2. előadás

Programegység, fordítási egység, könyvtári egység, beágyazás, blokkszerkezet, alprogramok, csomagok

Szintaxis és szemantika

- A nyelv *szintaxisa:*
  azoknak a szabályoknak az összessége, amelyek az adott nyelven írható összes lehetséges, *formailag helyes* programot (jelsorozatot) definiálják.
- A nyelv *szemantikája:*
  az adott nyelv programjainak *jelentését* leíró szabályok összessége.

Példa

- DD / DD / DDDD 01 / 02 / 2004

  2004. január 2. vagy 2004. február 1.?
- Ada / C++ I = 5

  3.1 + 4

- A szintaxis befolyásolja a programok
  - olvashatóságát
  - megbízhatóságát

A szabályok ellenőrzése

- A fordítóprogram végzi
- Fordítási hibák
  - lexikai, szintaktikus, statikus szemantikai
  - például típushelyesség
- Futási hibák
  - dinamikus szemantikai
- A professzionális programfejlesztéshez:
  - minél előbb derüljön ki (inkább fordításkor)
  - szigorú nyelv (pl. erősen típusos)

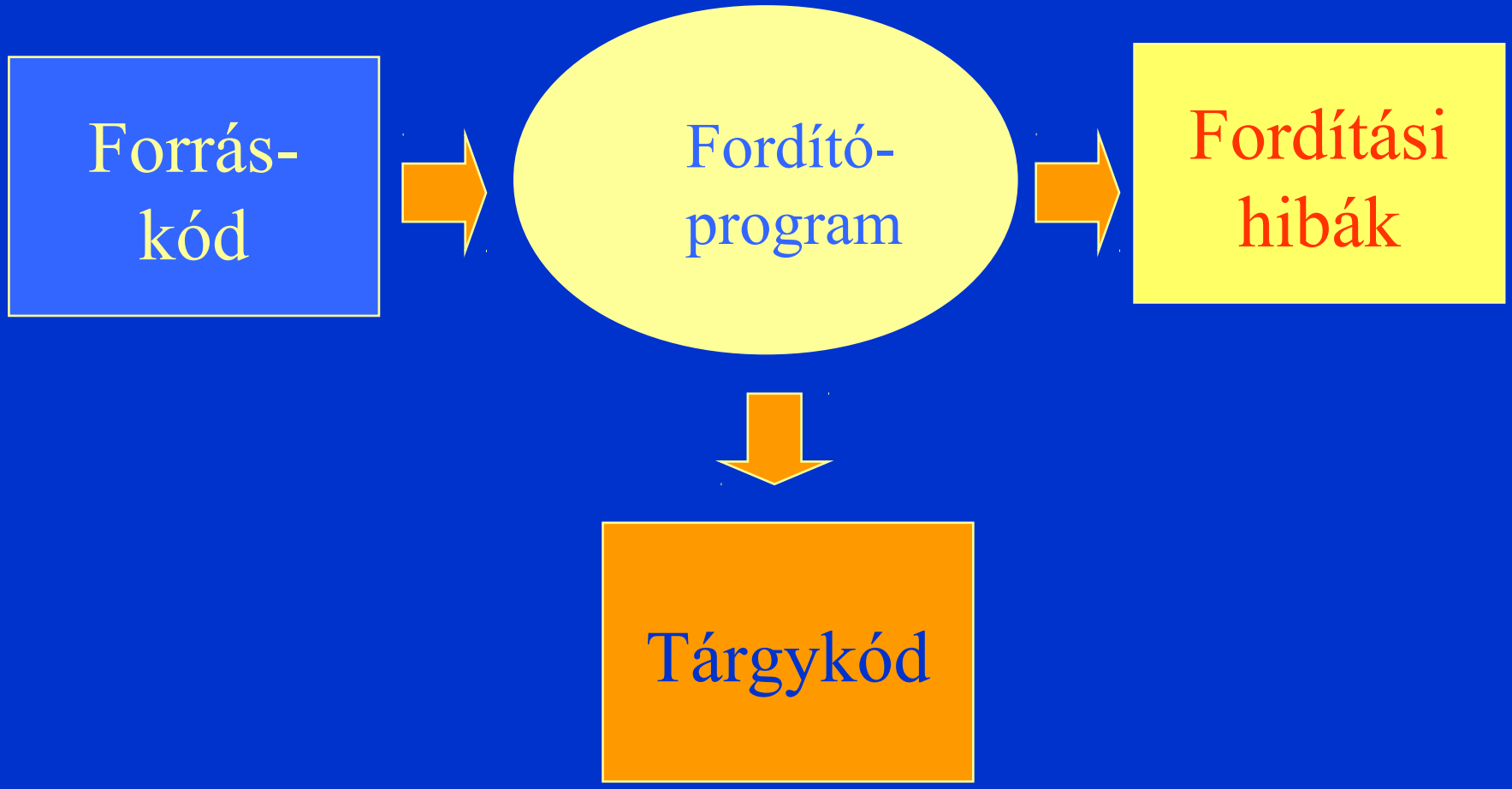
Interpreter és fordítóprogram

- Az *interpreter* egy utasítás lefordítása után azonnal végrehajtja azt
- A *fordítóprogram* átalakítja a programot egy vele ekvivalens formára
  - a számítógép által közvetlenül végrehajtható forma
  - egy másik programozási nyelv

Előnyök, hátrányok

- Az interpreter esetében közvetlenebb a kapcsolat a programozó és a program között
  - gyorsabb, egyszerűbb visszajelzés
  - általában kísérletezésre (pl. fejlesztésre) használható
  - gyakran pontosabb üzenetek a futási hibákról
- A lefordított programok „hatékonyabban futnak”
  - egyszer kell fordítani, sokszor lehet futtatni
  - az ellenőrzéseket csak egyszer kell megcsinálni
  - lehet optimalizálni

Fordítás:



Virtuális gép

- Java, .NET
