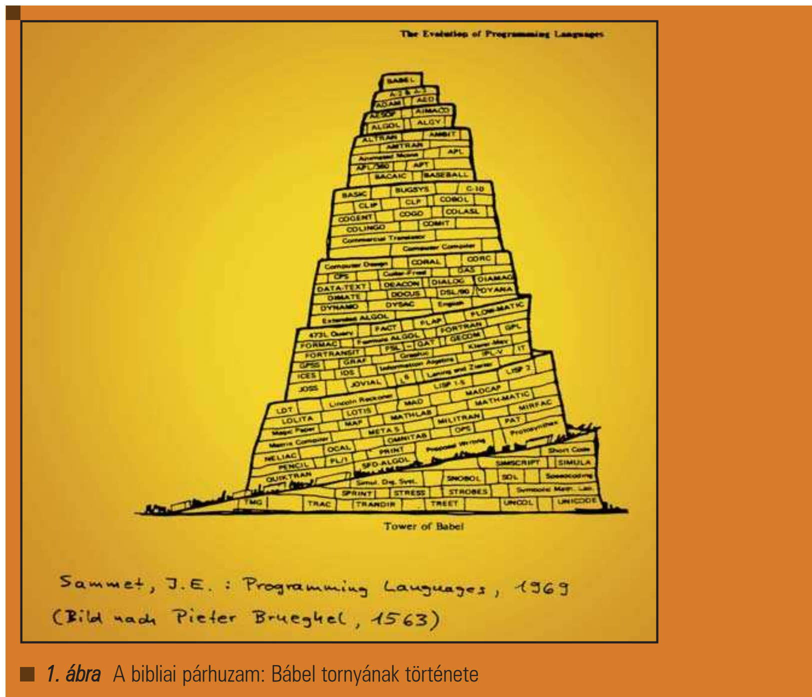
Biztonság, megbízhatóság, ADA

Biztosan felmerült már a kérdés sokakban, hogy azokon a helyeken, ahol életek függnek számítógépektől, ahol a hibák lehetőségét teljesen minimalizálni kell, milyen programozási eszközt, vagy eszközöket használnak. Ma már temérdek programnyelv létezik. Vannak ismertek és kevésbé ismertek. Véleményem szerint az utóbbiak közé tartozik az ADA nyelv, mely megpróbált békét teremteni a programozási nyelvek bábeli zűrzavarában, sokszínűségével, gazdagságával és elég szigorú szabályaival.

© Kiskapu Kft. Minden jog fenntartva

A nyelv története 1970-es évek elejére nyúlik vissza. Ekkor szervezte meg ugyanis a ***US DoD (United States Department of Defense)*** a ***HOLWG (High Order Language Working Group)*** nevű projektet ***William Whitaker*** vezetésével. Először nem is az volt a feladat, hogy teremtsenek egy új nyelvet. A projekt célja, az akkori programozási nyelvek vizsgálata volt. Sajnos a szigorú követelményeknek nem nagyon feleltek meg a korabeli nyelvek. A követelmények tömören a következők voltak: algoritmusok implementálásához jó és egyszerű jelölés rendszert biztosítson, eszközt adjon a programok bonyolultságának kezelésére, valamint lehetőséget adjon arra, hogy programozáskor el tudjunk vonatkoztatni a számítógéptől. 23 létező programozási nyelv jöhetett számításba: ***FORTRAN, COBOL, PL/I, HAL/S,***



1. ábra A bibliai párhuzam: Bábel tornyának története

© Kiskapu Kft. Minden jog fenntartva

2. ábra Ada Augusta Byron

TACPOL, CMS-2, CS-4, SPL/I, JOVIAL J3, JOVIAL J73, ALGOL 60, ALGOL 68, CORAL 66, Pascal, SUMULA 67, LIS, LTR, TRL/2, EUCLID, PDL2, PEARL, MORAL, EL/I

Sajnos a vizsgálat során a szakértők rájöttek: nincs olyan nyelv mely kívétel nélkül minden specifikált követelménynek megfelel. Arra az álláspontra jutottak, hogy ki kell alakítani egy új nyelvet, mely az eddigi nyelvek értékeiből merít és egymagában meg fog felelni a célnak. Tettek bizonyos megkötéseket. Például nem vehették számításba, az akkor már elavultnak számított nyelveket, mint a ***CORAL*** vagy a ***FORTRAN***. Voltak melyekből lehetett meríteni elemeket, például ***LIS***, ***RTL***. Alapnyelvnek választották a ***PASCAL***, ***ALGOL 68*** és ***PL/I*** nyelveket.

A kitűzött pályázatot végül is egy franciaországi labor nyerte el: ***CII Honeywell Bull*** és alapnyelvül a ***Pascalt*** választották.

