Programozási nyelvek, környezetek kifejezőerejének vizsgálata

Törley Gábor

pezsgo@elte.hu
ELTE IK Média- és Oktatásinformatikai Tanszék

Absztrakt. Az írott és szóbeli kommunikációban nagy szerepe van a megértést segítő eszközöknek: a tárgy többféle módon való megközelítésével, például írásban a fontosabb részek kiemelésével; szóban hangsúlyváltoztatásokkal, nonverbális kommunikációval növelhetjük mondandónk kifejezőerejét, érthetőségét. A cikkben programozási nyelvek és fejlesztői környezetek vannak összehasonlítva egymással, abból a szempontból, hogy milyen módon és mértékben segítik, támogatják a programozási feladatok megoldását.
A programozásban és természetesen a programozás-oktatásban is felmerül az igény megértést segítő eszközök után. Mik azok a tulajdonságok, amelyek meghatározzák egy program érthetőségét, olvashatóságát? Melyek azok az elvek, amelyek betartásával eszközt adunk a programozók, a programozást tanulók kezébe ahhoz, hogy a kész algoritmusból minél hatékonyabban helyesen működő programot alkothassanak? Ezekre a kérdésekre keressük a választ.

1. Bevezetés

Vizsgálatunkkal a programozás-oktatáshoz szeretnénk újabb ismereteket hozzátenni. Nem a kezdő lépések oktatási kérdései érdekelnek bennünket, hiszen a programozás stílusával és a nyelv kapcsolatával foglalkozunk, amely ideje korán lenne a „kezdetek kezdetén”. Feltesszük, hogy a tanuló már birtokolja az algoritmizálás alapjainak (szekvencia, elágazás, ciklus, eljárás, függvény, változók használata) ismeretét, és jártasságot szerzett valamilyen programozási nyelvben (várhatólag Comenius Logo vagy Imagine).

Programozási nyelvek kifejezőereje alatt értsük a nyelvek azon tulajdonságát, amely (struktúrájával, szintaktikájával stb.) támogatja a programozási feladat megoldását. Egy nyelv annál kifejezőbb, minél kevesebb szellemi befektetést, időt kíván a programozótól a kódolás során. Azaz az algoritmus meglétét feltételezve gátolja vagy támogatja a program végleges „formába öntését”. A kérdésfelvetés hasonlatos az elemi aritmetika jól ismert kérdéséhez, hogy ti. a számírás miként hatott/hat a számolási készségre.[1] [1] Manapság a program készítése során nemcsak a nyelvvel, hanem az azt kiegészítő fejlesztői környezettel dolgozunk. A vizsgálat folyamán a nyelvet és a környezetet komplex egységként fogjuk kezelni, hiszen pl. egy jó hibakereső funkció, ami nem a nyelv szerves része, is hozzájárul ahhoz, hogy minél kevesebb fejtöréssel jussunk el a megoldásig.

Ez a cikk a módszeres programozás [2] négy lépésének – kódolás, tesztelés, hibakeresés- és javítás – gyakorlatát mutatja be és összehasonlítja ebből a szempontból a különböző nyelveket és fejlesztői környezeteket. Célszerű külön választani a nyelv és a környezet tulajdonságait a következők szerint [3]:

- Nyelvi szempontok
  - Értelmes alapszavak
  - Egyszerű és következetes programszerkezet
  - Egyszerű kódolás, könnyen tanulhatóság
- Fejlesztői környezet szempontjai
  - Kulcsszavak kiemelése
  - Kódolást segítő szolgáltatások (pl. intellisence)
  - Fordítás, hibaüzenetek
  - Nyomkövető rendszer

Vizsgálatunkat most csak az imperatív nyelvek osztályára korlátozzuk.

[1] arab/indiai számjegyes írás vs. római számjegyekkel történő…, pláne a kínai számábrázolás…

Végezzünk el egy gondolatkísérletet: képzeljük magunkat a tanuló helyébe! Előtte van a kész (nem feltétlenül formálisan leírt) specifikáció és az algoritmus. A következő lépés a kódolás.

A szokásos „Helló világ” példa helyett az alábbi feladatot megoldó programdarabot tekintsük az összehasonlítás példájaként. A feladat: legyen adott egy egészekből álló számsorozat, döntsük el, van-e köztük páros szám!

A programozás során első lépésként specifikálni kell a programot, majd elkészíteni az algoritmust.

A specifikáció az ELTE informatika tanári szakán alkalmazott konvenciókat követi. (Ennek formális vagy informális volta most nem fontos, a lényeg, hogy a megoldáshoz egyértelmű információkkal szolgáljon.)

Specifikáció:

```
Bemenet:      N∈N, Tömb∈Z*, Páros: Z→L      [L={igaz, hamis} – Logikai értékek halmaza]

Kimenet:      Vane∈L

Előfeltétel:  Hossz(X)=N

Utófeltétel:  Vane ≡ ∃i∈[1..N]:Páros(Tömb_i)
              ahol
              Páros(x)=igaz, ha x ≡ 0 (mod 2),
              Páros(x)=hamis minden egyéb esetben.
```

Alább csak a lényegi algoritmust közöljük és vizsgáljuk. A beolvasást és a kiíratást csak a nyelvvel, ill. a környezettel kapcsolatban fogjuk tárgyalni, hiszen ott markánsan eltérőek lesznek.

Algoritmus:

```
Eljárás Vane_páros(Konstans N: Egész; Tömb: tTömb; Változó Vane: Logikai):
    Változó
       I: Egész

    I:=1
    Ciklus amíg I≤N és nem Páros(Tömb(I))
       I:=I+1
    Ciklus vége

    Vane = (I≤N)
Eljárás vége
```

Először vizsgáljuk meg, mik azok a jellemzők, amelyek az *algoritmus* olvashatóságát segítik. A példa jól mutatja ezeket: beszédes változó-, típus- és eljárásnevek; kiemelt (jelen esetben vastagon szedett) kulcsszavak, programblokkok elejének, végének jelölése; egységbe szervezése („bekezdéses” írásmód). Megállapíthatjuk, ezek egy része stilisztikai jellegű, a programozón múlik, más része viszont a nyelvhez, a fejlesztői környezethez áll közelebb.

2. Érdekes programkészítési lépések

Négy, az oktatásban használatos nyelvet, nyelvcsaládot vizsgálunk: (1) Pascal/Delphi, (2) (Visual) C++, C#, (3) Java, (4) Visual Basic. E sort kiegészítjük kettő fiatalabb scriptnyelvvel is: (5) Ruby és (6) Python, amelyek a jövőben kaphatnak nagyobb szerepet az programozás tanításában.

Megvizsgáljuk, hogy a kódolási szabályoknak ismerete, léte vagy nem léte milyen hatással van a kódolási folyamatra, illetve a programszerkesztő, a programozási környezet milyen szolgáltatásokkal segíti, támogatja a kódolás folyamatát. Külön figyelmet szentelünk az I/O megoldására.

A következő részben megvizsgáljuk a hibaüzenetek információtartalmát és szigorúságát, azaz azt, hogy mikor deklaráltatik a fordító szerint késznek a program, s így mennyi és milyen „rejtett” hiba marad(hat) a következő fázisra.