- A forráskódot lefordítjuk egy univerzális tárgykódra (Java: bájtkód)
- Ez a tárgykód egy virtuális számítógép által végrehajtható forma
- A virtuális számítógép lényegében egy interpreter, de nem „forráskódot” értelmez, hanem ezt a „tárgykódot”

Fordításkor

- A lefordítandó program a *forrásprogram*.
- A fordítás eredményeként kapott program a *tárgyprogram*.
- Az az idő, amikor az utasítások fordítása folyik, a *fordítási idő*.
- Az az idő, amikor az utasítások végrehajtása folyik, a *végrehajtási idő*.

Fordítási egység

- *Fordítási egység vagy modul:* a nyelvnek az az egysége, ami a fordítóprogram egyszeri lefuttatásával, a program többi részétől elkülönülten lefordítható.
- Több modulból álló programok esetében a fordítás és a végrehajtás között: *a program összeszerkesztése*.
- FORTRAN

Szerkesztés:

A szerkesztési idő

- Általában a fordítással „egyidőben”
  - közvetlenül a fordítás után
  - futtatható program
  - static time
- Futási/végrehajtási időben (dynamic time)
  - nem feltétlenül készül futtatható program (Java)
  - nem feltétlenül kerül be minden a futtatható programba (dll, so)

Futási idő

- A program betöltődik a memóriába
- Relatív címek feloldása
- Tényleges programvégrehajtás

- Dinamikus szerkesztés
- Java: bájtkód ellenőrzése
- Java: JIT fordítás

A statikus és a dinamikus szerkesztés előnyei/hátrányai

- Statikus
  - csak egyszer kell megcsinálni, nem minden futtatáskor
- Dinamikus
  - megoszthatók nagy könyvtárak a programok között, kisebb méretű futtatott programok
  - csak az szerkesztődik be, amire szükség van
  - ez esetleg mégis csökkenti a végrehajtási időt
  - verziókhoz könnyebben alkalmazkodó program
  - gyakori futási hiba (könyvtár hiánya vagy rossz verziója miatt)

