return statement> ::= return [ <expression> ] ;
<secondary unit> ::= <library unit body>|<subunit>
<select alternative> ::= [ when <condition> => ] <selective wait alternative>
<select statement> ::= <selective wait>|<conditional entry call>
    |<timed entry call>
<selected component> ::= <prefix> . <selector>
<selective wait alternative> ::= <accept alternative>
    |<delay alternative>|<terminate alternative>
<selective wait> ::= select <select alternative>
    { or <select alternative>}
    [ else <sequence of statements> ]
    end select ;
<selector> ::= <simple name>|<character literal>|<operator symbol>| all
<sequence of statements> ::= <statement> {<statement>}
<simple expression> ::= [ <unary adding operator> ] <term>
    {<binary adding operator><term>}
<simple name> ::= <identifier>
<simple statement> ::= <null statement>|<assignment statement>
    |<procedure call statement>|<exit statement>|<return statement>
    |<goto statement>|<entry call statement>|<delay statement>
    |<abort statement>|<raise statement>|<code statement>
<slice> ::= <prefix> ( <discrete range> )
<statement> ::= {<label>} <simple statement>
    | {<label>} <compound statement>
<string literal> ::= " {<graphic character>} "
<subprogram body> ::= <subprogram specification> is
    [ <declarative part> ]
    begin <sequence of statements>
    [ exception <exception handler> {<exception handler>} ]
    end [ <designator> ] ;
<subprogram declaration> ::= <subprogram specification> ;
<subprogram specification> ::= procedure <identifier> [ <formal part> ]
    | function <designator> [ <formal part> ] return <type mark>
<subtype declaration> ::= subtype <identifier> is <subtype indication> ;
<subtype indication> ::= <type indication> [ <constraint> ]
<subunit> ::= separate ( <parent unit name> ) <body>
<task body> ::= task body <task simple name> is
    [ <declarative part> ]
    begin <sequence of statements>
    [ exception <exception handler> {<exception handler>} ]
    end [ <task simple name> ] ;
<task declaration> ::= <task specification> ;
<task specification> ::= task [ type ] <identifier> [ is
    {<entry declaration>}{<representation clause>}
    end [ <task simple name> ] ]
<term> ::= <factor> {<multiplying operator><factor>}
<terminate alternative> ::= terminate ;
<timed entry call> ::= select <entry call statement>
    [ <sequence of statements> ]
    or <delay alternative>
```

end select ;
<*type conversion*> ::= <*type mark*> (<*expression*>)
<*type declaration*> ::= <*full type declaration*>
|<*incomplete type declaration*>|<*private type declaration*>
<*type definition*> ::= <*enumeration type definition*>
|<*integer type definition*>|<*real type definition*>
|<*array type definition*>|<*record type definition*>
|<*access type definition*>|<*incomplete type definition*>
<*type mark*> ::= <*type name*>|<*subtype name*>
<*type representation clause*> ::= <*length clause*>
|<*enumeration representation clause*>
|<*record representation clause*>
<*unary adding operator*> ::= **+** | **–**
<*unconstrained array definition*> ::= **array** (<*index subtype definition*>
{ **,** <*index subtype definition*>}) **of** <*component subtype indication*>
<*use clause*> ::= **use** <*package simple name*> { **,** <*package simple name*>} ;
<*variant part*> ::= **case** <*discriminant simple name*> **is**
<*variant*> {<*variant*>}
end case ;
<*variant*> ::= **when** <*choice*> {‖ <*choice*>} **=>** <*component list*>
<*with clause*> ::= **with** <*library unit simple name*>
{ **,** <*library unit simple name*>} ;

æ

Feladatok

Az alábbi feladatok az Ada programozási nyelv használatának a gyakorlására szolgálnak. A feladatok többsége algoritmikus szempontból egyszerű, nehézséget csak az absztrakt programnak a nyelvre való átültetése okozhat.

A feladatokat osztályokba soroltuk aszerint, hogy a kódolás milyen nyelvi elemek használatát igényli. Ezt jelzi a feladat sorszáma után feltüntetett ①, ② stb. jel. Azokat a feladatokat, amelyek megoldása valamivel nehezebb, egy csillaggal (① *), azokat pedig, amelyek hosszabb gondolkodást igényelnek, két csillaggal (① **) jelöltük meg.

1. ① Készítsünk egy programot, ami kiszámítja a π közelítő értékét az alábbi képlet alapján!

   a) $\frac{\pi}{2} = \frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdot \ldots$

   b) $\pi = 4(1 - \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \ldots)$

   c) $\pi = 3 + 4(\frac{1}{2 \cdot 3 \cdot 4} - \frac{1}{4 \cdot 5 \cdot 6} + \frac{1}{6 \cdot 7 \cdot 8} - \ldots)$

2. ① Készítsünk egy programot, ami ε pontossággal kiszámítja az alábbi közelítő értékét!

   a) $ln\ 2 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} + \ldots$

   b) $ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots$

   c) $ln\ x = 2\ (\frac{x - 1}{x + 1} + \frac{(x - 1)^3}{3(x + 1)^3} + \frac{(x - 1)^5}{5(x + 1)^5} + \ldots)$

3. ① Készítsünk egy programot, ami kinyomtatja a következő összeg első N részletösszegét!

   a) $\sum_{n=1}^{\inf} \frac{n!}{n\sqrt{n}}$

   b) $\sum_{n=1}^{\inf} \frac{((n+1)!)^n}{2!4! \ldots (2n)!}$

4. © Függvények egy sorozatát az alábbi módon definiáljuk: $u_1(t) = 1$, $u_2(t) = t$, $u_{k+1}(t) = 2tu_k(t) - u_{k-1}(t)$ $(k = 2, 3, ..., n-1)$. Készítsünk programot, ami adott n és t értékekre meghatározza $u_n(t)$ értékét!

5. © Készítsünk olyan eljárást, ami az adott n, h egész és y_0 valós értékekből meghatározza y_n értékét az alábbi formula alapján:

$$y_{i+1} = y_i + \frac{1}{4}k_i + \frac{3}{4}m_i \quad (0 \le i < n),$$

$$k_i = h(y_i^2 + y_i + 1),$$

$$m_i = h((y_i + \frac{2}{3}k_i)^2 + y_i + \frac{2}{3}k_i + 1)$$

6. © Adott a valós A_N vektor. Határozzuk meg S értékét az alábbi formula alapján:

$$a) \quad S = \frac{1}{N-1}\sum_{i=1}^{N}(a_i - \frac{1}{N}\sum_{j=1}^{N}a_j^2)$$

$$b) \quad S = \sqrt{\frac{1}{N}\sum_{i=1}^{N}a_i^2 - (\frac{1}{N}\sum_{j=1}^{N}a_j)^2}$$

7. © Készítsünk programot az alábbi összeg kiszámítására (a_0, b_0 és ε adott kezdőértékek):

$$S = 1 - \frac{1}{2}\sum_{n=0}^{a_n^2 - b_n^2 < \varepsilon} 2^n(a_n^2 - b_n^2), \quad ahol$$

$$a_n = \frac{1}{2}(a_{n-1} + b_{n-1}), \quad b_n = \sqrt{a_{n-1}b_{n-1}}$$

8. © Adottak a valós X_n és Y_n vektorok. Készítsünk programot az a_k együtthatók kiszámítására az alábbi formula alapján:

$$a_k = \frac{4}{n\pi}\sum_{i=1}^{n}\frac{y_i u_k(x_i)}{\sqrt{1 - x_i^2}}, \quad k = 1, ..., n,$$

$$u_1(z) = 1, \quad u_2(z) = z, \quad u_{k+1}(z) = 2zu_k(z) - u_{k-1}(z), \quad k = 2, ..., n$$

9. © Készítsünk programot az alábbi összeg kiszámítására:

$$a) \quad S = 1 + x + \frac{x^2}{2!} - \frac{3x^4}{4!} + ... + (-1)^{N-1}\frac{(2N-1)x^{2N}}{(2N)!},$$

$$b)\quad S = \frac{2}{3}sin\ 2x - \frac{3}{8}sin\ 3x + ... + (-1)^N \frac{N}{N^2 - 1}sin\ Nx$$

10. ☺ Készítsünk olyan programot, ami kiszámítja az alábbi tört értékét!

$$\cfrac{1}{1+\cfrac{1}{3+\cfrac{1}{\ldots n+\cfrac{1}{n+1}}}}$$

11. ☺ Készítsünk olyan programot, ami ε pontossággal kiszámítja $\sqrt[k]{x}$ értékét az alábbi rekurzív formula alapján :

$$a)\quad y_0 = x, \quad y_{n+1} = \frac{1}{k}((k-1)y_n + \frac{x}{y_n^{k-1}})$$

$$b)\quad y_0 = x, \quad y_{n+1} = \frac{y_n}{k^2}((k^2-1) + \frac{1}{2}(k+1)\frac{x}{y_n^k} - \frac{1}{2}(k-1)\frac{y_n^k}{k})$$

12. ☺ Adottak egy valós vektorban egy polinom együtthatói.
a) Készítsünk olyan programot, ami kiszámítja a deriváltjának együtthatóit!
b) Készítsünk olyan programot, ami kiszámítja a integráljának együtthatóit!
c) Készítsünk olyan programot, ami kiszámítja a négyzetének együtthatóit!