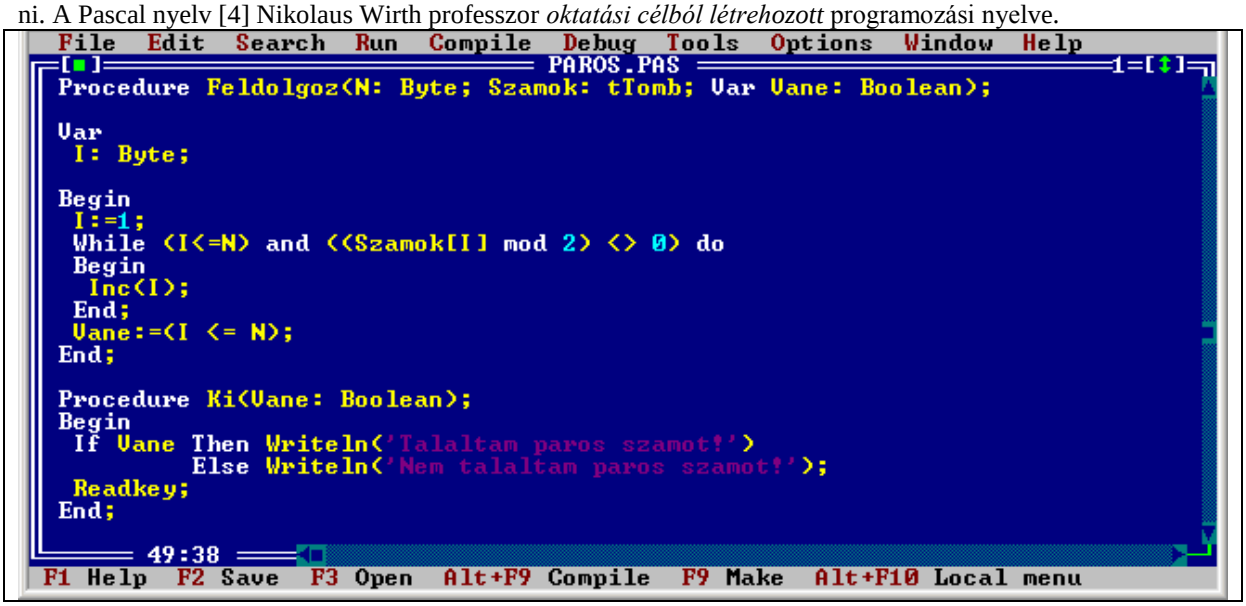
A tesztelés fázisában a környezetek nyomkövető rendszerének létét, szolgáltatásait hasonlítjuk össze.

2.1. Programszerkesztés

Első nehézségként merülhet fel (egy magyar diák számára), hogy a fenti nyelvek az angol nyelvhez állnak közel. Ha a diák rendelkezik kezdő angol nyelvtudással, az segíti a megértést, megkönnyíti a kódolás folyamatát. A környezetek, nyelvek ismertetésére nem fogunk kitérni.

2.1.1. Pascal/Delphi

Oktatási környezetben, Magyarországon, leggyakrabban a Pascal nyelvet használják a programozás tanítására. Két környezetet fogunk ismertetni, a „klasszikus" Borland Pascal 7.0-t (BP) és a Borland Turbo Delphi Explorert [2]. Az utóbbi környezet oktatási célra ingyenesen elérhető. Létezik egy Free Pascal névre hallgató ingyenes, a BP küllemével és működéséhez nagyon hasonló, ingyenes fejlesztői környezet is,[3] amiről ebben a cikkben nem fogunk értekezni. A Pascal nyelv [4] Nikolaus Wirth professzor *oktatási célból létrehozott* programozási nyelve.



1. ábra. Borland Pascal 7.0

A *programszerkezet* könnyen követhető és memorizálható. Elkülönített részek vannak a konstansoknak, a típusdefinícióknak, változók deklarálásának, a programtörzs helyének, minden eljárásban és függvényben. [5] A kódolás szabályait tekintve, a Pascal nyelv egyik fő jellemzője, hogy alulról felfelé építkezik, azaz a deklarálásnak mindig meg kell előznie a használatot. Tehát, egy eljárásban csak akkor lehet felhasználni egy változót, ha azt deklaráltam, hasonlóan, egy eljárást is csak akkor lehet meghívni, ha már deklaráltatott (legalább a fejsora `forward`-dal). A tömbök statikusak (bár a Delphiben már megjelenik egy speciális dinamikus tömb is), tehát a méretének ismertnek kell lennie már fordításkor (pontosabban az index típusának kell rögzítve lennie), azaz legkésőbb a kódoláskor. [6]

A Pascal jó példa az *értelmes alapszavak* tekintetében is. Minimális angoltudással könnyen megfejthető a program működése. Az I/O kezelése egyszerű, a parancsok egyértelműen utalnak a beolvasásra/kiíratásra (`read, readln, write, writeln`). Előnyösnek mondható, hogy a nyelv nem különbözteti meg a kis- és nagybetűket, így ennek megjegyzésére nem kell időt fordítani.

Hibák veszélyforrása, hogy nem minden típusazonosító védett alapszava a nyelvnek (pl. `byte, word, integer`…). A típusdeklarációs részben előfordulhat a következő típusdefiníció: `byte = string`, a fordító nem fog hibát jelezni. Nyilván típusütközést fog tapasztalni a programozó, ha ezek után a `byte` típust pozitív egészként kívánja használni.

[2] http://www.turboexplorer.com/
[3] http://www.freepascal.org

A Pascalban komoly veszélyeket rejt magában a hozzáférési jog nélkül szervezett paraméterátadás, mellesleg nehezen érthető a `Const`-tal történő paraméterátadással való összevetése.

Nem minden esetben következetes a nyelv. Az aktuális paramétereket vesszővel, a formálisakat pontosvesszővel választja el, hasonlóképpen az a szabály sem igaz, hogy minden sor végére ki kell tenni a pontosvesszőt, lásd `if-then-else`.

Az *összetett szerkezetek* eleje-vége jelzése sem egységes: pl. `begin-end, record-end, while-end, repeat-until`.

Az elágazások egymásba ágyazhatóak, így létrehozhatók ránézésre többféleképpen értelmezhető szerkezetek:

```
if a=5 then
if a=4 then
else writeln(a);
```

A fenti kódrészlet példázza az ún. „csellengő else” problémát (C++, C#, Java nyelvekre is jellemző). A kódból nem derül ki, melyik feltételhez tartozik az `else` ág. A nyelv nem követeli meg ebben az esetben a `begin-end` használatát, amely egyértelművé tenné a dilemmát.