A fejlesztés során a kiinduló projekt nevéből fakadóan a ***DoD-1*** nevet kapta, majd 1979 májusában keresztelték el, mint ***ADA***. A név ki mást takarhatna, mint a világ első programozóját ***Ada Augusta Byront***, ***Lord Byron*** lányát, aki ***Babbage*** asszisztense volt.

A fejlesztési szempontokat jobbára megbízhatóság és karbantarthatóság övezte. Ezért a nyelv erősen típusos lett és lehetőséget ad adatabsztrakciók kezelésére is. Karbantarthatóságot szolgálván több fordítási egységből áll. Ami pedig nagyon is lényeges, eszközöket szolgáltat a kivételesen hibás esetek kezelésére.

Az első 1983-as szabvány tehát a Pascalra épül, de ezen kívül hordozta többek között az ***EUCLID***, ***ALGOL 68***, ***SUE***, ***MODULA***, ***CLU*** nyelvek vonásait.

1995-re megnőttek az igények, megnőtt a hardverek kapacitása új technológiák terjedtek el, itt volt az ideje, hogy az ***ADA*** is felzárkózzon. Ebben az évben született meg az ***ADA95***, mely több új tulajdonságot hordozott: könnyebben illeszkedett más nyelveken íródott rendszerekhez, támogatta az objektum orientált programfejlesztést, a könyvtárszerkezetek létrehozását is hierarchikusabbá tették valamint párhuzamos programozást támogató elemekkel is bővült a repertoár.

Bár nehezen programozható, mégis valahol ebben rejlik az erőssége, nem hagyja, hogy a programozó butaságot kövessen el. Amire csak figyelmeztethet a fordító, arra figyelmeztet is, de nem csak a fordító lett ügyesen megalkotva, a nyelv szabályai eleve a logikus gondolatmenetet segítik elő.

Nagyon sok helyen használják, ahol a biztonság elsődleges szempont. ***ADA*** program irányítja többek között a ***Boeing 777***-esek erőforrás kezelő rendszerét valamint fék berendezéseit. Kigondolná hogy a ***GPS*** rendszerekhez tartozó műholdakban is helyet kapott pár ***ADA***-ban fejlesztett rendszer. A felhasználási területeket lehetne még sorolni: rakéták vezérlőrendszerei, párizsi, kairói, metró rendszer,a francia ***TGV*** irányítórendszere.

Az ***ADA*** felhasználási területeiről Többet megtudhatunk ha ellátogatunk a ➲ ***http://www.adahome.com/Ammo/Success*** címre.

Természetesen az ***ADA*** rendelkezik ***Linux*** alatt futó fordítóval is, melynek neve ***GNAT***. A fordító beszerezhető a ➲ ***http://www.gnat.com/*** címről, de a legtöbb ***Linux*** disztribúció alapból tartalmazza.

Most, hogy már tudunk valamit a nyelvről, ideje kipróbálni, hogy hogyan is néz ki a gyakorlatban. Mivel nem férne bele ebbe a szerény cikkbe, minden amit érdemes tudni, csak pár alap dolog kerül most a porondra, mely talán kedvet fog csinálni egyeseknek, hogy közelebbről is megismerjék e remek nyelvet.

Tehát egy ***ADA*** program egy vagy több ***programegységből*** áll, ezek többnyire külön is fordíthatóak, valamit egymásba ágyazhatók. Programegység lehet: ***alprogram***, ***csomag***, ***sablon***, ***taszk*** és ***védett egység***. Ne húzzuk tovább az időt! Lássuk végre egy ***„Hello World!”*** programot ***ADA*** nyelven (*1. lista*).

Már ebből a pár sorból is feltűnhet, hogy mennyire hasonlít a szintaxis a ***PASCAL*** nyelvre. Az ***ADA*** források általánosan ***adb*** kiterjesztésűek és lehetőleg megegyezik a fájlnév a programegység nevével, ebben az esetben ***hello.adb***. Az első sor megadja, hogy mely könyvtári egységet szeretnénk felhasználni. A második sor ebben az esetben a ***programegység*** (a programunk most egy fordítási egységet tartalmaz) ***specifikációja***, ami megadja, hogy mire és hogyan lehet felhasználni az általunk írt programegységet. A ***törzs*** pedig a 3-tól 5. sorig terjedő rész. Ez tartalmazza a programunk implementációs részét. A törzs is több részre bomlik: ***deklarációs rész***, ***utasítássorozat***, ***kivételkezelés*** (ez persze opcionális). Lássunk egy másik példát (*2. lista*).