13. ☺ Készítsünk olyan eljárást, amelynek bemenő paraméterei egy m-ed és egy n-ed rendű polinom együtthatóit tartalmazó vektorok, kimenő paramétere egy $m+n$ hosszúságú vektor, s az eljárás kiszámítja a szorzatpolinom együtthatóit!

14. ☹ Készítsünk olyan modult, ami polinomokra vonatkozó eljárásokat exportál (polinomok összeadása, kivonása, számmal való szorzása stb.)!

15. ☺ Adott a valós A szám. Készítsünk olyan programot, ami meghatározza az S és N számokat úgy, hogy $S * N$ minimális legyen, és $|\frac{S}{N} - A| < \varepsilon$ teljesüljön!

16. ☺ Készítsünk olyan programot, ami ε pontossággal kiszámítja $tg\ z$ értékét az alábbi rekurzív formula alapján (feltesszük, hogy $z \neq \frac{\pi}{2} \pm n\pi (n = 1, 2, ...)$) :

$$tg\ z = \frac{z}{1 - y_3}, \qquad y_i = \frac{z^2}{i - y_{i+2}}$$

17. ☺ Készítsünk olyan programot, ami kiszámítja C_n^p értékét az alábbi rekurzív formulák alapján:

$$C_1^1 = 1, \quad C_n^{p+1} = \frac{n-p}{p+1}C_n^p, \quad C_{n+1}^p = C_n^{p-1} + C_n^p$$

18. ☹ Készítsünk generic package-t a komplex számok kezelésére! Az elemi aritmetikai operációkon kívül készítsünk függvényeket az alábbiak kiszámítására

is (a egy komplex számot jelöl, (r, ϕ) az a polárkoordinátás alakja, n egy egész szám):

$$\sqrt[n]{a} = \sqrt[n]{r} \cdot (cos\frac{\phi + 2k\pi}{n} + i \cdot sin\frac{\phi + 2k\pi}{n}), \quad (k = 0, ..., n - 1)$$

$$a^n = e^{n \cdot ln\ a}$$

$$sin\ a = a - \frac{a^3}{3!} + \frac{a^5}{5!} - \frac{a^7}{7!} + \dots$$

$$cos\ a = 1 - \frac{a^2}{2!} + \frac{a^4}{4!} - \frac{a^6}{6!} + \dots$$

19. ☺ Készítsünk egy olyan modult, ami egész számpárokkal ábrázolt valós számok műveleteit tartalmazza! Az (a, b) szám értéke $a \cdot 10^b$, tehát pl. az (a, b) és a (c, d) számok szorzata $(a \cdot c, b + d)$.

20. ☹ Készítsünk olyan programot, ami meghatározza azoknak az egész koordinátájú pontoknak a számát, amelyek az $\frac{y^2}{16} + \frac{x^2}{25} + 1 = 0$ ellipszisbe esnek!

21. ☹ Készítsünk olyan programot, ami szögfüggvénytáblázatot nyomtat!

22. ☹ Készítsünk egy olyan programot, ami kinyomtatja az alábbi függvény értékeinek táblázatát az $[a, b]$ intervallumban 0.5 lépésközzel!

$a)$ $f(x) = \frac{x}{(1+x)^2}$

$b)$ $f(x) = \frac{1+x}{1+x^3}$

$c)$ $f(x, n, p) = x^n p^x (1 - p)^{n-x}$

23. ☹ Ha egy testet elengedünk, szabadon esni kezd. Az elengedéstől számított t. másodpercben a kiindulási ponttól mért távolságát lábban az $s = 16 * t^2$ képlet adja (1 láb=30.48 cm). Nyomtassunk táblázatot az 1–20 másodpercekről, és ebben adjuk meg a test által megtett utat cm-ben!

24. ☹ Készítsünk egy olyan programot, ami a tanult közelítő eljárások valamelyikének felhasználásával megkeresi az alábbi egyenlet egy gyökét (pl. intervallumfelezés, húrmódszer)!

$a)$ $x^3 - 2x + 1.2 = 0$, $a\ [-2, -1]\ intervallumban$

$b)$ $x^5 - 0.5 = 0$, $a\ [0, 1]\ intervallumban$

$c)$ $e^x - x^2 + 1 = 0$, $a\ [-2, -1]\ intervallumban$

$d)$ $e^x - x^2 - 2 = 0$, $a\ [0.5, 0.6]\ intervallumban$

25. ☺ Készítsünk egy olyan programot, ami a tanult közelítő eljárások valamelyikének felhasználásával kiszámolja az alábbi függvény határozott integrálját (pl. Simpson-módszer, trapéz formula)!

   a) $\int_{-1}^{1} (1 + sin\ x)^2$

   b) $\int_{-1}^{1} x^2 + \sqrt{x^2 + 0.1}$

   c) $\int_{-1}^{1} \frac{x}{1 + x^2}$

   d) $\int_{-2}^{-1.5} x^3 - 3x^2 - x + 9$

26. ☹ Készítsünk programot, ami Monte-Carlo módszerrel kiszámítja az $(x - 0.5)^2 + (y - 0.5)^2 < 0.25$ körlap területét! (Állítsunk elő kb. 200 pontot egyenletes eloszlással véletlenszerűen a $[-1,1] \times [-1,1]$, tartományban, majd vizsgáljuk meg, hogy a pontok közül hány esik a körlapra.)

27. ☹ Készítsünk programot, ami Monte-Carlo módszerrel kiszámítja egy függvény határozott integrálját! (Foglaljuk téglalapba a függvényt az intervallum határai és a függvény egy alsó- és egy felső korlátja segítségével. Állítsunk elő egyenletes eloszlással véletlenszerűen pontokat, s állapítsuk meg, hogy a függvénygörbe alatt vagy felett vannak-e. Ezután egy osztással megkapható a görbe alatti terület közelítése.)

28. ☹ Készítsünk egy olyan programot, ami a tanult eljárások valamelyikének felhasználásával megoldja az alábbi egyenletet!

   a) $y_{\prime} = x + cos\frac{y}{\pi},\ x_0 = 1.7,\ y_0 = 5.3,\ az\ [1.7, 5.2]\ intervallumban$

   b) $y_{\prime} = x - y,\ x_0 = 0,\ y_0 = 0,\ a\ [0,1]\ intervallumban$

   c) $y_{\prime} = \frac{y}{e^x + y^2},\ x_0 = 0,\ y_0 = 1,\ a\ [0,1]\ intervallumban$

29. ☹ Készítsünk egy olyan programot, ami a Runge-Kutta módszer felhasználásával megoldja az alábbi egyenletet!

   a) $y_{\prime} = 0.25y^2 + x^2,\ y(0) = -1,\ a\ [0, 0.5]\ intervallumban$

   b) $y_{\prime} = \frac{y}{x} - y^2,\ y(0) = 1,\ az\ [1,2]\ intervallumban$

30. ☺ Készítsünk egy olyan programot, ami az Euler-módszer felhasználásával megoldja az alábbi egyenletet!

   a) $y_{\prime} = \frac{1}{2}xy,\ y(0) = 1,\ a\ [0,1]\ intervallumban$

b) $y' = 1 + xy^2,\ y(0) = 0,\ a\ [0,1]\ intervallumban$

c) $y' = ax + by + c,\ ahol\ a, b, c, y(0)\ valamint\ az\ intervallum\ adottak$

31. ⓓ Készítsünk egy olyan programot, ami a Newton-módszer ($x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$) felhasználásával megoldja az alábbi egyenletet!

a) $log_{10}x - \frac{1}{x^2} - 1 = 0, \quad ha\ x_0 = 4$

b) $cos\ x - 3x = 0, \quad ha\ x_0 = 0.3$

c) $e^x - 2x - 21 = 0, \quad ha\ x_0 = 3$

32. ⓔ Készítsünk egy olyan programot, ami a Seidel-módszer felhasználásával megoldja az alábbi egyenletrendszert!

a)
$$\begin{array}{rcrcrcl} 4x_1 & + & 0.24x_2 & - & 0.08x_3 & = & 8, \\ 0.09x_1 & + & 3x_2 & - & 0.15x_3 & = & 9, \\ 0.04x_1 & - & 0.08x_2 & + & 4x_3 & = & 20. \end{array}$$

b)
$$\begin{array}{rcrcrcl} 10x_1 & + & x_2 & + & x_3 & = & 12, \\ 2x_1 & + & 10x_2 & + & x_3 & = & 13, \\ 2x_1 & + & 2x_2 & + & 10x_3 & = & 14. \end{array}$$

33. ⓔ Készítsünk egy olyan programot, ami a Gauss-módszer felhasználásával megoldja az alábbi egyenletrendszert!