Előfordító

- Forrásszöveg átalakítása
- Forrásszövegből forrásszöveget csinál
- Például a C/C++ előfordító:

  #include

  #define

  #if, #else, #endif - generatív programozás

  #ifdef, #ifndef, #undef (feltételes fordítás)

Programegység és fordítási egység

- A *programegység* egy részfeladatot megoldó, tehát funkcionálisan összefüggő programrész.
- A *fordítási egység* egy önállóan kezelhető, tehát viszonylag zárt egység.
  - nem csak logikailag, hanem technikailag is darabolható a program
  - könyvtárak: újrafelhasználhatóság

Fordítási egység a C/C++ nyelvekben

- Amit az előfordító előállít egy forrásfájlból
  - más forrásfájlok beillesztése
  - makrók feldolgozása
- Tipikusan: egy .c, illetve .cpp fájlba bekerülnek a hivatkozott (#include) fejállományok
- Globális változók, függvények, osztálydefiníciók, sablondefiníciók sorozata

Fordítási egység az Adában (1)

- Például a főprogram
- Egy (paraméter nélküli) eljárás + a használt könyvtárak megnevezése (with utasítás)

```
with Ada.Text_IO;
procedure Hello is
begin
   Ada.Text_IO.Put_Line("Hello");
end Hello;
```

Fordítási egység az Adában (2)

- Általában is lehet egy alprogram programegység definíciója a kapcsolatok megadásával

```
with Text_IO;
procedure Kiír( S: String ) is
begin
    Text_IO.Put_Line(S);
end Kiír;
```

```
function Négyzet( N: Integer )
 return Natural is
 begin
        return N ** 2;
 end Négyzet;
```

Program több fordítási egységből

```
function Négyzet( N: Integer ) return Integer is
begin
    return N ** 2;
end Négyzet;
```

```
with Ada.Integer_Text_IO, Négyzet;
procedure Kilenc is
begin
    Ada.Integer_Text_IO.Put( Négyzet(3) );
end Kilenc;
```

Program több fordítási egységből

```
function Négyzet( N: Integer ) return Integer is
begin
    return N ** 2;
end Négyzet;
```

```
with Ada.Integer_Text_IO; with Négyzet;
procedure Kilenc is
begin
    Ada.Integer_Text_IO.Put( Négyzet(3) );
end Kilenc;
```

Fordítási egységek kapcsolatai

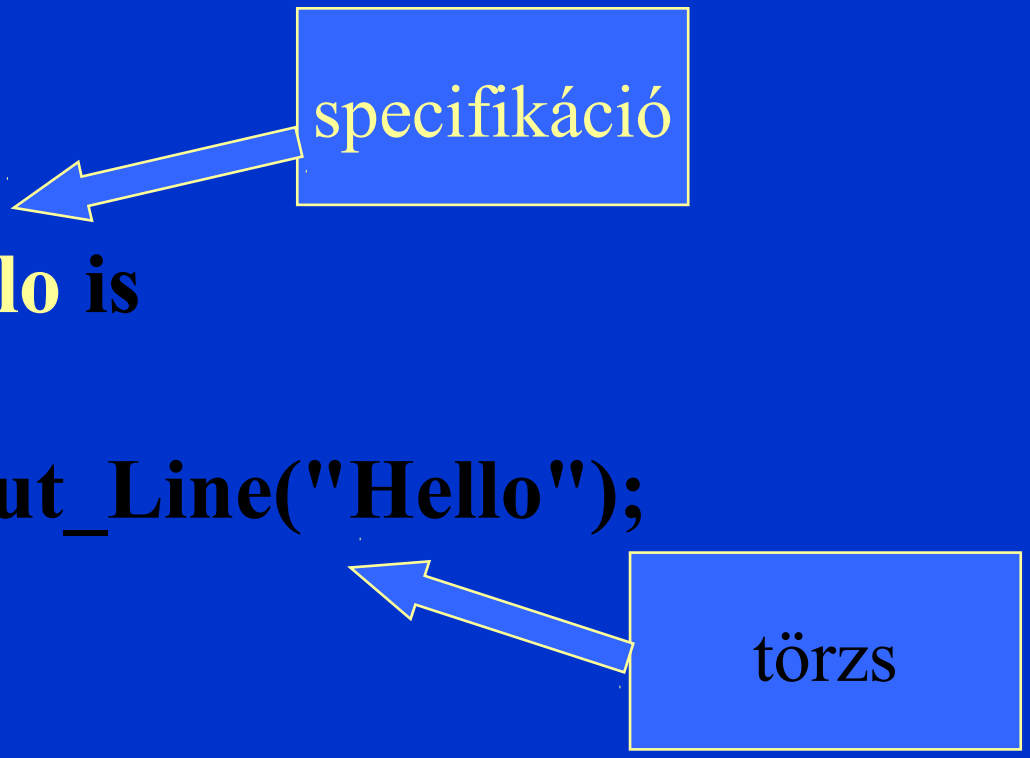
- Ada: with utasítás
  - a fordító ellenőrzi, mit, hogyan használunk
  - például a használt alprogram paraméterezése
- C++: ?
  - közösen használt fejállományok (.h)
  - csúf szerkesztési hibák
  - könnyebb hibát véteni
- Más nyelvek: export/import lista