Tekintsük először a BP verziót. Az 1. ábrán jól látható, hogy a nyelv kulcsszavai ki vannak emelve (fehérrel), illetve a számokat és a szövegeket (Stringek) különböző színekkel (kék, illetve lila) jeleníti meg a szerkesztő.[4]

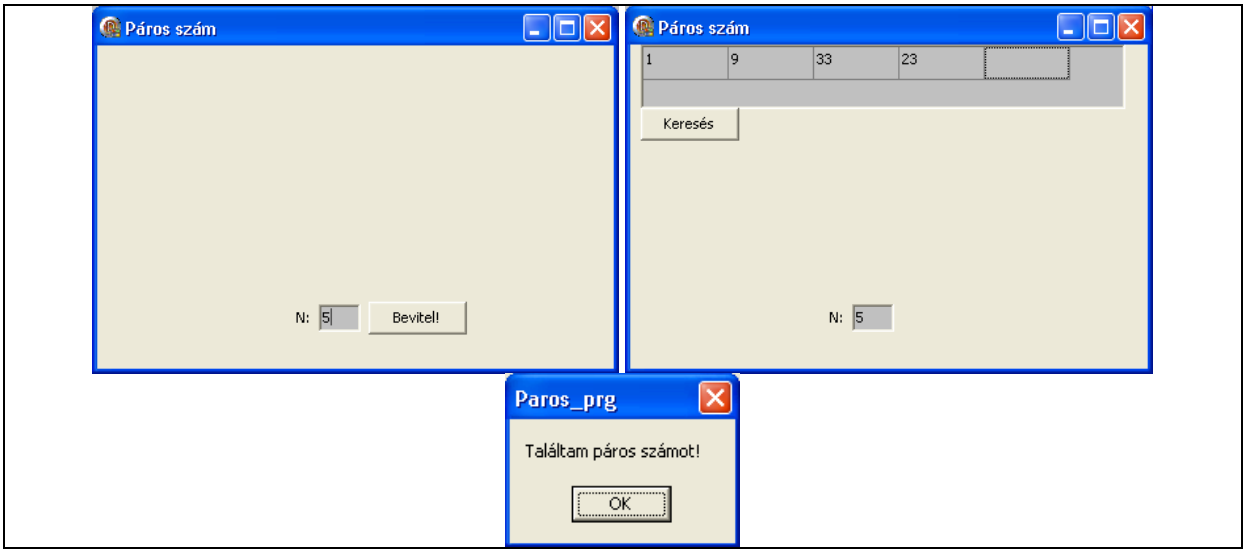
A Delphiben is lehetőségünk van hasonló elven (mint feljebb) konzolos programokat írni, ha használjuk az `{APPTYPE CONSOLE}` direktívát. A fentiekkel való hasonlóság miatt a környezet ezen szolgáltatását nem elemezzük.

A környezet komponensorientált „arca” sokban különbözik a fentitől. Ez a rendszer már objektum-orientáltan gondolkodik, ezt a paradigmát kínálja fel. Programunk objektumként fog viselkedni, a komponensek az objektum változóit manipulálják az objektum metódusain keresztül. Hogyan befolyásolja ez a tény az érthetőséget? A paradigmaváltás és a komponensorientáltság miatt a beolvasás és a kiíratás megvalósítása markánsan különbözik az előbbiekben tárgyalttól. A lényegi rész (*Feldolgoz* eljárás), a kódolás szempontjából szinte ugyanaz (lásd 2. ábra).

```
40  procedure TParos.G_SzamokClick(Sender: TObject);
    
    Var
     I: Byte;
     Vane: Boolean;
45
    begin
     I:=0;
     While (I<=(StrToInt(N.Text)-1)) and ((StrToInt(Szamok.Cells[I,0]) mod 2) <> 0) do
        Begin
50        Inc(I);
        End;
     Vane :=(I<=(StrToInt(N.Text)-1));
     If Vane then ShowMessage('Találtam páros számot!')
              else ShowMessage('Nem találtam páros számot.');
    end;
```

2. ábra. Delphi – a lényegi rész

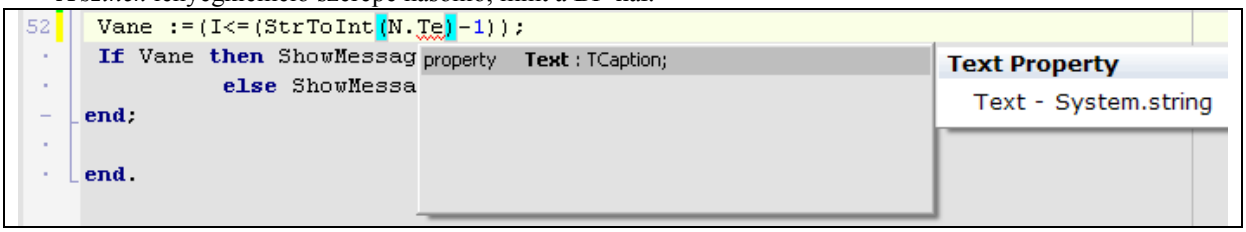
[4] A színeket szabadon beállíthatja a felhasználó, a kép egy ilyen példát mutat be.



3. ábra. Delphi – „input-output”

A *be- és kimenet különbözősége* (lásd 3. ábra) felveti azt a kérdést, hogy át tudjuk-e hidalni az algoritmus szintjén ezt a különbséget? Egyáltalán át kell-e hidalni? Vizuális fejlesztő környezet esetén, ahol a be- és kimenet megvalósítása komponenseken keresztül történik, az előkészítést is „vizuálissá” kell tenni. Az algoritmusbeli I/O-rész (amit korábban nem részleteztünk) szerepe alig több mint ezen információk specifikálása. Tehát ilyen környezetben az I/O megszervezésénél igen sok marad a kódolásra.

A *színek* lényegkiemelő szerepe hasonló, mint a BP-nál.



4. ábra. Delphi – „intellisence”

Programszerkesztés közben – a többi 4GL környezethez hasonlóan – a szerkesztő segít a programozónak kiválasztani, hogy az adott komponens melyik metódusát vagy property-jét érheti el. Így nem kell pontosan fejből tudnia sem az azonosítót, sem a paraméter(ek) típusát (lásd 4. ábra). További online segítség: az összetartozó zárójeleket azonos színnel emeli ki, mikor a kurzorral valamelyiken megállunk vagy áthaladunk. Ez a funkció a zárójelezési hibák kiküszöbölésében és felderítésében jelent támaszt.

2.1.2. (Visual) C++, C#

A nem vizuális C++ programok [7] írásához széles körben elterjedt a Dev-C++ nevű ingyenes program.[5]

A C nyelv és minden leszármazottja sokkal engedékenyebb, mint az előzőekben tárgyalt Pascal. Nincsenek külön típusdefiníciós és deklarációs részek. A változókat ott deklaráljuk, ahol éppen szükségét érezzük. A nyelv alulról felfele építkezik, hasonlóan az előbbi nyelvhez.