a)
$$\begin{array}{rcrcrcrcl} 7.9x_1 & + & 5.6x_2 & + & 5.7x_3 & - & 7.2x_4 & = & 6.68, \\ 8.5x_1 & - & 4.8x_2 & + & 0.8x_3 & + & 3.5x_4 & = & 9.95, \\ 4.3x_1 & + & 4.2x_2 & - & 3.2x_3 & + & 9.3x_4 & = & 8.6, \\ 3.2x_1 & - & 1.4x_2 & - & 8.9x_3 & + & 3.3x_4 & = & 1. \end{array}$$

b)
$$\begin{array}{rcrcrcl} 6x_1 & - & x_2 & - & x_3 & = & 11.33, \\ -x_1 & + & 6x_2 & - & x_3 & = & 32, \\ -x_1 & - & x_2 & + & 6x_3 & = & 42. \end{array}$$

34. ⓔ Készítsünk olyan programot, ami eldönti, hogy az $x_1 = (5, -3, 2, 4), x_2 = (2, -1, 3, 5), x_3 = 4, -3, 5, -7)$ vektorok lineárisan függetlenek-e! (A feladat visszavezethető a $\lambda_1 x_1 + \lambda_2 x_2 + \lambda_3 x_3 = 0$ egyenlet megoldására. Használjuk erre a Gauss-Jordan módszert!)

35. ⓓ Készítsünk olyan programot, ami kiírja adott A_0 tőke 1,2,...,10 évi kamatos kamattal növelt értékeinek táblázatát (p a kamatláb) :

$$A = A_0(1 + \frac{p}{100})^n$$

36. ☺ Készítsünk egy olyan programot, ami kinyomtatja egy adott év adott hónapjának naptárát!

37. ☺ Az egyetemen a pénztár olyan címletekben fizeti ki a hallgatók ösztöndíjait a csoportvezetőnek, hogy az mindenkinek odaadhassa az ösztöndíjat további pénzváltás és -visszaadás nélkül, s közben a pénztár a lehető legkevesebb bankjegyet és érmét adja ki. Készítsük el a címletezés programját, aminek adata a csoport tagjainak ösztöndíjai, eredménye az, hogy hány darab 1000-, 500-, 100-, 50-, 20-, 10-, 5-, 2- és 1-forintost kell a csoportvezetőnek kiadni!

38. ☺ Adott egy 1582 és 4902 közötti dátum (év, hónap, nap). Készítsünk programot, ami megállapítja, hogy ez hányadik nap az évben!

39. ☺ Készítsünk programot, ami megállapítja, hogy egy adott dátum hányadik munkanap az évben, ill. megfordítva, ami meg tudja adni, hogy az n. munkanap az év melyik napjára esik! (Az ünnepnapok egy része — pl. a karácsony — ismert, mások egy táblázatban adottak.)

40. ☹ Készítsünk egy modult a dátum (év, hónap, nap) típus megvalósítására, a dátumok közötti aritmetikai műveletek (dátum + valahány nap, két dátum között eltelt napok száma, két dátum közül melyik a korábbi stb.) elvégzésére!

41. ☺ Készítsünk programot, ami meghatározza egy adott évben az összes olyan pénteki napot, ami 13-ára esik!

42. ☺ Készítsünk programot, ami a természetes számok számjegysorozatában meghatározza az n. számjegyet!

43. ☺ Készítsünk programot, ami megszámolja, hogy az m és n között elhelyezkedő természetes számok összesen hány számjegyből állnak!

44. ☺ Készítsünk olyan programot, ami meghatározza azokat a háromjegyű természetes számokat, amelyek számjegyeinek köbösszege kiadja magát a számot!

45. ☹ Készítsünk olyan programot, ami meghatározza azokat a természetes számokat, amelyekre teljesül, hogy felírásukban és négyzetük felírásában együttesen a 0 -t kivéve minden számjegy szerepel, de csak egyszer!

46. ☺ Készítsünk olyan programot, ami meghatározza azokat a d_i természetes számokat, amelyekre teljesül, hogy $d_n \neq 0$ és $\forall i \in 0,...,n0 \leq d_i \leq (i+1)!$, valamint $\sum_{i=0}^{n} d_i \cdot (i+1)!$ egy előre megadott számmal egyenlő!

47. ☹ Készítsünk programot, ami egy adott N természetes számot felbont két prímszám összegére (nem bizonyított, hogy ez mindig lehetséges)!

48. ☹ Bármely természetes szám előállítható legfeljebb négy természetes szám négyzetösszegeként. Készítsünk programot, ami egy adott N természetes

számhoz megadja azokat a természetes számokat, amelyek négyzetösszegeként előállítható!

49. ✆ Készítsünk programot, ami egy adott N természetes számról megállapítja, hogy előállítható-e három természetes szám négyzetösszegeként, s ha igen, akkor felsorolja az összes ilyen előállítást!

50. ✆ Készítsünk programot, ami egy adott N természetes számhoz megadja az N -nél kisebb $2^p - 1$ alakú természetes számokat, ahol p prímszám!

51. ✆ Készítsünk programot, ami meghatározza azt a legnagyobb N természetes számot, amire $N! \leq 10^{50}$!

52. ✆ Tökéletesnek nevezzük azokat a természetes számokat, amelyek osztóinak (az 1 -et beleértve, de önmagát nem) összege kiadja magát a számot. Készítsünk programot, ami M és N között megadja az összes tökéletes számot!

53. ✆ Ismert, hogy az m és n nemnegatív egész számok ($m \geq n$) legnagyobb közös osztója meghatározható az alábbi rekurzív függvény értékének kiszámításával:

$$lnko(m,n) = \begin{cases} m & , \; ha \; n = 0 \\ lnko(n, [\frac{m}{n}]) & , \; ha \; n \neq 0 \end{cases}$$

a) Készítsünk programot két természetes szám legnagyobb közös osztójának meghatározására!
b) Készítsünk programot n db természetes szám legnagyobb közös osztójának meghatározására! ($lnko(a_1, ..., a_n) = lnko(lnko(a_1, ..., a_{n-1}), a_n)$.)
c) Készítsünk programot két természetes szám legkisebb közös többszörösének meghatározására!

54. ✆ Adott az N természetes szám. Készítsünk programot, ami megállapítja, hogy
a) igaz-e, hogy a számban minden számjegy különböző!
b) igaz-e, hogy a számban egy számjegy háromszor szerepel, a többi számjegy pedig mind különböző!

55. ✆ Készítsünk programot egy személyi szám ellenőrző jegye helyességének megállapítására! A személyi számok utolsó jegye az ellenőrző jegy, amit az alábbi képlettel határoztak meg (10-es maradékot adó személyi számot nem adtak ki):

$$(x_1 + 2 \cdot x_2 + 3 \cdot x_3 + \ldots + 10 \cdot x_{10}) \; mod \; 11$$

56. ✆ Készítsünk programot, ami prímtényezőire bontja az N természetes számot (vagyis meghatározza azokat az I, J, K, L, P, Q, R, S természetes számokat, amelyekre $N = I^P J^Q K^R ... L^S$)!

57. ✆ Adott egy 1-nél nagyobb természetes szám. Ha páros, akkor felezzük, különben szorozzuk meg hárommal, majd adjunk hozzá egyet, s ismételjük

ezt a műveletet, amíg 1 -et nem kapunk! Készítsünk programot, ami a fenti eljárást alkalmazza, s megadja, hogy hány lépés után jutott el az 1-hez! (Nem bizonyított, hogy ez az eljárás minden természetes számra véges.)

58. ② Készítsünk programot, ami meghatározza az $a_0, a_1, ..., a_n$ számsorozat elemeit ($a_i \in \{-1, 0, 1\}, a_n \neq 0$) úgy, hogy $\sum_{i=0}^{n} a_i 3^i$ egy előre megadott érték legyen!

59. ② Készítsünk programot, ami előállít egy olyan természetes számot, aminek decimális alakjában csak a 0 és a 7 számjegyek fordulnak elő, s a szám osztható egy előre megadott természetes számmal!

60. ③ Készítsünk modult tetszőleges elemekből álló sorozat típus megvalósítására!

61. ② Készítsünk programot lineáris egyenletrendszer megoldására a trianguláris felbontás módszerével!

62. ③ Készítsünk olyan modult, ami egy tetszőleges elemekből álló vektor rendezésére exportál különböző eljárásokat (quicksort, heapsort stb.)!

63. ③ Készítsünk olyan modult, ami numerikus elemekből álló vektorokon értelmezett műveleteket exportál (összeadás, szorzás, norma számolása, merőlegesség ellenőrzése stb.)!