1. lista Hellló világ! (hello.adb)

```
With Text_IO;
Procedure Hello is
Begin

Text_IO.Put_Line("Hello");
End Hello;
```

2. lista Adjunk össze két számot

```
With Integer_Text_IO;

Procedure Osszead is
       A,B:Integer;
Begin

Ada.Integer_Text_IO.Get(A);

Ada.Integer_Text_IO.Get(B);

Ada.Integer_Text_IO.Pu(A+B);
End Osszead;
```

© Kiskapu Kft. Minden jog fenntartva

3. lista „for” ciklus az ADA-ban

```
With Integer_Text_IO;
Procedure Forciklus is
Begin
        For i in 1..10 loop

Ada.Integer_Text_IO.Put(i);
        End loop;
End Forciklus;
```

4. lista „while” ciklus az ADA-ban

```
...
I:Integer;
Begin
        While I<=10 loop

Ada.Integer_Text_IO.Put(I);
                        I := I + 1;
        End loop;
...
```

5. Lista „exit when” konstrukció az ADA-ban

```
...
        loop

Ada.Integer_Text_IO.Get(N);
                exit when N=1;
        end loop;
...
```

Esetünkben a deklarációs rész az *„A,B:Integer;”* sor. Ebben a részben szerepelhetnek, változók, konstansok, típusok és más programegységek is, hiszen említettük, hogy a programegységek egymásba ágyazhatók.
A változókhoz lehet kezdőértéket is rendelni: `A:Integer := 3;`. Néhány alapvető típus az *ADA*-ban: `Integer`, `Natural`, `Positive`, `Float`, `Boolean`, `Character`, `String`. Ezek beépített típusok.
A következőkben áttekintjük az alapvető programozási konstrukciókat. Például igen érdekesen alakult az *ADA*-ban a ciklusok helyzete.
Az egyszerű *„for”* ciklusok ügyesen úgy lettek kialakítva, hogy a ciklusváltozó a ciklusmagon belül nem változtatható meg. Gondoljunk csak bele hányszor estünk már abba a hibába, hogy egy cikluson belül megváltoztattuk a ciklusváltozó értékét és persze elszállt a programunk.
Az *ADA* ezt egyszerűen küszöböli ki (*3. lista*).
Amint észrevettük az *„i”* ciklusváltozót külön nem is deklaráltuk. A változó hatóköre csak cikluson belülre terjed ki. Értékére tehát csak hivatkozni lehet, átírni azt nem tudjuk.
Ha például visszafelé szeretnénk az [1,10] intervallumon *„száguldani”*, akkor az *„in”* szócska után csak be kell szúrnunk a *„reverse”* szót.
Lássunk példát *„while”* ciklusra is: *4. lista*.
Az eredményünk az előző példával megegyező. Egyszerűen írhatunk végtelen ciklusokat is, oly módon, hogy elhagyjuk a `while <feltétel>` részt, azaz csak `loop` és `end loop` közé építjük a ciklusmagot. Mivel az *ADA* támogatja a párhuzamos rendszerek fejlesztését, ezért került bele ilyen egyszerű módon a végtelenített ciklus.
Az ilyen ciklusoknak a *taszkok* programozásában van nagy szerepe. Végtelenített ciklus kapcsán lehet megemlíteni az `exit when <feltétel>` konstrukciót. Ennek segítségével akár `while` vagy `for` ciklusból is ki lehet lépni (*5. lista*)
Az elágazások konstruálása is kicsivel szigorúbb, mint más nyelvekben.
Minden *„if”* utasítás a következőképpen épül fel:

```
if <feltétel> then
utasítások
end if;
```

Az *„end if;”* mindig kötelező! Feltételek kiértékelésében optimalizálásra ad lehetőséget a következő megoldás: *„and then”*.
Ezt a konstrukciót kihasználva erőszakoljuk a rendszert feltétel lusta kiértékelésére. Például ha feltételünk a következő: *„A AND B”* akkor optimálisabb lehet *„A AND THEN B”* formában. Az előző esetben A és *B* feltétel értéke mindig kiértékelődik, a másodikban ha A hamis, akkor tudjuk, mivel ÉS műveletről lévén szó, hogy az egész feltétel hamis, tehát a programunk *B* értékét már figyelembe sem veszi.
Lássuk most hogyan is alkothatunk alprogramokat. Az alprogramokat *ADA*-ban két csoportra lehet bontani: *eljárások*, *függvények*. A *függvények* valamely értéket adnak vissza eredményül, az *eljárások* a program változóit változtathatják meg, tehát a program állapotterében *„turkálhatnak”*. Itt is újítást vezet be az *ADA*. Az alprogramok paraméterezhetőek ez aránylag természetesen hangzik, viszont a paraméterek specifikálásakor megadhatjuk, hogy azok ki illetve bemeneti paraméterek az alprogram törzse szempontjából.
Hogy világosan lássuk miről is van szó, tanulmányozzuk a *6. listában* látható példát.
Amint látjuk az `in` illetve `out` szócskák segítségével adhatjuk meg egy alprogram számára, hogy paramétere milyen módon érhető el a törzsben.
Függvények esetén mindig `in` módon, azaz csak olvasható módban kapjuk a paramétereket.
A `Fuggveny`-ben például hibásnak számítana a `B := 3;` utasítás. Nem így az `Eljaras`ban hiszen ott megadtuk, hogy a *„B”* írható és olvasható is egyben, az A viszont itt is csak olvasható.
A fenti példa jól szemlélteti az alprogramok két fajtájának szintaxisát is, valamint kiderül, hogy függvényekben a `return` kulcsszóval adhatunk vissza értéket.
Az *ADA* lehetőséget ad többször felhasználható fordítási egységek leprogramozására is. Ezek az egységek nagyszerű előnye, hogy típusokkal is paraméterezhetőek. Például írhatunk