Sokan dicsérik a C nyelv leszármazottjait tömörségükért. Például tömören foglalhatók keretbe a programblokkok (a `{` és `}` jelekkel), mégis világosan elkülönülnek környezetüktől. (L. 5. ábra) E tömör fogalmazásmódra negatív példaként említhetők a „`++`” és „`--`” operátorok [3]. A kódból nem derül ki a különbség az „`i++`” és „`++i`”, illetve az „`i--`” és „`--i`” között, így nem segítik a kód olvashatóságát, átláthatóságát. A két kifejezés mellékhatása

[5] http://www.bloodshed.net

ugyanaz, az `i` változó értéke inkrementálódni fog (és a legtöbbször ilyen szerepben használatos), ellenben fő hatásában különböznek, miszerint kiértékeléskor az „`i++`” kifejezés az eredeti, míg a „`++i`” az eggyel megnövelt értéket fogja tartalmazni. Az „`i--`” és „`--i`” működési elve ugyanez, azzal a különbséggel, hogy eggyel csökkenti az `i` változó értékét. Tehát összetett kifejezésnél nagyon nem mindegy, hogy a két, egymástól csupán szintaktikai „apróságban” különböző operátor közül melyiket használja a programozó.

Az alapszavak érthetőek, a beolvasás és a kiíratás esetében a „`cin`” és a „`cout`” nem utalnak egyértelműen a beolvasás és kiírás funkciójára; több „etimologizálásra” van szükség a kulcsszavak megértéséhez. Ezzel szemben kifejező az adatfolyam irányának jelölése (`cin >> n`, azaz a konzol inputról az `n`-be fog „vándorolni” az érték).

```
void be(int n, int szamok[])
{
    for (int i=0; i<n; ++i)
    {
        cout << "Kerem a(z) " << (i+1) << ". szamot: ";
        cin >> szamok[i];
    }
}

void feldolgoz(int n, int szamok[], bool& vane)
{
    int i = 0;
    while ((i < n) && ((szamok[i] % 2) != 0))
    {
          i++;
    }
    vane = (i < n);
}

void ki(bool vane)
{
     if (vane) cout << "Talaltam paros szamot!\n";
     else cout << "Nem talaltam paros szamot.\n";
}

int main(int argc, char *argv[])
{
    int n;
    cout << "Ez a program megmondja, hogy van-e paros szam a tombben.\n";
    cout << "Hany darab szam van a tombben?\n";
    cin >> n;
    int szamok[n];
    be(n,szamok);
    bool vane = false;
    feldolgoz(n,szamok,vane);
```

5. ábra. Dev-C++

Hibaforrás lehet C-jellegű nyelvek összetett utasításainak törzse helyett véletlenül írt üres (;) utasítás. Hasonlóképpen C++-ban és Javaban ha nem írja le a programozó a `break` utasítást a többágú elágazás (`switch`) ágainak végére, akkor nem csak az az ág fog lefutni, ahol teljesült a feltétel, hanem az összes többi, amely alatta van. Így működése teljesen eltér a kétágú „klasszikus” elágazás működésétől (`if-then-else`). A fordító nem követeli meg a `break` utasítást, így hibák keletkezhetnek ebből a hiányosságból. A C# nyelv fordítója már kötelezővé teszi a `break` elhelyezését az ágak végére.

A C++ is erősen típusos nyelv, mégis a kifejezések típushelyességének ellenőrzése közben a fordító nem jelez hibát típusütközésnél, csak figyelmeztetést ad, és automatikusan végzi el a típuskonverziót. Van, amikor helyesen, van, amikor nem, így extra hibaforrás kerül a kódba és annak érthetősége is csorbul.

A C-jellegű nyelvek közös tulajdonsága, hogy az azonosítókban megkülönbözteti a kis- és nagybetűket. Ennek hátránya, hogy a metódusokat (is) betűhelyesen kell megjegyezni. Igaz az is viszont, hogy a saját azonosítók megalkotásánál erre építeni lehet, a saját névkonvenciók kialakításánál haszonnal kamatoztatható.

A C++ vizuális testvére [8] is ingyenesen letölthető. A Visual Studio 2008 Express Edition [6] (VS) tartalmazza a C++, C# és Visual Basic nyelveket. Az Express kiadás csak az alap funkciókat tartalmazza, ami a középiskolai oktatáshoz elegendő.

[6] http://www.microsoft.com/express/

A 6. ábrán láthatjuk a korábbi feladat megoldásának lényegi részét. A nyelv megkülönbözteti az objektum- és az osztályszintű metódusokhoz való hozzáférést. Előbbit „: :”-al, az utóbbit „->”-al jelöli. Kérdés, hogy ezt a megkülönböztetést el lehet-e magyarázni egy kezdő programozó számára? Rengeteg „fölösleges” információt tartalmaz a forrásfájl pl. a komponensek inicializálása is itt kapott helyet, amely „bonyolult hatást” kelt.

A környezet hasonló felépítésű, mint a Delphi. Egységesebbek – így könnyebben megjegyezhetőek – a számunkra fontos metódusok és property-k elnevezése, mint a Delphiben. Pl. Delphiben a gomb felirata esetében a Caption-t, a szövegmező esetén a Text-et használja, míg a VS mindkettő esetben az utóbbit.

```
private: System::Void g_Keres_Click(System::Object^  sender, System::EventArgs^  e) {
             int i = 0;
             int n = System::Convert::ToInt32(N->Text);
             while ((i < n) && ((System::Convert::ToInt32(Szamok[i,0]->Value) % 2) != 0))
                {
                    i++;
                }
             bool vane = (i < n);
             if (vane) MessageBox::Show("Találtam páros számot!");
             else MessageBox::Show("Nem találtam páros számot!");
         }
```

6. ábra. Visual C++ – egy jellemző kóddarab

Hasonlóan a Delphihez, ez a környezet is legördülő listákkal segít megtalálni a szükséges komponenst.

```
private void g_Keres_Click(object sender, EventArgs e)
{
          int i = 0;
          int n = System.Convert.ToInt32(N.Text);
          while ((i < n) && ((System.Convert.ToInt32(Szamok[i, 0].Value) % 2) != 0))
             {
                 i++;
             }
          bool vane = (i < n);
          if (vane) MessageBox.Show("Találtam páros számot!");
          else MessageBox.Show("Nem találtam páros számot!");
}
```

7. ábra. C#

A 7. ábrán a C#-os megoldás látható. Megállapítható, hogy ez a nyelv [9] sokkal letisztultabb, érthetőbb, összehasonlítva a C++-szal. Lehetőség van konzolos programok írására, hasonlóan a Delphihez. Az osztály és objektum szintű metódusok tagkiválasztó operátora egységes (`.`), illetve a paraméterkezelésnél világosan elkülönülnek az értékszerinti (pl.: `int a`), referencia (pl.: `ref int a`) és kimenő paraméterek (pl.: `out int a`). Abban az esetben, ha olyan függvényre van szükség, mely egynél több értéket szolgáltat vissza, akkor lesz szükség kimenő paraméterek létrehozására. Az I/O parancsai (`Read, ReadLine, Write, WriteLine`) egyértelműen utalnak szerepükre.

A fejlesztőkörnyezet több kulcsszót emel ki, az eljárások fejsorai olvashatóbbak és a forrásfájl csak a kódot tartalmazza, a komponensek adatai más fájlban foglalnak helyet.

2.1.3. Java

A Java [10] programozásához is léteznek ingyenes eszközök. Ezek közül az egyik az Eclipse [7]. A nyelv kódolási szabályai nagyon hasonlatosak a C++-hoz. Azonban meg kell említenünk néhány specifikus jellemzőt.