64. ③ Készítsünk olyan modult, ami egy általános mátrix típusra exportálja a gyakoribb mátrixműveleteket (mátrixok összeadása, kivonása, szorzása, számmal való szorzása, mátrixokon értelmezett relációk stb.)!

65. ③ Készítsük el a főátlóra szimmetrikus mátrixok típusának egy hely szempontjából gazdaságos reprezentációját, s írjuk meg a típusműveletek alprogramjait (indexelés, mátrixok összeadása, kivonása, szorzása, számmal való szorzása, mátrixokon értelmezett relációk stb.)!

66. ③ Készítsük el egy ritka (hézagosan kitöltött) mátrix típusnak a hely szempontjából gazdaságos reprezentációját, s írjuk meg a típusműveletek alprogramjait (indexelés, mátrixok összeadása, kivonása, szorzása, számmal való szorzása, mátrixokon értelmezett relációk stb.)!

67. ① Készítsünk olyan programot, ami meghatározza egy vektor érték szerint középső elemét!

68. ① Adott egy egész számokból álló vektor és egy K egész szám. Készítsünk olyan programot, ami átrendezi a vektor elemeit úgy, hogy a K-nál kisebb elemek állnak elöl!

69. ① Készítsünk egy programot, ami egy mátrixban felcserél két sort és két oszlopot úgy, hogy a mátrix maximális eleme az első sor első eleme legyen!

70. ☺ Készítsünk olyan eljárást, ami a paraméterként kapott négyzetes mátrixot az óra járása szerint 90 fokkal elforgatja!

71. ☺ Készítsünk olyan programot, ami a $2n$ méretű A mátrixot blokkokra osztja $A = \begin{matrix} A_1 & A_2 \\ A_3 & A_4 \end{matrix}$ alakban, majd a blokkokat felcseréli: A_1 -et A_4 -el, A_2 -t pedig A_3 -mal!

72. ☺ Készítsünk olyan eljárást, ami az adott X_n vektorról eldönti, hogy az az 1,...,n számok permutációja-e!

73. ☺ Az A_N vektor természetes számokat tartalmaz. Határozzuk meg azoknak az elemeknek az indexét, amelyek összege megegyezik egy előre adott értékkel!

74. ☺ Az A_N vektor természetes számokat tartalmaz. Határozzuk meg azoknak az elemeknek az indexét, amelyek (a vektorbeli sorrendjükben) a leghosszabb növekvő sorozatot alkotják!

75. ☺ Készítsünk olyan eljárást, ami az adott X_n vektor ismeretében meghatározza a P_n vektor értékét úgy, hogy P az 1,...,n számok permutációja legyen és teljesüljön $x_{p_i} \leq x_{p_{i+1}}$ $(i = 1, ..., n-1)$!

76. ☹** Adott egy valós mátrix és egy R valós szám. Bontsuk a mátrixot a lehető legkevesebb részmátrixra úgy, hogy egy részmátrixon belül bármely két elem különbsége kisebb legyen R -nél!

77. ☹ Készítsünk programot, ami kiszámolja egy mátrix sajátértékeit és sajátvektorait a Jacobi-módszer alapján!

78. ☺ Készítsünk generic package-ket különféle numerikus módszerekre!

79. ☺ Készítsünk olyan programot, ami egy négyzetes mátrixot felbont két mátrix összegére úgy, hogy az egyik tag szimmetrikus, a másik ferdén szimmetrikus legyen!

$$A = S + T, \quad S = \frac{1}{2}(A + A^*), \quad T = \frac{1}{2}(A - A^*)$$

80. ☺ Készítsünk olyan programot, ami egy négyzetes mátrixról eldönti, hogy az elemek átrendezésével szimmetrikus mátrixszá tehető-e!

81. ☺ Legyen $A^{(1)}, A^{(2)}, ...$ négyzetes bináris mátrixok egy sorozata, ahol $A^{(i)}$ dimenziója 3^i. Készítsünk programot az $A^{(i)}$ mátrix előállítására, ha

$$A^{(1)} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}, \qquad A^{(i)} = \begin{pmatrix} A^{(i-1)} & 0 & A^{(i-1)} \\ 0 & A^{(i-1)} & 0 \\ A^{(i-1)} & 0 & A^{(i-1)} \end{pmatrix} !$$

82. ⓓ* Adott az egészekből álló C_{mn} mátrix és a D_m vektor. Készítsünk programot annak a csak 0 és 1 számokat tartalmazó X_n vektornak az előállítására, amelyre $C \cdot X = D$!

83. ⓓ Adott az S_N számsorozat. Egészítsük ki a sorozatot további M darab szám hozzávételével $(s_{N+1}, ..., s_M)$ úgy, hogy teljesüljön az alábbi feltétel: $s_1 = s_M, s_2 = s_{M-1}$, ...!

84. ⓒ Adott a síkon koordinátáival N darab pont. Kössük össze a pontokat egy olyan vonallal, ami mindegyik ponton (pontosan egyszer) átmegy, s nem metszi önmagát!

85. ⓓ Az $y = ax + b_1$ és az $y = ax + b_2$ egyenesek a síkot három tartományra osztják. Készítsünk olyan programot, ami megállapítja, hogy adott N darab pont közül melyik hányadik tartományba esik!

86. ⓓ Adott a síkon N db kör. Készítsünk olyan programot, ami megállapítja, hogy a körök közös része üres-e!

87. ⓓ Adott végpontjai koordinátáival a síkon két szakasz. Határozzuk meg a metszéspont koordinátáit!

88. ⓓ Adott koordinátáival a sík N pontja. Rakjuk sorba a pontokat a rajtuk és az origón áthaladó egyenesnek az x - tengellyel bezárt szöge szerint növekvő sorrendbe!

89. ⓒ Adott koordinátáival a sík N pontja. Határozzuk meg a ponthalmaz konvex burkát!

90. ⓒ Adott koordinátáival a sík N pontja. Határozzuk meg a ponthalmazhoz a legkisebb olyan négyzetet, amelybe mindegyik pont beleesik!

91. ⓒ Adott koordinátáival a sík N pontja. Készítsünk programot, ami kiválaszt három olyan pontot, amelyek (legfeljebb ε eltérést megengedve) egy egyenesre esnek!

92. ⓓ Adott a síkon csúcspontjai koordinátáival egy sokszög (a csúcspontokat a kerületen az óra járása szerint elfoglalt helyük sorrendjében adjuk meg). Készítsünk programot, ami eldönti, hogy a sokszög konvex-e!

93. ⓓ Készítsünk egy olyan programot, ami egy alfanumerikus képernyőre valamilyen jelekből (pl. csillagokból) egy adott sugarú kört rajzol!

94. ⓓ Készítsünk egy olyan programot, ami egy alfanumerikus képernyőre valamilyen jelekből (pl. csillagokból) egy olyan egyenest rajzol, ami áthalad az origón, s aminek pontjai egyenlő távolságra vannak az A és a B ponttól!

95. ⓓ Készítsünk egy olyan programot, ami egy alfanumerikus képernyőre vala-

milyen jelekből (pl. csillagokból) kirajzolja egy függvény grafikonját!

a) $y = \dfrac{x^4 + 8}{x^3 + 1}$, *a* $[-1, 1]$ *intervallumban*

b) $y = sin\ x^2$, *a* $[0, \pi]$ *intervallumban*

96. ✆ Egy véletlenszámgenerátort ellenőrzünk abból a szempontból, hogy a (0,1) intervallumban eléggé egyenletesen generálja-e a számokat. Ehhez az intervallumot k egyenlő részre osztjuk, majd generálunk n db véletlenszámot. Rajzoljunk hisztogramot!

97. ✆ Ha r_i-k egyenletes eloszlású véletlenszámok a $[-1, 1]$ intervallumban, akkor az alábbi képletekkel megadott pontok egyenletes eloszlásúak az egységgömbön. Készítsünk programot, ami ellenőrzi, hogy a pontok eloszlása valóban egyenletes-e!

$$x = \frac{2(r_2 r_4 + r_1 r_3)}{r_1^2 + r_2^2 + r_3^2 + r_4^2},$$

$$y = \frac{2(r_3 r_4 - r_1 r_2)}{r_1^2 + r_2^2 + r_3^2 + r_4^2},$$

$$z = \frac{r_1^2 + r_4^2 - r_2^2 - r_3^2}{r_1^2 + r_2^2 + r_3^2 + r_4^2},$$

$$ahol \sum_{i=1}^{4} r_i^2 < 1.$$

98. ✆ Adott a síkon szemközti csúcsai koordinátáival egy téglalap, amelynek oldalai párhuzamosak a koordináta-tengelyekkel, és adott a végpontjai koordinátáival egy szakasz. Készítsünk olyan programot, ami kiszámítja a szakasznak azt a darabját, ami a téglalapon belülre esik (lehet, hogy a teljes szakasz kívül esik a téglalapon)!