A programegységek részei

- **a specifikáció** tartalmazza azt az információt, ami más egységek felé látható kell legyen
  - ***hogyan kell használni***
- **a törzs** tartalmazza az implementációs részleteket: ez **rejtett** más programegységek felé
  - ***hogyan működik***

A programegységek részei: példa

```
with Text_IO;

procedure Hello is
begin
    Text_IO.Put_Line("Hello");
end Hello;
```



Csomag programegység

- Logikailag kapcsolatban álló entitások (alprogramok, típusok, konstansok és változók) gyűjteményeit definiálják
- Bonyolultabb szerkezetű, mint az alprogram

- Alprogram: végrehajtjuk
  Csomag: a komponenseit használjuk

Csomag specifikációja és törzse

```
package A is
        …
end A;
```

```
package body A is
        …
end A;
```

Csomag specifikációja

- Nem tartalmaz implementációkat (törzseket)
- Tartalmazhat például: alprogramdeklarációt

```
package Ada.Text_IO is
    …
    procedure Put_Line( Item: in String ) ;
    …
end Ada.Text_IO;
```

Csomag törzse

- Implementációkat (törzseket) is tartalmazhat

```
package body Ada.Text_IO is
    …
    procedure Put_Line( Item: in String ) is
        …
    end Put_Line;
    …
end Ada.Text_IO;
```

Csomagok különálló fordítása

- A specifikációja és a törzse is külön-külön fordítható
- Egy külön fordított csomag két fordítási egységet tesz majd ki (!)

- Ilyenek lesznek a sablonok is
- Az alprogramoknál egy fordítási egység van

Csomag: specifikáció és törzs külön fordítható

- A csomagok használó programegységek fejlesztéséhez elég a csomag specifikációja
  - párhuzamos fejlesztés, team-munka
  - interfészek
- A kliens is fordítható, ha a specifikáció már le van fordítva
- Szerződés

A GNAT specialitásai

- Minden fordítási egységet külön fájlba kell írni
  - a fájl neve megegyezik a programegység nevével
- Csomag specifikációja: .ads
- Csomag törzse (body): .adb
- Alprogram: .adb

Könyvtári egység

- Külön fordított programegység
- Az Adában: 1 vagy 2 fordítási egység
  - pl. alprogram versus csomag
- Újrafelhasználhatóság
- Szabványos könyvtárak, saját könyvtárak

Csomag (helyett) más nyelvekben

- C++: osztály, névtér
- Java: osztály/interfész, csomag
- Modula-2, Clean: module