A Java is objektumorientált nyelv, így mielőtt bármit is elkezdenénk, minimálisan szükségünk lesz egy osztálydefinícióra és egy main eljárásra. Még ha egy „Helló, Világ!”-szerű programot akarnánk is írni, a „lényegi rész” egy sora mellé még legalább hat, esetleg érthetetlen sort szükséges bebillentyűzni. Kérdés, ez mennyiben segíti a feladat

[7] http://www.eclipse.org/downloads

megoldását, szükséges-e tisztában lenni az objektumorientáltsággal az ilyen fajta programok megírása előtt? (L. 8. ábra)

```
1  import java.util.*;
2
3  public class Paros {
4
5      /**
6       * @param args
7       */
8      public static void main(String[] args) {
9          // TODO Auto-generated method stub
10         System.out.println("Ez a program eldönti, hogy van-e páros szám a sorozatban.");
11         int n = 0;
12         System.out.println("Add meg a tömb elemszámát!");
13         Scanner olvas = new Scanner(System.in);
14         n = olvas.nextInt();
15         int[] szamok = new int[n];
16         be(n, szamok);
17         boolean vane = feldolgoz(n, szamok);
18         ki(vane);
19     }
```

8. ábra. Java – főprogram

A Java paraméterátadása különbözik az eddig jellemzett nyelvekétől. Csak érték szerinti átadást ismer, tehát a paraméter helyén levő értékről másolat készül. Az eljárás ezt a másolatot használja, nem az eredetit. Ezért kellett függvénnyel meghatározni a „`vane`” értékét (lásd 8. ábra). Ez a szabály csak az egyszerű típusokra vonatkozik. Pl. tömb esetében az átadásnál a referenciáról készül másolat, így a tömbben véghezvitt változtatások megmaradnak. Sajnálatos, hogy ebben a kérdésben nem egységesek a nyelv szabályai. Paraméterként csak deklarált és inicializált értékek adhatóak át.

I/O tekintetében a nyelv filozófiája nem egységes. A kiíratás művelete (`System.out.println`) jól tükrözi annak célját és működését. Ezzel szemben a beolvasás jóval bonyolultabb. A fenti példában a `Scanner` osztály metódusait használtuk fel. Látható, hogy a beolvasás olyan, mint egy értékadás, így távolabb viszi a kódot az algoritmikus nyelvtől.

```
public static boolean feldolgoz(int n, int[] szamok)
{
    int i = 0;
    while ((i < n) && ((szamok[i] % 2) != 0))
    {
        i++;
    }
    return (i < n);
}
```

9. ábra. Java – a feldolgoz függvény

A fejlesztői környezet sok mindenben támogatja a programozót. Az eddig megismert segédeszközökön túl a zárójelezés esetében kiteszi helyette írás közben a csukó zárójelet, illetve ha valamelyik oldalon több vagy kevesebb lenne, a hibás, a többletet okozó zárójelet pirossal aláhúzza. Mikor a kurzor egy változó nevén van, a szerkesztő kiemeli az összes azonos nevűt (Lásd 9. ábra).

2.1.4. Visual Basic

A Visual Basic (VB) [11] ugyancsak része a fentebb említett VS környezetnek. Valójában a VB a Basic nyelv egy implementációja, amely az alapnyelv strukturálatlan felépítését hivatott kibővíteni strukturált és objektum orientált elemekkel, de ezek erősen kötődnek a fejlesztői környezethez. Más nyelvre való továbblépés sem egyszerű az elavult gyökerek miatt. [12]

```
Private Sub g_keres_Click(ByVal sender As System.Object, ByVal e As
    Dim i As Integer
    i = 0
    Dim n As Integer
    n = Convert.ToInt32(Me.N.Text)
    Szamok(n, 0).Value = 0
    While i < n And Convert.ToInt32(Szamok(i, 0).Value) Mod 2 <> 0
        i += 1
    End While
    Dim vane As Boolean = (i < n)
    If vane Then
        MessageBox.Show("Találtam páros számot")
    Else
        MessageBox.Show("Nem találtam páros számot")
    End If
End Sub
```

10. ábra. Visual Basic

Nyelvi szempontból különbözik az első háromtól. Egy utasítás egy sorba írandó, így fölösleges azok pontosvesszővel való lezárása. Habár a nyelv típusos nyelv, nem annyira szigorú, mint a Pascal alapúak. A változó deklarálására használt „`Dim`” (lásd 11. ábra) annak méretét és elemeinek típusát adja meg. Pozitív példa, hogy külön kulcsszó (`ByVal`) különbözteti meg az értékszerinti paraméterátadást a címszerintitől). Hasonlóan külön kulcsszó létezik a függvényeknek (`Function`) és eljárásoknak (`Sub`), így közelebb áll az algoritmikus nyelvünkhöz.

Sokat segít a megértésben, hogy minden ciklusnak és elágazásnak könnyen felismerhető és megjegyezhető kulcsszavú kezdete és vége van (pl. `While … End While`), így könnyű olvasható kódot írni.

A környezet mindent megtesz azért, hogy „kitalálja” a programíró helyett, melyik kulcsszavat vagy metódust szeretne használni. Amint elkezd valamit írni a programozó, rögtön látni fogja a lehetőségeket. Ami kifogásolható ebben a rendszerben, hogy egy komponens nevének kiválasztásánál az enter leütésével a kurzor új sorra ugrik, holott még a program írója nem jutott el a megfelelő tulajdonságig. Ctrl-Enter leütésével lehet orvosolni a problémát, s akkor a megfelelő property kiválasztásáig ugyanabban a sorban marad a kurzor. Ez a tulajdonság egyedi a VB-ben, megnehezíti a kezdőlépéseket, ha nem ismert, milyen billentyűkombinációt kell használni.

Alapértelmezésben a fordító minden esetben kiértékeli a teljes logikai kifejezést, ezért volt szükséges felvenni egy páros számot a tömb végére (N+1. elemként), hogy mindenképp leálljon a ciklus. Ez nem szerencsés, mert el kell térni az algoritmustól. Tapasztalatunk szerint, ez a fajta kiértékelési stratégia csak ebben a környezetben fordult elő a vizsgáltak közül. A többi környezet a lusta stratégiát használta alapbeállításként.

2.1.5. Ruby

Egy igen fiatal nyelvről [13] van szó (1995-ben született), amely sokat örökölt a Perl, Python és Smalltalk nyelvektől. Hasonlóan a fent tárgyaltakhoz, a Ruby fejlesztői környezete [8] is ingyenes.