99. ✆* Rendelkezésünkre áll a Balaton vízfelszíne igen sok pontjának azonos időben mért hőmérséklete. Készítsünk programot a hőmérsékletek szintvonalas ábrázolására!

100. ✆ Készítsünk olyan modult, ami szövegben egy adott szöveg megkeresésére különböző eljárásokat exportál!

101. ✆ Készítsünk olyan modult, ami új típusműveleteket ad a `STRING` típushoz (karakter vagy string keresése stringben, az összes szóköz kihagyása a stringből, kivéve azokat, amelyek nem szóköz után következnek, stringek összehasonlítása a szóközök figyelmen kívül hagyásával, stringbe egy string beszúrása, stringből egy részstring elhagyása stb.)!

102. ☺ Készítsünk rekurzív eljárást, ami
a) egy szövegből kihagyja a szóközöket!
b) egy szövegben megszámolja a magánhangzókat!

103. ☺ Készítsünk programot, ami egy 4000-nél kisebb természetes számot római számmá alakít át!

104. ☺ Készítsünk olyan modult, ami római számokon végez el aritmetikai műveleteket!

105. ☺ Készítsünk programot, ami egy szövegben megkeresi a magas-, mély- és vegyes hangrendű szavakat!

106. ☹* Készítsünk programot, ami egy verset úgy nyomtat ki, hogy bejelöli az időmértéket!

107. ☹** Készítsünk egy interaktív programot, ami a beolvasott szavakat szótagolva írja vissza!

108. ☹* Készítsünk egy olyan programot, ami egy természetes számot betűkkel kiír! (Tehát pl. 241 esetén a Kettőszáznegyvenegy szöveget nyomtatja.)

109. ☺ Az egyik legegyszerűbb titkosírást az ABC betűinek ciklikus eltolásából kaphatjuk meg. Készítsünk olyan programot, ami egy szöveget kódol úgy, hogy minden betű helyébe az ABC-ben nála adott hellyel hátrább levő betűt teszi!

110. ☹ Egy titkosírást a következő módon készítünk el: helyezzünk el betűket egy négyzetes mátrixban, s erre tegyünk rá egy négyzetrácsos lapot, amelynek négyzetei egy-egy betű fölé esnek, s amelynek néhány négyzete ki van vágva. A látható betűk adják a megfejtés első részét. Ezután fordítsuk el ezt a lapot 90 fokkal jobbra, és folytassuk a betűk leolvasását. Ezt a műveletet még kétszer megismételve, megkapjuk a teljes szöveget. Készítsünk egy programot, ami a fenti módon megfejt egy titkosított szöveget!

111. ☹ A titkosírás egy módja a következő is: írjuk be a szöveget mondjuk sorfolytonosan egy mátrixba, majd a középső elemtől elindulva és spirálisan kifelé tartva járjuk be a mátrix elemeit. Az így kapott karaktersorozat a titkosított szöveg. Készítsünk egy programot, ami a fenti módon titkosít és megfejt egy szöveget!

112. ☹ Készítsünk egy programot, ami egy véletlenszámgenerátor felhasználásával titkosít egy szöveget! Inicializáljuk a véletlenszámgenerátort egy olyan értékkel, amit megjegyzünk, s amire szükségünk lesz majd a megfejtéskor is. Állítsunk elő pontosan annyi egyenletes eloszlású véletlenszámot 1 és 4 között, mint amilyen hosszú a szöveg. Ezután a szöveg minden karakterét cseréljük fel azzal a karakterrel, amit úgy kapunk, hogy a neki megfelelő véletlenszámot hozzáadjuk. Megfejtéskor ugyanazt a véletlenszámgenerátort

kell használnunk, s ugyanazzal az értékkel kell inicializálni.

113. ◐* Egy text-file-ban a szöveget üres sorokkal tagoltuk bekezdésekre. Készítsünk olyan programot, ami úgy nyomtatja a file tartalmát, hogy az egyes bekezdések első sora előtt három szóközt hagy, s a bekezdéseken belül a sorokat egyformán 60 hosszúságúra hozza (ez történhet szavak áthelyezésével a következő sorba, a következő sorból szavak áthozásával, szóközök beszúrásával, esetleg szavak elválasztásával)!

114. ◑ Készítsünk programot, ami egy karaktersorozatról eldönti, hogy az a Modula-2 szintaxisa szerint egy szabályos felsorolási típus definíció-e!

115. ◐ Adott egy input rekordban egy zárójelezett kifejezés (maga az egész kifejezés is zárójelben van).
a) Készítsünk olyan programot, ami eldönti, hogy a zárójelezés helyes-e!
b) Készítsünk olyan programot, ami megadja (kiírja) az összes zárójelezett részkifejezést!

116. ◑ Készítsünk olyan programot, ami előállítja az összes helyes N hosszúságú zárójelezést! Helyes zárójelezés a (), továbbá, ha P és Q helyes zárójelezések, akkor helyes zárójelezés (P) és $(P)(Q)$ is.

117. ◑ Adott egy kifejezés, ami egybetűs változókból, zárójelekből és bináris operátorokból (+,-,*,/) áll. Készítsünk egy olyan programot, ami a kifejezést fordított lengyel formára hozza!

118. ◑* Adott egy kifejezés, ami egybetűs változókból, zárójelekből és bináris operátorokból (+,-,*,/) áll. Készítsünk egy olyan programot, ami kiszámítja a kifejezéshez az alábbiakban definiált az $S(K)$ értéket! Írjuk fel a kifejezés szintaxisfáját, jelölje a fát K, a baloldali részfát K_l, a jobboldalit K_r. Ha K az üres fa, akkor $S(K) = 0$, különben pedig a következő rekurzív formulával adjuk meg az értékét:

$$S(K) = \begin{cases} S(K_l) + 1 & , \; ha \; K_l = K_r \\ max(S(K_l), S(K_r)) & , \; \text{különben} \end{cases}$$

119. ◑* Adott egy kifejezés, ami egybetűs változókból, zárójelekből és bináris operátorokból (+,-,*,/) áll. Készítsünk egy olyan programot, ami kiszámítja a kifejezés szintaxisfájának magasságát!

120. ◑* Adott egy kifejezés szintaxisfájának preorder és inorder bejárása során kapott sorozat. Készítsünk programot, ami ebből előállítja a szintaxisfa postorder bejárásakor előálló szimbólumsorozatot!

121. ◑* Készítsünk olyan programot, ami beolvas egy logikai kifejezést, és kinyomtatja az igazságtábláját! A kifejezés egybetűs változókat, logikai operátorokat és (kerek) zárójeleket tartalmazhat. (Egyszerűbb a feladat, ha feltesszük, hogy a kifejezés szintaktikusan helyes.)

122. ✆ Adott egy szövegfile, amelynek a sorai nem haladják meg a 80 karaktert. Másoljuk át a file-t úgy, hogy a szavak közé szóközöket beszúrva minden sort kiegészítünk 80 hosszúságúra!

123. ✆ Adott egy file-ban egy szöveg, ami csak betűkből áll. Nyomtassuk a szöveget úgy, hogy minden betűt Morze-jelekkel ábrázolunk! A Morze-jelek között hagyjunk egy-egy szóközt, és egy szót nyomtassunk egy sorba!

124. ✆ Készítsünk egy interaktív szótárprogramot! A program számára lehessen definiálni egy angol szó jelentését, és lehessen visszakérdezni egy szó jelentését! Visszakérdezéskor engedjük meg, hogy a kérdezett szavakat rövidítsék! A definiált szavakat tároljuk lemezen! (A gyors visszakeresés érdekében használhatunk pl. kiegyensúlyozott bináris fát.)

125. ✆ Állapítsuk meg két stringről, hogy azok a szóközöktől eltekintve megegyeznek-e!

126. ✆ Adva van egy távirat szövege. Készítsünk programot, ami meghatározza a távirat továbbításáért fizetendő díjat!

127. ✆ Adva van egy névsor, amiben a nevek egy vezetéknévből, és valahány utónévből állnak. Készítsünk programot, ami úgy írja ki a neveket, hogy a vezetéknévnek csak a kezdőbetűje legyen nagy betű, a vezetéknév betűi között legyen egy-egy szóköz, majd utána egy vessző, s ezután csak az utónevek nagy kezdőbetűi álljanak, mindegyik ponttal lezárva!

128. ✆ Tegyük fel, hogy horizontális tabulátor karaktereket tartalmazó szöveget akarunk listázni olyan nyomtatón, ami nem ismeri a tabulátorokat. Készítsünk egy programot, ami előkészíti a szöveget a nyomtatáshoz: a tabulátor karaktereket a megfelelő számú szóközzel helyettesíti! Tegyük fel, hogy a tabulátor-pozíciók az 1, 9, 17, ... pozíciók. (Általánosabb lesz a megoldásunk, ha a tabulátor-pozíciók felsorolásával paraméterezhető a programunk.)

129. ✆ Az előző feladat "megfordítása" is értelmes: Készíthetünk olyan programot, amely egy szöveg-file-ban a hosszú szóköz-sorozatokat úgy rövidíti le, hogy tabulátor karakterekkel cserél fel valahány szóközt.