A use utasítás

- Csomag használata: a komponenseit használjuk
- Minősített névvel

  Text_IO.Put_Line("Hello");
- A minősítés elhagyható, ha:

  use Text_IO;
- Csak csomagra alkalmazható
  - Pascal with utasítása: rekordok komponenseit

use utasítás nélkül

```
with Text_IO;
procedure Hello is
begin
    Text_IO.Put_Line("Hello");
end Hello;
```

use utasítással

```
with Text_IO;
use Text_IO;
procedure Hello is
begin
    Put_Line("Hello");
end Hello;
```

Más nyelvekben use-szerű

C++: using namespace

Java: import

Programegység beágyazása

- Ha nem akarom külön fordítani a használt programegységet
- Vagy könyvtári egységet csinálok a programegységből, vagy beágyazom
- Ada: bármilyen programegységet bármilyen programegységbe
- ALGOL 60 (blokkszerkezetes nyelv)

Programegység beágyazása: példa

```
with Ada.Integer_Text_IO;
procedure Kilenc is
      function Négyzet( N: Integer ) return Integer is
      begin
            return N ** 2;
      end Négyzet;
begin
      Ada.Integer_Text_IO.Put( Négyzet(3) );
end Kilenc;
```

Blokk utasítások egymásba ágyazása

```
declare
        …
begin
        …
        declare … begin … end;
        …
end;
```

Beágyazás a C++ nyelvben

- Blokk utasítások: { … { … } … }
- Programegységeknél: osztályba függvény
  class A { int f(int i){ return i; } };
- Osztályba osztály
- Függvénybe függvény nem ágyazható
  - Nem nevezzük blokkszerkezetesnek
- C, Java stb.

Mire jó a beágyazás?

- Ha egy programegység hatókörét szűkre akarjuk szabni
  - lokális, nem globális
- Speciális, nem akarjuk újrafelhasználhatóvá tenni
  - logikai indok
  - olvashatóság
  - egységbe zárás
  - bonyolultság kezelése
- Hatékonysággal kapcsolatos indokok

Alprogram

- Programegység
- Végrehajtás kezdeményezése: meghívással
  - vannak más „hívható egységek” is...
- Eljárás és függvény

- C jellegű nyelvek: csak „függvény”
  - esetleg void visszatérési értékkel

Eljárások és függvények

- Az *eljárások* általában a változók által kifeszített téren (állapottéren) vagy a program környezetén elvégzendő transzformációkat adnak meg.
- A *függvények* valamilyen értéket állítanak elő, de transzformációt nem végeznek. Sem a program változóinak értékére, sem a program környezetére nincsenek semmilyen hatással: a függvényeknek nincs *mellékhatásuk*. (Elméletben!)

Ada függvény

```
function Faktoriális ( N: Natural ) return Positive is
    Fakt: Positive := 1;
begin
    for I in 1..N loop
        Fakt := Fakt * I;
    end loop;
    return Fakt;
end Faktoriális;
```

Ada eljárás

```
procedure Cserél ( A, B: in out Integer ) is
    Temp: Integer := A;
begin
    A := B;
    B := Temp;
end Cserél;
```

Névválasztás

- Eljárás végrehajtása: utasítás.
  Eljárás neve: ige.

  Egyenest_Rajzol(Kezdôpont,Végpont);

- Függvény végrehajtása:
  kifejezés értékének meghatározása.
  Függvény neve: főnév vagy melléknév.