[8] http://www.rubyonrails.org/down

Maga a nyelv kinézetre sokban különbözik az eddig tárgyaltaktól. Jelentősen eltér a kulcsszavak alkalmazásában.

```
1  def be(szamok)
2    puts 'Ez a program eldonti, hogy van-e paros szam a tombben'
3    puts 'Ird be N erteket!'
4    n = 0
5    n = Integer(gets)
6    for i in 1..n
7      puts 'Ird be a ' + i.to_s + '. elemet:'
8      szamok[i] = Integer(gets)
9    end
10   return n
11 end
12
13 def feldolgoz(n, szamok)
14   i = 1
15   while (i <= n) and ((szamok[i] % 2) != 0)
16     i = i + 1
17   end
18   return (i <= n)
19 end
20
21 def ki(vane)
22   if (vane) then puts 'Talaltam paros szamot'
23     else puts 'Nem talaltam paros szamot'
24   end
25 end
26 #Főprogram
27   szamok = Array.new
28   n = be(szamok)
29   vane = feldolgoz(n, szamok)
30   ki(vane)
```

11. ábra. Ruby

A 11. ábrán látszik, hogy a program nagyon nem kezeli szigorúan a típusokat, egy változó típusa az értékadásnál dől el (hasonlóan a PHP-hoz és a Perl-hez). Ez a szintaktikai „lazaság” nem segíti a program olvashatóságát és sok hibalehetőséget rejt magában.

A paraméterátadás hasonlóan működik, mint a Java-ban.

Zavaró lehet, hogy a függvény nincs markánsan megkülönböztetve az eljárástól. A fejsorát tekintve nem különböznek egymástól. Az különbség a törzsben valahol felbukkanó „`return`” utasítás léte, ami értéket ad az eljárásnak, így téve függvénnyé azt.

A ki- és beolvasás szintakszisa érthetetlen eltéréseket mutat. Szöveg beolvasás vagy kiíratása esetén az utasítás mögé kell írni a változó nevét vagy a szövegkonstanst (`gets 'szöveg', puts s`). Számok esetén a beolvasás értékadásnak látszik, hasonlóan a Javahoz (`n = Integer(gets)`), a kiírásnál pedig típuskonverzióra van szükség (`puts n.to_s`).

A nyelv elég fiatal, ezért a programszerkesztők még nincsenek felvértezve az összes „kényelmi” szolgáltatással. Általában igaz (hasonlóan az ábrán látható SciTE programhoz), hogy az alapszavak kiemelésén kívül más segítséget nem nyújt a programozóknak.

2.1.6. Python

Ez a nyelv [14] is igencsak fiatal, 1991-ben született. Ez is ingyenes programszerkesztővel bír.[9] Az alapcsomaggal telepített szerkesztőben látható a példa megoldása:

[9] http://www.python.org/download/

```
# -*- coding: cp1250 -*-
def be(szamok):
    n = input('Add meg N értékét: ')
    for i in range (0, n):
        szamok.append(input('Add meg a tomb '+str(i+1)+'. értékét! '))
    return n

def feldolgoz(n, szamok):
    i = 0
    while i < n and (szamok[i] % 2) != 0:
        i = i + 1
    return (i < n)

def ki(vane):
    if vane:
        print "Találtam páros számot!"
    else:
        print "Nem találtam páros számot"
# Főprogram
print 'Üdvözöllek, ez a program eldönti, hogy van-e páros szám a tömbben'
szamok = []
n = be(szamok)
vane = feldolgoz(n,szamok)
ki(vane)
```

12. ábra. Python

A program szerkezetében és nyelvi elemeit tekintve nagyon hasonlít a Rubyhoz (utóbbi egyik ősének tekinti a Pythont). Az előző nyelvekhez képest új „megoldással” rukkol elő. Nincsenek `Begin-End` párok, az összetartozó programblokkok ugyanabban a bekezdésben vannak, így a programozónak nincs más lehetősége, újabb bekezdésben kell írnia a ciklushoz vagy elágazáshoz tartozó részeket, különben nem úgy fog viselkedni a program, ahogy azt elvárja. Ez az ún. margószabály nagyban hozzájárul a kód olvashatóságához. Vegyük észre, hogy ez az algoritmikus nyelvünkkel való összhang miatt tovább egyszerűsíti a kódolást.

Kódolási szabályokat tekintve teljesen megegyezik a Ruby nyelvvel.

Fura a nyelv számlálós ciklusa: szokatlan a szintakszisa és ebből nem kitalálható a szemantikája. A „`for i in range(1, n)`” azt sugallja, hogy a ciklusmag $1 \leq i \leq n$ esetére fog lefutni. Nem ez az igazság, ugyanis a ciklusmag $1 \leq i < n$ esetére fog lefutni, ezért látható a 12. ábrán, hogy 0 és n között fog n-szer lefutni a ciklus.

A programszerkesztő igen egyszerű, a színezésen kívül nem nyújt több, az olvasást segítő szolgáltatást.

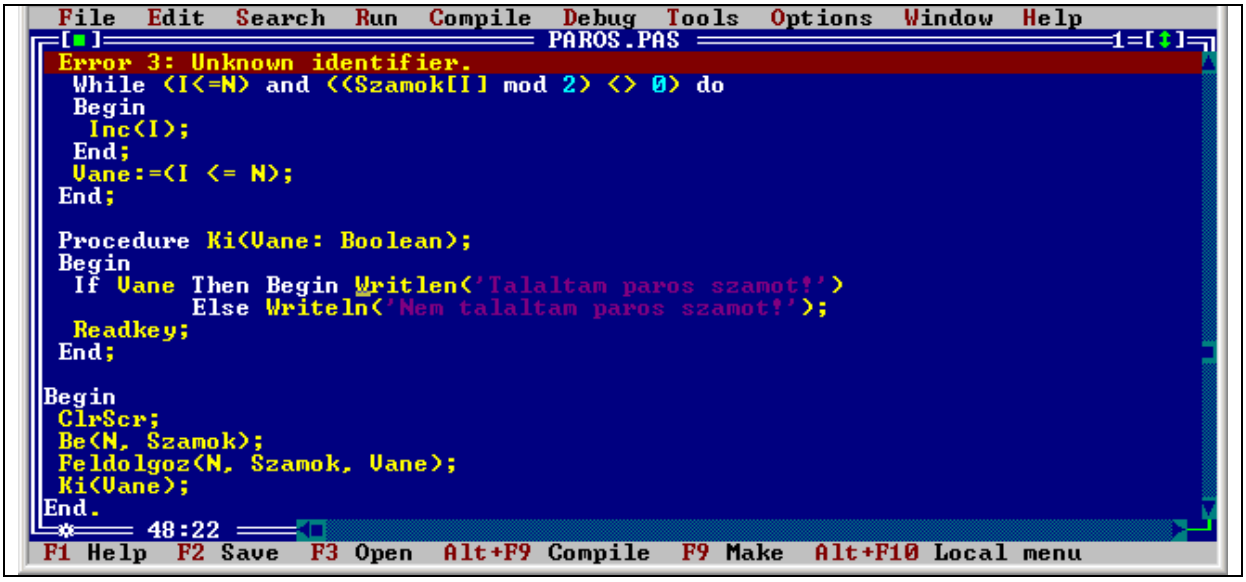
2.2. Fordítás, szintaktikai és statikus szemantikai hibák javítása

A nyelvek fordítói és az általuk küldött információk (hibaüzenetek) tartalma és minősége a környezet tulajdonságaihoz állnak közelebb. Fontos, hogy a hibaüzenetből a programozó tudja, hogy milyen jellegű hiba történt, hol található a hiba és hogy miképp lehet megoldani azt.