130. ✆ Készítsünk olyan modult, ami eljárásokat exportál egy infix formájú kifejezés postfix alakjának előállítására (a kifejezésben csak az aritmetikai alapműveletek jelei, valós számok és zárójelek szerepelnek), és egy postfix formájú kifejezés értékének kiszámítására!

131. ✆* Készítsünk egy nem interaktív szövegszerkesztő programot! A program egy file-ból beolvasott parancsok alapján módosítsa a text-file-t úgy, hogy az eredmény egy új file-ban keletkezzen. A módosító parancsok a következők (a módosító parancsokban a sorok sorszámait növekvő sorrendben kell megadni):

- #DEL *sorszám* { , *sorszám* } — törli a sorszámaikkal megadott sorokat;
- #INC *sorszám* — a megadott sorszámú sor után beszúrja a parancssor után a következő parancssorig megadott sorokat.

132. ⊕* Készítsünk egy olyan programot, ami szintezve (tehát a beágyazottságuknak megfelelően) kiírja egy Pascal program összes eljárásának nevét!

133. ⊕ Másoljunk át úgy egy szintaktikusan helyes Modula-2 modult, hogy a megjegyzéseket közben kihagyjuk belőle!

134. ⊕ Határozzuk meg, hogy egy bizonyos szó a Modula-2 forrásprogram mely soraiban fordul elő!

135. ⊕ Számoljuk meg egy szintaktikusan helyes Modula-2 implementációs modulban, hogy melyik alapszó hányszor szerepel!

136. ⊕ Gyűjtsük ki egy szintaktikusan helyes Modula-2 implementációs modulból a benne definiált eljárások és függvények neveit!

137. ⊕ Egy szintaktikusan helyes Pascal programban az alapszavak betűit változtassuk nagybetűkre, a megjegyzések és szövegliterálok betűit ne változtassuk, a többi betűt cseréljük kisbetűre!

138. ⊕ Készítsünk olyan programot, ami egy szintaktikusan helyes Ada programban megkeresi azokat a címkéket, amelyekre nincsen hivatkozás!

139. ⊕ Készítsünk programot, ami
a) elemez egy karaktersorozatot, s eldönti, hogy az egy szabályos Ada <*numeric literal*>-e!
b) megadja egy szabályos Ada <*numeric literal*> decimális értékét!
c) egy szabályos Ada <*numeric literal*>-nak kirajzolja a szintaxisfáját!

<*numeric literal*> ::= <*decimal literal*>|<*based literal*>
<*decimal literal*> ::= <*integer*> [. <*integer*>] [<*exponent*>]
<*integer*> ::= <*digit*> { [<*underline*>] <*digit*> }
<*exponent*> ::= E [*] <*integer*> | E - <*integer*>
<*based literal*> ::= <*base*> # <*based integer*>
[. <*based integer*>] # <*exponent*>
<*base*> ::= <*integer*>
<*based integer*> ::= <*extended digit*> { [<*underline*>] <*extended digit*>
<*extended digit*> ::= <*digit*>|<*letter*>

140. ⊕ Adott egy CDL-nyelvű szabály (programegység) törzsének (némileg leegyszerűsített) szintaxisa. Készítsünk programot, ami egy stringről eldönti, hogy az megfelel-e ennek a szintaxisnak!

<*szabály*> ::= <*csoport*> "."
<*csoport*> ::= <*alternatíva*>|<*alternatíva*> ";" <*csoport*>

<alternatíva> ::= <tag>|<tag> "," <alternatíva>
<tag> ::= "+" | "-" | "" | "*" <azonosító> |*
"(" [<címke>] <csoport> ")"

141. ◐ Adott az RSX-11M operációs rendszer parancsainak (némileg leegyszerűsített) szintaxisa. Készítsünk programot, ami egy stringről eldönti, hogy az megfelel-e ennek a szintaxisnak!

<parancs> ::= <parancskód> <paraméterek> "=" <paraméterek>|
<parancskód> <paraméter>
<paraméterek> ::= <paraméter>|<paraméter> <paraméterek>
<paraméter> ::= [<egység>] [<felhasználó>] <file>
<egység> ::= "DK" <szám> ":" | "LP" <szám> ":"
<felhasználó> ::= "[" <szám> "," <szám> "]"
<file> ::= <azonosító> [<toldalék>]
<toldalék> ::= <file-kiegészítő> [<verzió>]
<file-kiegészítő> ::= "." | "." <azonosító>| "." ""*
<verzió> ::= ";" | ";" "" | "." "*"*

142. ◐ Készítsünk egy telefonos információkat nyilvántartó programot! A program adjon lehetőséget a név és a telefonszám szerinti visszakeresésre is!

143. ◐* Készítsünk egy órarendet nyilvántartó programot!
a) A program tudjon választ adni bizonyos kérdésekre, pl. meg tudja adni egy tanár óráinak időpontját és helyét, vagy azt, hogy egy adott teremben, adott időben kinek van órája stb.!
b) A program tudja ellenőrizni azt, hogy az órarendben nincsen-e ellentmondás!

144. ◐ Készítsünk programot könyvtári kölcsönzések nyilvántartására és a nyilvántartás lekérdezésére!

145. ◐** Készítsünk egy olyan programot, ami egy vasúti menetrendet tárol, s lehetőséget ad annak lekérdezésére! (Bonyolultabb a feladat, ha a program az átszállással elérhető állomásokra is meg tudja adni az indulási és érkezési időket.)

146. ◐ Adott három file, ami egy öttusa-verseny eredményeit tartalmazza. Az első file rekordjai az indulók rajtszámait és nevét tartalmazzák. A második file a csapatok nevét és a csapatok három versenyzőjének a rajtszámát tartalmazzák. A harmadik file az elért eredményeket tartalmazza olymódon, hogy egy rekord a versenyző rajtszámából, a versenyszám megnevezéséből és az elért pontszámból áll. Készítsünk táblázatot az egyéni versenyről és a csapatok versenyéről!

147. ◐ A metrón utasszámlálást végeznek: az állomásokat megsorszámozzák, s minden állomáson minden utasnak a kezébe adnak egy kártyát az állomás sorszámával, a kiszálló utasoktól pedig a kártyákat összegyűjtik. Az összegyűjtött kártyákat minden óra végén egy-egy zsákba zárják. A

kártyák megszámlálása után ismerjük minden állomásra minden óra végén az egyes állomásokról érkezett utasok számát. Készítsünk olyan programot, ami különféle grafikonok nyomtatásával segíti az adatok kiértékelését!

148. ⓢ Készítsünk egy tetszőleges elemekből álló halmazt megvalósító modult (a halmazt reprezentálhatjuk pl. egy hash-táblával)!

149. ⓢ Készítsünk egy tetszőleges elemekből álló kétirányú listát megvalósító modult (Create, PutFirst, Put, PutLast, GetFirst, Get, GetNext, GetPrev, Overwrite, Seek, Delete, Concat, Isempty, Length műveletekkel)!

150. ⓢ Készítsünk egy tetszőleges elemekből álló rendező-fát megvalósító modult (create, isempty, write, first, read műveletekkel)!

151. ⓢ Készítsünk egy tetszőleges elemekből álló n-áris fát megvalósító modult!

152. ⓒ Készítsünk olyan programot, ami eldönti egy irányítatlan egyszerű gráfról, hogy van-e olyan köre, ami minden csúcson áthalad!

153. ⓓ Készítsünk olyan programot, ami eldönti egy irányítatlan egyszerű gráfról, hogy van-e olyan csúcsa, amit minden más csúccsal él köt össze!

154. ⓓ Készítsünk olyan programot, ami eldönti egy irányítatlan egyszerű gráfról, hogy az teljes gráf-e (vagyis, hogy bármely két csúcsa össze van-e kötve)!

155. ⓓ Készítsünk olyan programot, ami meghatározza egy irányítatlan egyszerű gráf komplementer gráfját (azaz: megadja, hogy milyen éleket kellene hozzáadni a gráfhoz, hogy teljes gráfot kapjunk)!

156. ⓒ** Készítsünk programot, ami meghatározza egy irányítatlan gráf összes teljes részgráfját!

157. ⓒ* Készítsünk programot, ami egy irányított gráfban megkeresi azt a csúcsot, amelyből minden más csúcs elérhető (úttal)!

158. ⓢ* Valósítsunk meg privát típusként egy irányított gráf típust!

159. ⓒ* Készítsünk programot, ami egy irányított gráfban, ahol az élekhez hosszértékeket is rendeltünk, megkeresi egy adott csúcsból a többi csúcsba vezető legrövidebb utakat!

160. ⓒ** Készítsünk olyan programot, ami eldönti egy irányítatlan egyszerű gráfról, hogy hányszorosan összefüggő!