6. lista

```
Function Fuggveny(A,B:
➥Integer) return Integer is
Begin
        Return A+B;
End Fuggveny;

Procedure Eljaras(A:in
➥Integer;B:in out Integer) is
Begin
        B := A + B;
End Eljaras;
```

olyan csomagokat, melyek alkalmasak bármilyen típusra létrehozni egy vermet, vagy bináris fát. Ezzel a lehetőséggel a *C++*-ban is találkozhatunk. Viszont ami igen ritka és még a nagyszerű *C++* sem büszkélkedhet vele, az az, hogy az egyes csomagok nem csak típusokkal, hanem ***alprogramokkal is paraméterezhetőek!***
Az eddigiekből nem nagyon derülhetett ki, hogy miért is olyan kemény dolog *ADA*-ban programozni. A nehézségek, akkor kezdődnek mikor el kezdünk alkalmazni típusokat és azokból alkotott konstrukciókat.
A ***szintaxis*** talán nem is nehéz, hiszen aki kicsit is avatottabb a programozásban, biztos, hogy találkozott a *Pascal* nyelvvel. Ami új lehet szintaxis terén, például hogy tömbök esetén nem *[,]* jelek segítségével hivatkozunk az adott indexű elemre, hanem sima zárójelekkel. Tehát a

```
V:array of Integer(1..10);
```

10 hosszúságú, egészeket tartalmazó vektorunk 4. eleme: `V(4)`. Alapvetően azonban tényleg a *Pascal* szintaxisra épít a nyelv. Tehát ha a szintaxis nem is okoz, akkor a *„filozófia”* (nem teljesen a szemantika) okozhat fejtörést.
Az *ADA* nyelv igen nagy utat tett meg a mai napig. Fejlesztése természetesen nem ért véget. Az utolsó (*ADA95*) kiadás óta nem jelentkezett sok újdonság, de aki ellátogat a ➲ *http://www.gnat.com* címre, ott elég sűrűn láthatja az *ADA 2005* megnevezést. Valami várható a közeljövőben.
A nyelv sokoldalú, színes és nagyon hasznos dolgokat ad a programozó kezébe. Ne gondoljuk, hogy olyan könnyen megtanulhatjuk autodidakta módon, oktatóanyagokból a nyelvet, mint más esetekben!
A nyelv biztonságra törekszik, ezért sok szempontból megköti a programozó kezét. Szigorú típusossága elegendő ahhoz, hogy látszólag lehetetlenné tegye a könnyű programozhatóságot. Nagyon jó típuselméleti ismeretek szükségesek a könnyebb elsajátításhoz.
Manapság igen kevés programozó mondhatja, hogy tapasztalt *ADA* programozó. Viszont kitartással, türelemmel és sok sok idővel igen értékes tudásra tehetünk szert, ha magunkévá tesszük az *ADA* filozófiáját. Láthatjuk, hogy a szigor a precizitást és biztonságot vonja maga után és ez azt eredményezi, hogy a nyelv szélsőséges helyzetekben, hosszútávon üzemelő rendszerek fejlesztésénél, a levegőben repülőkön, a földön csillagászati kutató állomásokon, a föld alatt metrórendszerek irányításában is megállja a helyét. Mivel ember alkotta, ezért rendelkezik hibával, de a többi nyelvhez képest, az *ADA* kialakulását hatalmas nemzetközi erőfeszítések, tervezések előzték meg. A programozás terén ezért is említhetjük a precizitás, megbízhatóság szavak társaságában az *ADA* nyelvet.

Radics Péter
(peter.radics@gmail.com)

Az ELTE-n tanulok programtervező matematikus szakon. Hobbim a kosárlabda, autóvezetés, webdesign, programozás. Főleg webes alkalmazások fejlesztése érdekel. 4 éve megrögzött Linux felhasználó vagyok.

© Kiskapu Kft. Minden jog fenntartva