Mitől jó egy hibaüzenet? Jó, ha a hibaüzenet tömör, udvarias, következetes, pozitív, építő, és aktív nyelvezetet használ. Jól karakterizálja a hibát és szükség esetén közvetlenül a súgó megfelelő oldala is elérhető róla. [15]

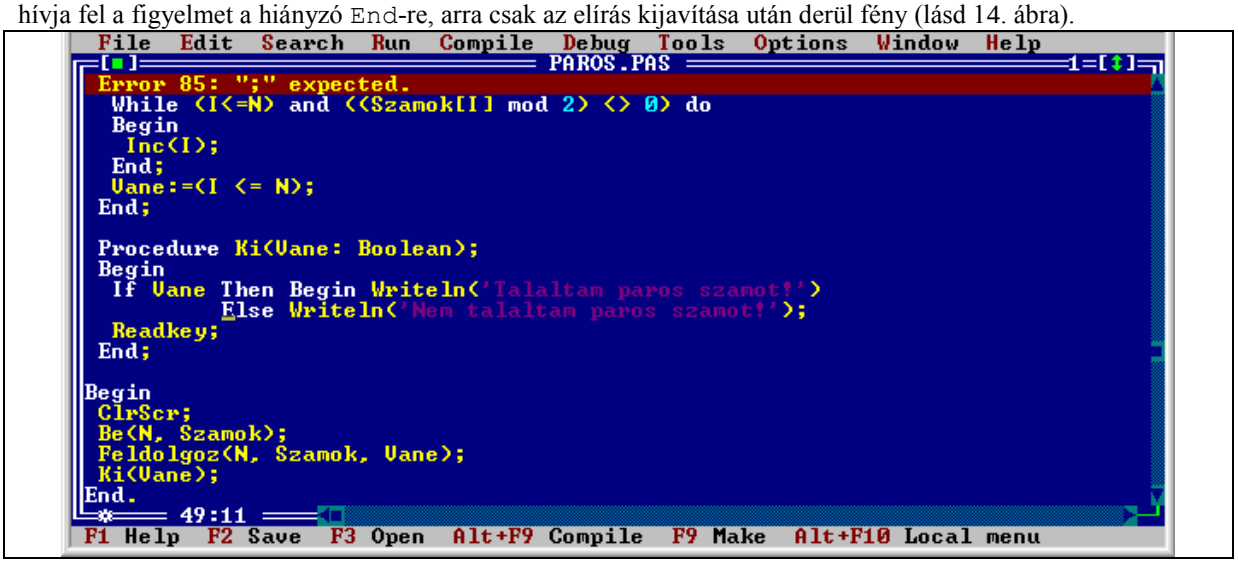
Jelen írás keretein belül nem lehetséges minden fajta típushibára példát mutatni és elemezni, ezért két hibát elemzünk: Tegyük fel, hogy a tanuló elgépelte az egyik utasítást/változó azonosítóját és nem zárt le egy programblokkot.

A *Borland Pascal* környezet fordítójának tulajdonsága, hogy az első adandó hibánál leáll, és csak azt jeleníti meg, így ha több hiba van a programban, legalább annyiszor le kell fordítani a kódot. Ha felderítené az összes hibát, esetleg időt lehetne megspórolni. Viszont így a következményhibák hatásai nem terhelik a programozót, amely oktatási szempontból előnyös.



13. ábra. Hibaüzenet BP-ban

A hibaüzenetek nem túl beszédesek, és nem is mindig értelmesek. Egy változóazonosító, illetve eljárás nevének elírásakor ugyanazt az üzenetet kapjuk („Ismeretlen azonosító”, lásd 13. ábra). Előnyös, hogy ha hibát talál a fordító, a kurzor rögtön a hibás szóra, vagy legalábbis a környékére ugrik. Az ábrán látható, hogy ez a hibaüzenet nem hívja fel a figyelmet a hiányzó `End`-re, arra csak az elírás kijavítása után derül fény (lásd 14. ábra).

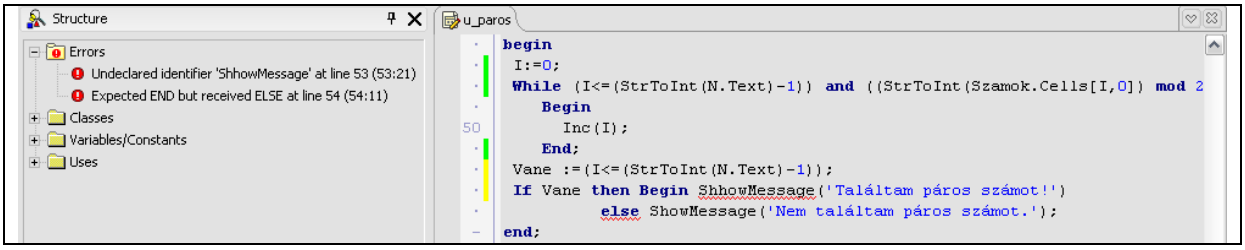


14. ábra. A hibaüzenet nem pontos

Vagy talán mégsem. A fordító a pontosvesszőt hiányolja, holott a szükséges `End` a hiba. Pusztán a hibaüzenetből a kezdő programíró nem fog rájönni, mit rontott el. A súgó sem ad több segítséget.

Összességében a hibaüzenetek tömörek, de nem nyújtanak ennél több segítséget, illetve nem elég specifikusak.

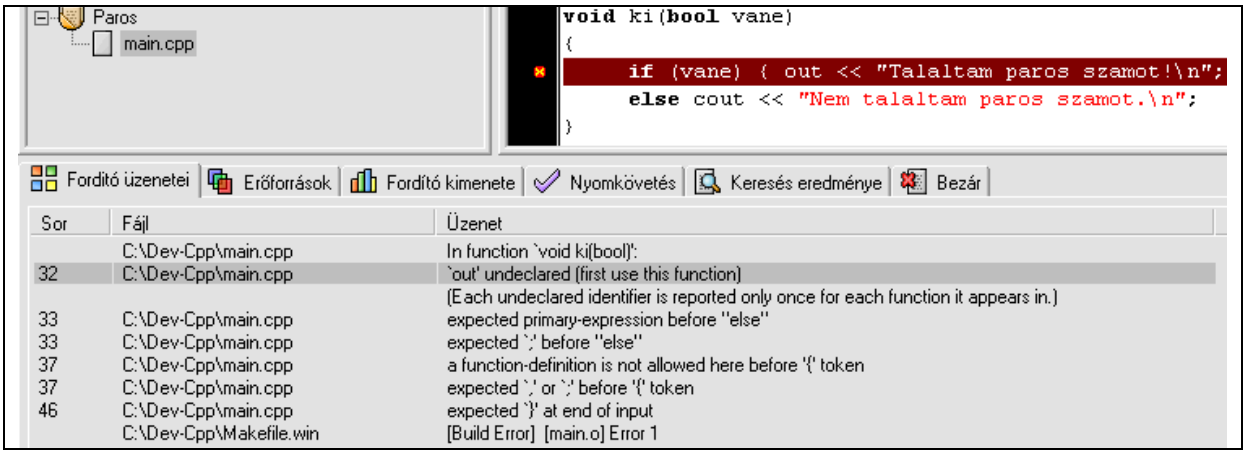
A *Delphi* már nemcsak az első hibaüzenetet, hanem a lehetséges *összest* kiírja. Sőt hibajelzéseket *már kódolási időben* is ad: a háttérben dolgozó fordító folyamatosan figyelmeztet az éppen lekövetett szintaktikus hibákra (lásd 15. ábra, bal oszlop). Fordításkor kijelöli a hibás sort és lent kiírja az üzeneteket. Ez a megoldás jobban megmutatja, hogy hol történt a hiba.