161. ⓒ* Készítsünk olyan programot, ami egy gráfban keres egy olyan utat a p és q csúcsok között, ami nem halad át az x csúcson!

162. ⓒ* Készítsünk olyan programot, ami egy gráfban meghatároz egy olyan utat,

ami minden élet pontosan egyszer tartalmaz!

163. ✆* Készítsünk olyan programot, ami megkeresi egy irányítatlan egyszerű gráfban a minimális kört!

164. ✆** Készítsünk olyan programot, aminek adata két irányítatlan egyszerű gráf, s eldönti, hogy az első részgráfja-e a másodiknak!

165. ✆** Adott egy irányítatlan, de esetleg többszörös éleket tartalmazó síkbarajzolható gráf, ami egy kémiai szerkezet rajza. Készítsünk olyan programot, ami megkeresi azokat az atomokat, amelyek több gyűrűben is benne vannak!

166. ✆ Adott egy irányítatlan, de esetleg többszörös éleket tartalmazó síkbarajzolható gráf. Ez egy kémiai szerkezet rajza, ami csak a szénatomokat tartalmazza. Készítsünk olyan programot, ami meghatározza, hogy az egyes szénatomokhoz hány (a rajzon fel nem tüntetett) hidrogén kapcsolódik!

167. ✆** Adott egy irányítatlan, de esetleg többszörös éleket tartalmazó síkbarajzolható gráf, ami egy kémiai szerkezet rajza. Készítsünk olyan programot, ami megszámolja, hogy a szerkezet hány gyűrűt tartalmaz!

168. ✆* Készítsünk programot, ami meghatározza egy irányítatlan gráf egy feszítőfáját!

169. ✆* Adva van N darab elvégzendő feladat, s egy táblázat, amelyben feltüntettük, hogy melyik feladat elvégzése mennyi ideig tart, s melyek azok a feladatok, amiket csak a feladat befejezése után lehet elkezdeni. Készítsünk olyan programot, ami meghatározza, hogy az összes feladat elvégzése legkevesebb mennyi időt igényel!

170. ✆ Egy csoport elemeit az 1, a, b jelekkel ábrázoljuk, s adott a csoport szorzási táblája. Készítsünk programot, ami kiszámítja egy szorzat (az 1, a, b jelekből álló sorozat) eredményét!

$*$	1	a	b
1	1	a	b
a	a	b	1
b	b	1	a

171. ✆** Adott egy csoport a szorzási táblájával. Készítsünk programot, ami kiszámítja a csoport generátorát (azaz: az csoport elemeinek azt a részhalmazát, amelynek elemeiből szorzással előállítható a csoport többi eleme)!

172. ✆ Adott a H és G csoport a szorzási táblájával. Készítsünk programot, ami eldönti, hogy H részcsoportja-e a G csoportnak (azaz: az H elemei egyben G elemei, s a szorzás eredménye H-ban ugyanaz, mint G-ben)!

173. ⍾** Adott a H és a G csoport a szorzási táblájával. Készítsünk programot, ami eldönti, hogy a csoportok izomorfak-e (azaz: található-e egy-egy értelmű művelettartó megfeleltetés a H és a G elemei között)!

174. ⍢ Az életjátékban a sejtek egy négyzetrácsba rendezve helyezkednek el, s a sejteknek két állapotuk (élő, holt) lehet (egy sejtállomány pl. egy mátrixszal ábrázolható). A sejteknek — kivéve a szélén álló sejteket — nyolc szomszédjuk van. Egy holt sejt helyén akkor keletkezik új sejt, ha pontosan három élő szomszédja van, ill. egy élő sejt csak akkor marad életben, ha két vagy három élő szomszédja van (különben kihal). Készítsünk programot, ami egy tetszőleges kiindulási állapotból kiindulva kiírja egy sejtállomány megadott számú állapotváltozását!

175. ⍢ Adott egy játéktábla, amelyen egymás mellett nyolc lyuk van, mindegyikbe legfeljebb egy golyó fér. Kezdetben a baloldali négy lyukban egy-egy fekete, a jobboldali háromban egy-egy fehér golyó van. A megengedett műveletek: egy golyót áthelyezhetünk a szomszédos üres lyukba, vagy pedig egy golyót átugorva a második lyukba (ha az üres). A fekete golyókat azonban csak jobbra, a fehéreket csak balra vihetjük. Készítsünk programot, ami a fenti szabályokat betartva átmozgatja a fekete golyókat a tábla jobboldalára, s fehér golyókat a tábla baloldalára!

176. ⍢ Adott egy játéktábla, amelyen egymás mellett kilenc lyuk van, s mindegyikben egy-egy fekete vagy fehér golyó van. Készítsünk programot, ami a szomszédos golyók felcserélgetésével átmozgatja a fekete golyókat a tábla jobboldalára, a fehér golyókat pedig a tábla baloldalára!

177. ⍢ Adott egy játéktábla, amelyen egymás mellett kilenc lyuk van, s mindegyikben egy-egy kék, fehér vagy piros golyó van. Készítsünk programot, ami rendre tetszőleges két golyót felcserélve átmozgatja a kék golyókat a tábla baloldalára, a piros golyókat a tábla jobboldalára, míg a fehér golyók középen maradnak!

178. ⍢ Adott két halom gyufaszál. Két játékos felváltva lép, s az nyer, aki az utolsó gyufaszálat felveszi! Egy lépés abból áll, hogy a játékos az egyik halomból elvesz tetszőleges számú (de legalább egy) gyufaszálat. Készítsünk interaktív programot, ami eljátsza az egyik játékos szerepét!

179. ⍢ Adott két halom gyufaszál. Két játékos felváltva lép, s az nyer, aki az utolsó gyufaszálat felveszi! Egy lépés abból áll, hogy a játékos az egyik halomból elvesz tetszőleges számú (de legalább egy) gyufaszálat, vagy pedig mindkét halomból elvesz ugyanannyi (de legalább egy) gyufaszálat. Készítsünk interaktív programot, ami eljátsza az egyik játékos szerepét!

180. ⍾* Készítsünk olyan programot, ami megállapítja, hogy egy üres sakktáblán minimálisan hány lépésre van szüksége egy huszárnak, hogy egy adott A mezőről egy adott B mezőre jusson!

181. ✆ Készítsünk programot, ami meghatározza 12 huszárnak minden olyan elhelyezését a sakktáblán, amelyben minden mezőt támadnak!

182. ✆ Készítsünk egy rekurzív eljárást 8 vezér elhelyezésére egy sakktáblán úgy, hogy egyik se üsse a másikat!

183. ✆* Készítsünk olyan programot, ami megállapítja, hogy a sakktáblán egy adott állásban
   a) világos tehet-e gyaloglépést,
   b) világos képes-e ütni valamelyik gyalogjával,
   c) világos képes-e sakkot adni!

184. ✆* Készítsünk olyan programot, ami megállapítja, hogy a sakktáblán egy adott állásban világos hány lépést tehet!

185. ✆** Készítsünk olyan sakkprogramot, ami
   a) a világos királlyal és a vezérrel játszik a sötét király ellen;
   b) a világos királlyal és két bástyával játszik a sötét király ellen!

186. ✆** Készítsünk olyan programot, ami megállapítja, hogy a sakktáblán egy adott állásban
   a) világos adhat-e mattot egy lépésben,
   b) világos adhat-e mattot n lépésben!

187. ✆ Készítsünk interaktív programot, ami segít a történelmi dátumok tanulásában! A program tegyen fel kérdéseket (pl. "Mikor volt a mohácsi csata?"), s adja meg, hogy a válasz helyes volt, vagy sem. Rossz válasz esetén adja meg a helyes évszámot is, s néhány kérdés múlva tegye fel ismét ugyanezt a kérdést.

188. ✆ Készítsünk interaktív programot, ami segít valamely idegen nyelv szavainak tanulásában! A program kérdézze meg egy szó jelentését, s adja meg, hogy a válasz helyes volt, vagy sem. Rossz válasz esetén adja meg a helyes szót is, s néhány kérdés múlva tegye fel ismét ugyanezt a kérdést.

189. ✆ Válasszunk ki egy olyan helyzetet, amelyben N személy hasonló tevékenységet végez, s a tevékenység megkezdése és folytatása valami feltételtől függ. Ezután készítsünk tasko(ka)t a folyamatokat szimulálására, s helyezzünk el a programban megfelelő kiírásokat a folyamat nyomon követhetősége érdekében! Az egyes személyek paramétereit változtatva érdekes megfigyelni, hogyan változik az egész rendszer működése. Ilyen tevékenység lehet pl. a sörözés, ahol a folytatás feltétele az, hogy a korsóban legyen még sör. Ebben a feladatban nyilván szükség van egy kocsmárosra is, aki időnként körbejár, s tölt az üres korsókba.

190. ✆ Egy kereszteződésbe a különféle irányokból adott gyakorisággal érkeznek a gépkocsik. A kereszteződésben elhelyezett forgalmi lámpa adott szabályok

szerint működik. Készítsünk taskok felhasználásával egy programot az előálló forgalmi dugók vizsgálatára!