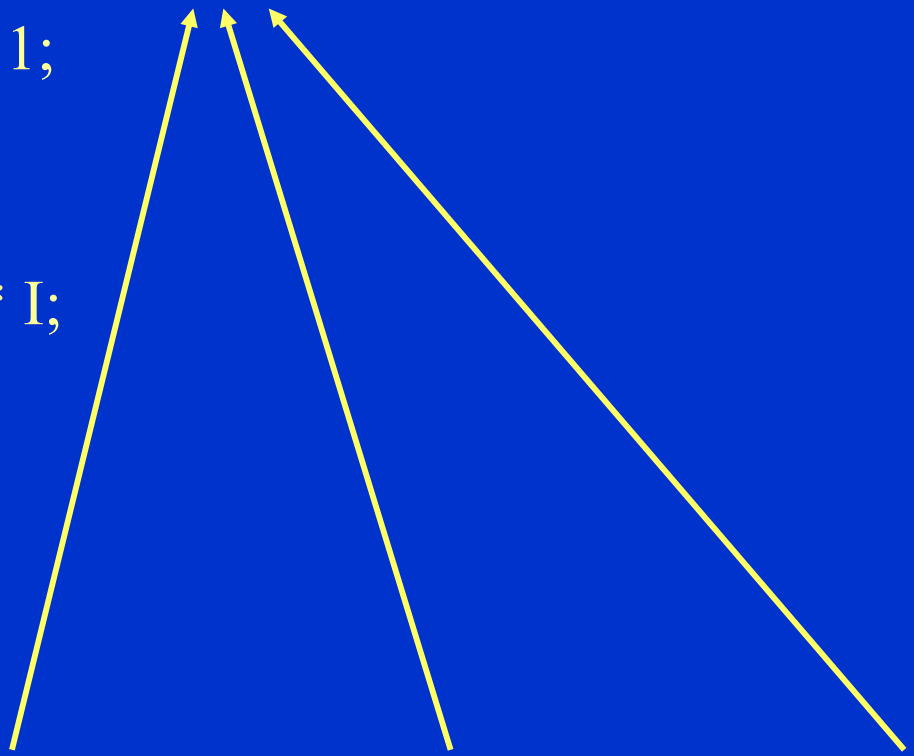
  if Elemek_Száma(Halmaz) > 0 then ...

Paraméterek, visszatérési érték

- Információ átadása / átvétele alprogramhívásnál:
  - paramétereken keresztül
  - visszatérési értéken keresztül
  - nem-lokális változón keresztül
- Paramétereknél: az információ áramlása
  - Merre: a paraméterek módja (Ada)
  - Hogyan: a paraméterátadás technikája (paraméterátadás módja)
- Alprogram hívásakor a *formális* paramétereknek *aktuális* paramétereket feleltetünk meg.

Aktuális és formális paraméter

```
function Faktoriális ( N: Natural ) return Positive is
    Fakt: Positive := 1;
begin
    for I in 1..N loop
        Fakt := Fakt * I;
    end loop;
    return Fakt;
end Faktoriális;

N_Alatt_K :=
        Faktoriális(N) / ( Faktoriális(K) * Faktoriális(L-K)
    )
```



Visszatérés: return

- A függvények visszatérési értékét és annak típusát a **return** kulcsszó után kell megadni.
- A típusra nincs megkötés
- A függvénynek tartalmaznia **kell** (egy vagy több) **return** utasítást is.

  (**Program_Error**, ha nem azzal ér véget!)

- Paraméter nélküli **return** utasítás eljárásokban állhat: hatására az eljárás véget ér.

Formális paraméterlista

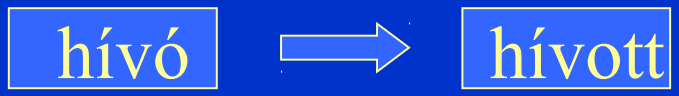
```
procedure Szia_Világ

function Maximum ( X: Integer; Y: Integer )
                                    return Integer

function Maximum ( X, Y: Integer )
                                    return Integer

procedure Cserél ( A, B: in out Integer )
```

Paraméterek típusa

- Az aktuális és a formális paraméter típusának meg kell egyeznie
  - fordítás közben ellenőrzött
- Paraméterátadáskor ellenőrzésre kerül az is, hogy az átadott értékek megfelelnek-e az altípus-megszorításoknak.
  - futási időben ellenőrzött

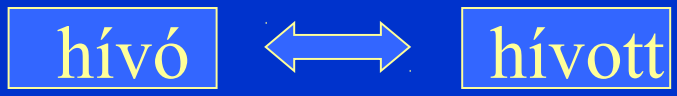
A paraméterek módja

- Az **in** módú: az *alprogramba* juttat információt a hívóból



- Az **out** módú: az alprogramban kiszámolt értéket a hívónak át tudjuk adni

- Az **in out** módú:
  - híváskor információt adnak az alprogramnak a hívótól
  - az alprogram befejeződésekor információt adnak a hívónak az alprogramtól.