191. ④ Készítsünk programot egy nemdeterminisztikus automata működésének szimulálására taskok segítségével olymódon, hogy nem egyértelmű esetben a program indítson el egy újabb taskot! Az automata legyen $A = (\{q0, q1, q2, q3, q4, q5\}, \{a, b, c\}, \delta, q0, \{q4\})$, ahol $\delta(q0, a) = \{q0, q1\}, \delta(q0, b) = \{q0, q2\}, \delta(q0, c) = \{q4\}, \delta(q1, c) = \{q0, q3\}, \delta(q2, b) = \{q2\}, \delta(q2, c) = \{q0, q3\}, \delta(q3, c) = \{q4\}$, az összes többi esetre $\delta(x, y) = \{q5\}$.

192. ③ Készítsünk általános eljárást egy determinisztikus automata működésének szimulálására! Az állapotok halmaza (típusa), a bemenő jelek halmaza (típusa), az átmeneti függvény, a kezdő- és végállapotok mind legyenek az eljárás paraméterei!

193. ③ Adva van két edény: az egyikben az $1, 2, ..., n$ számokkal megjelölt golyók vannak, a másik üres. Állítsunk elő 1 és n közötti egyenletes eloszlású véletlenszámokat, s az adott számmal jelölt golyót mindig tegyük át a másik edénybe. Készítsünk programot, ami szimulálja ezt a folyamatot, s vizsgáljuk az edények tartalmának változását!

194. ② Egy részeg (R pont) ember áll egy lámpaoszlopnál (origó). Arra vagyunk kiváncsiak, hogy N lépés után milyen messzire jut az oszloptól. Minden lépése egységnyi hosszúságú, s az iránya véletlenszerű. Készítsünk programot, ami K db kísérletet végez, s ezek alapján megadja a várható távolságot!

195. ④* Egy zárt teremben 30 dohányos ember van. Minden időegység végén 20% annak a valószínűsége, hogy eggyel nő, 20%, hogy eggyel csökken az égő cigaretták száma, 60%, hogy nincs változás. Egy cigaretta egy időegység alatt egy egységnyi füstmennyiséget termel. Ha a teremben a füstmennyiség eléri a 200 egységet, akkor a számítógép elindítja az elszívóberendezést, amely egy időegység alatt 30 egységgel csökkenti a teremben a füstmennyiséget. Ha a teremben a füstmennyiség 40 egységnél kevesebb, akkor a gép kikapcsolja az elszívót. Készítsünk programot, ami időegységenként kiírja a füstmennyiség értékét, a dohányzók számát és az elszívó kapcsolójának állását!

196. ② A bűvös négyzet egy olyan négyzetes mátrix, amelynek sorösszegei, oszlopösszegei és a fő- ill. a mellékátlójában levő elemek összege megegyezik, s ez az összeg $\frac{n(n^2+1)}{2}$ (ahol n -nel jelöltük a négyzet oldalát, és n páratlan). Készítsünk programot, ami az n érték ismeretében elkészít egy bűvös négyzetet!

197. ②** A latin négyzet egy olyan négyzetes mátrix, amelynek sorai és oszlopai az $1, ..., n$ számok különböző permutációi (n -nel jelöltük a négyzet oldalát). Készítsünk programot, ami az n érték ismeretében elkészít egy latin négyzetet!

198. ① Készítsünk egy programot, ami egy számsorozatról megállapítja, hogy

periodikus-e (egy sorozat akkor periodikus, ha előállítható egy elemsorozat többszöri egymás mellé írásával)!

199. ☺ Készítsünk programot, ami az 1,2,3,...,9 számok közé + és – jeleket helyez el úgy, hogy a kifejezés értéke egy előre megadott szám legyen egyenlő! (Pl. $122 = 12 + 34 - 5 - 6 + 78 + 9$.)

200. ☺ Készítsünk olyan programot, ami előállítja
a) az első n természetes szám összes permutációját!
b) az első n természetes szám összes olyan permutációját, ahol $p_i \neq i$ $(1 \leq i \leq n)$!

201. ☹ Adott egy N teremből álló labirintus egy bináris mátrix formájában: $a_{ij} = 1$ azt jelenti, hogy az i. és a j. termeket folyosó köti össze. Készítsünk programot, ami megadja, hogy milyen termeken keresztül juthatunk el egy adott teremből egy adott másik terembe!

202. ☹** Egy vállalat N állást hirdet meg. Jelentkezik rá ugyanannyi pályázó, s mindegyik közli, hogy melyik állásokat tudná betölteni. Készítsünk programot, ami eldönti, hogy a pályázókkal betölthető-e minden állás, s ha igen, akkor adja meg, melyiket milyen állásra kell felvenni!

203. ☹* Egy teherautó A -ból B -be megy. Az út mentén vannak az A_i, B_i helyek tetszőleges sorrendben (de a megfelelő B_i mindig az A_i párja után van). Minden A_i helyről a hozzá tartozó B_i helyre kell szállítani egységnyi mennyiséget úgy, hogy a teherautó egyszerre csak egy terhet szállíthat. Készítsünk programot, ami eldönti, hogy melyik szállításokat kell elvállalni ahhoz, hogy a legtöbb igényt kielégítsük!

204. ☹ Adott N darab tárgynak a súlya. Osszuk a tárgyakat két csoportra úgy, hogy a két csoport súlya a lehető legközelebb legyen egymáshoz!

T a r t a l o m

	Előszó	3
1.	Az Ada nyelv áttekintése	4
1.1.	Az Ada történeti háttere	4
1.2.	Az Ada rövid jellemzése	6
2.	Lexikális elemek	8
3.	Deklarációk	10
4.	Típusok	13
4.1.	Az altípus fogalma	14
4.2.	Származtatott típusok	15
4.3.	Diszkrét típusok	15
4.3.1.	Felsorolási típusok	16
4.3.2.	Az egész típusok	17
4.3.3.	A diszkrét típusok típusműveletei	17
4.4.	Valós típusok	18
4.4.1.	Lebegőpontos típusok	19
4.4.2.	Fixpontos típusok	20
4.5.	Tömb típusok	21
4.6.	A rekord típus	24
4.6.1.	A rekord diszkriminánsai	25
4.6.2.	Variáns rekord	26
4.7.	Pointer típusok	28
5.	Utasítások	31
5.1.	Az értékadás	31
5.2.	Az **if** utasítás	32
5.3.	A **case** utasítás	33
5.4.	Ciklus utasítások	33
5.5.	A blokk utasítás	34
5.6.	Az **exit** utasítás	36
5.7.	A **return** utasítás	36
5.8.	A **goto** utasítás	36
6.	Alprogramok	38
6.1.	Alprogram definíciója	38
6.2.	Alprogram hívása	39
6.3.	Paraméterátadás	41
6.4.	A formális paraméterk default értéke	42
6.5.	Alprogramok átlapolása	43
6.6.	Az operátorok átlapolása	44
6.7.	Példaprogramok	45
7.	A package	48

7.1.	A package specifikációja	48
7.2.	A package törzse	48
7.3.	A private típus	49
7.4.	A **limited private** típus	52
7.5.	Példaprogramok	52
8.	Hibakezelés, kivételkezelés	57
8.1.	Példaprogramok	60
9.	Láthatósági szabályok	61
9.1.	A **use** utasítás	63
9.2.	Az átnevezés	63
10.	Programstruktúra	65
10.1	Alegységek	68
10.2.	Példaprogramok	69
11.	Generic	76
11.1.	Generic deklarációja	76
11.2.	Példányosítás	79
11.3.	Példaprogramok	80
12.	Taskok	83
12.1.	A task specifikációja és törzse	83
12.2.	Task típusok, task objektumok	84
12.3.	Taskok aktivizálása, végrehajtása	85
12.4.	Taskok terminálása	85
12.5.	**entry**, **entry** hívások, **accept** utasítás	86
12.6.	A **delay** és az **abort** utasítás	88
12.7.	A **select** utasítás	89
12.7.1.	A szelektív várakoztatás	89
12.7.2.	A feltételes **entry** hívás	90
12.7.3.	Időhöz kötött **entry** hívás	91
12.8.	Példaprogramok	91
13.	Reprezentációs specifikációk	101
14.	Fordítási direktívák	103
15.	Input - output	104
15.1.	Szekvenciális file-ok kezelése	105
15.2.	Direkt file-ok kezelése	106
15.3.	Textfile-ok	107
15.3.1.	Karakteres és string I/O	110
15.3.2.	Integer I/O	111
15.3.3.	Float és fixed I/O	112
15.3.4.	A felsorolási típusokra vonatkozó I/O	114
15.4.	Példaprogramok	115
	Függelék	116
	A) A **STANDARD** package	116
	B) Az Ada fogalmainak összefoglalása	121
	C) Az Ada szintaktikus szabályai névsorba szedve	131
	Feladatok	138

æ