Példák

```
procedure Put ( Item: in Integer )

procedure Közös ( A, B: in Positive;
                  Lnko, Lkkt: out Positive)

procedure Cserél ( A, B: in out Integer )
```

Mit lehet tenni velük

- Egy **in** típusú formális paraméternek nem adhatunk értéket, csak olvashatjuk.
- Egy **out** módú paramétert lehet írni, és ha már kapott valamilyen értéket az alprogramon belül, lehet olvasni is.
- Egy **in out** módú paraméter értékét olvashatjuk és írhatjuk is.

- C++: const, Modula-3: READONLY

Például:

```
procedure E (  Vi: in Integer;
               Vo: out Integer;
               Vio: in out Integer )
is
begin
   Vio := Vi + Vo;      -- helytelen, Vo-t nem olvashatjuk
   Vi := Vio;           -- helytelen, Vi-t nem írhatjuk
   Vo := Vi;            -- helyes
   Vo := 2*Vo+Vio;      -- helyes, Vo már kapott értéket
end E;
```

Helyes

```
procedure Közös (      A, B: in Positive;
                       Lnko, Lkkt: out Positive )
is
       X: Positive := A;
       Y: Positive := B;
begin
       while X /= Y loop
              if X > Y then X := X - Y; else Y := Y - X; end if;
       end loop;
       Lnko := X;
       Lkkt := A * B / Lnko;
end Közös;
```

Helytelen

```
procedure Közös (      A, B: in Positive;
                       Lnko, Lkkt: out Positive )

is
begin
        Lkkt := A * B;
        while A /= B loop
                if A > B then A := A - B; else B := B - A; end if;
        end loop;
        Lnko := A;
        Lkkt := Lkkt / Lnko;
end Közös;
```

Helytelen tervezés

```
procedure Közös (      A, B: in out Positive;
                       Lnko, Lkkt: out Positive )

is
begin
        Lkkt := A * B;
        while A /= B loop
                if A > B then A := A - B; else B := B - A; end if;
        end loop;
        Lnko := A;
        Lkkt := Lkkt / Lnko;
end Közös;
```

Mi lehet aktuális paraméter:

- Egy **in** paraméternek átadhatunk egy tetszőleges kifejezést (például egy változót is): a kifejezés értéke lesz az aktuális paraméter.
- Egy **out** vagy **in out** paraméterként viszont csak egy „balértéket” (pl. változót) adhatunk át: ebbe a balértékbe kerül majd az alprogram által kiszámított érték (illetve **in out** mód esetén ez a balérték tartalmazza a bemenő paraméterértéket is).

Hívások

```
Közös(A,A+42,X,Y);                  -- helyes
Közös(Faktoriális(5),42,X,Y);       -- helyes
Közös(30,12,20,10);                 -- helytelen

Cserél(X,Y);                        -- helyes
Cserél(2,6);                        -- helytelen
```

Paraméter alapértelmezett módja: **in**

```
procedure Egyenest_Rajzol
                ( Kezdôpont, Végpont: Pont )
```

- Mindkét paraméter in módú
- Eljárásoknál inkább írjuk ki...

Függvényparaméterek módja

- Függvények formális paramétere csak **in** módú lehet!
- Csak számoljon ki egy értéket, és adja vissza…
- Nem is szoktuk kiírni az in szócskát...

Rekurzió

- Közvetlenül vagy közvetve önmagát hívó alprogram
- A ciklussal „egyenértékű”
- Elágazás van benne!

```
function Faktoriális ( N: Natural ) return Positive is
begin
        if N > 1 then return N * Faktoriális(N-1);
        else return 1;
        end if;
end Faktoriális;
```