15. ábra. Delphi: azonosítja a hibákat még fordítás előtt

A kódolás idejében, de még a programozó által indított fordítás előtt detektált hiba sok időt takaríthat meg a fejlesztőnek. A konkrét példánk esetében kódolás közben két hibajelzés is megjelenik, amelyek segítenek a probléma érzékelésében, sőt a megoldás megtalálásában is: a pirossal aláhúzott „`ShhowMessage`” és „`else`” is felhívja a figyelmet az elírásra. A hibaüzenetek tömörek, helyesek, rávilágítanak a problémára. A súgó itt jó segítőtárs a megoldásban.

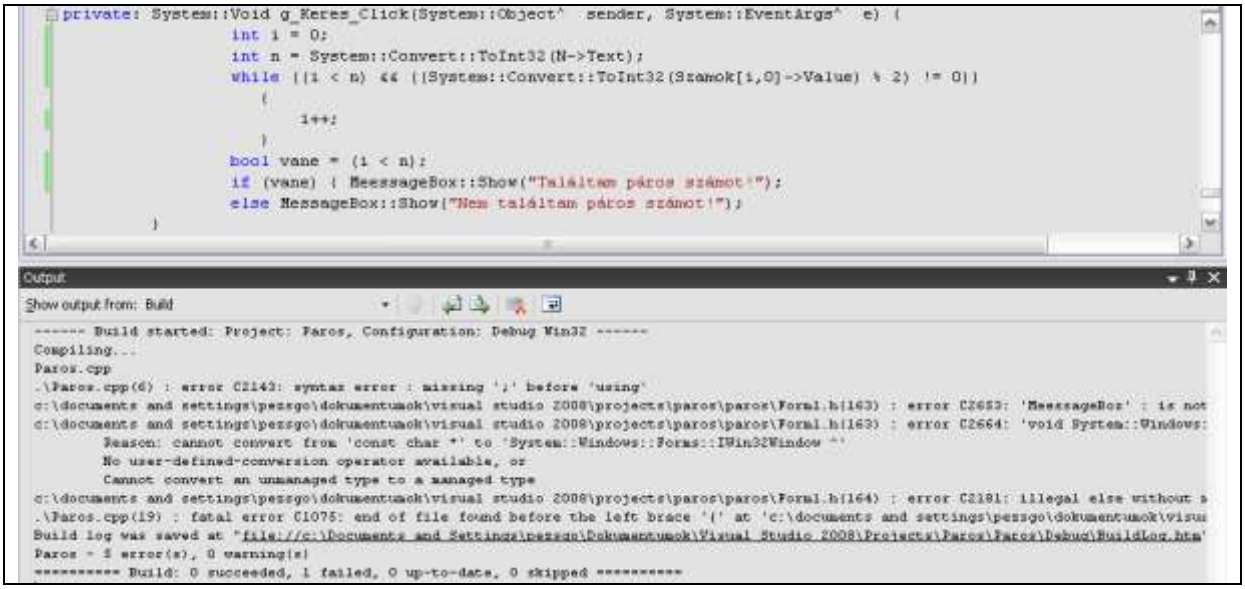
A Dev-C++ által használt fordító is tudósít az összes fellelhető hibáról.



16. ábra. C++: a hibaüzenetek nem segítenek hatékonyan

A fenti hibalista segít könnyedén detektálni az elgépelést, azonban a többi öt hibaüzenet nem fejezi ki világosan, hogy lemaradt egy záró kapcsos zárójel. Egy tapasztaltabb programozó kitalálja a hibaüzenetekből, hogy valóban, nem egyezik meg a nyitó és csukó kapcsos zárójelek száma, ám egy kezdő számára nem nyújt hatékony segítséget, összehasonlítva a Delphi hibaüzenetével.

A Visual C++ hibaüzenetei hasonlóan kevés információ tartalommal bírnak, mint a fent említett környezet.



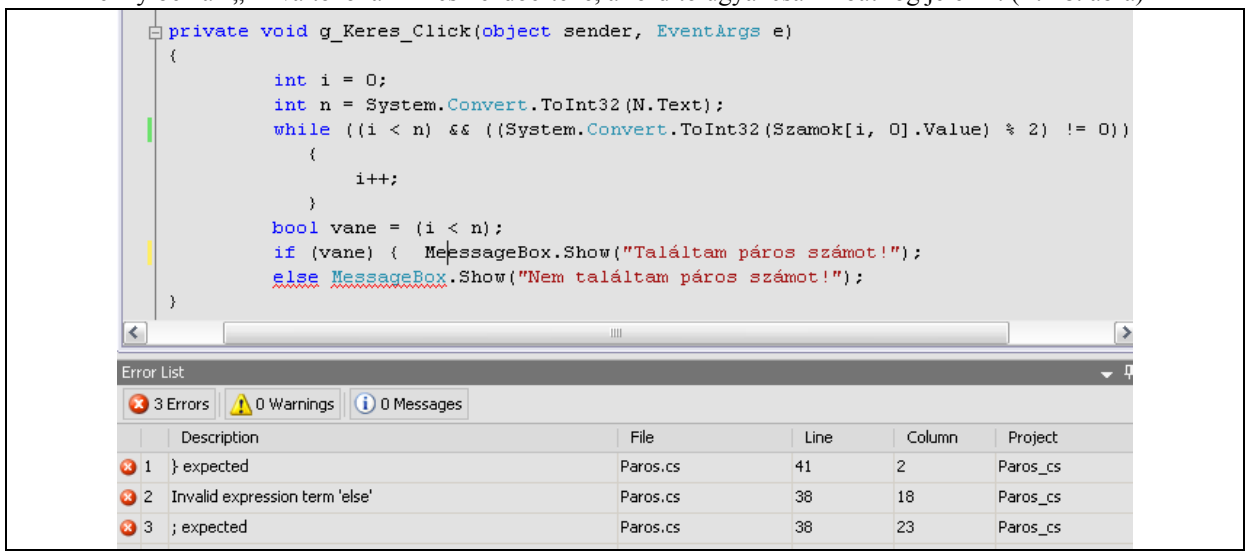
17. ábra. Visual C++: Bonyolult hibaüzenetek

A fenti ábrán látszik, hogy nincs automatikusan kijelölve az első hibás sor, ehhez végig kell kattintani a hibalista sorait. Az elgépelésre figyelmeztető hiba hamar megtalálható, de a hiányzó csukó kapcsos zárójelre, egzakt módon, semmi sem utal.

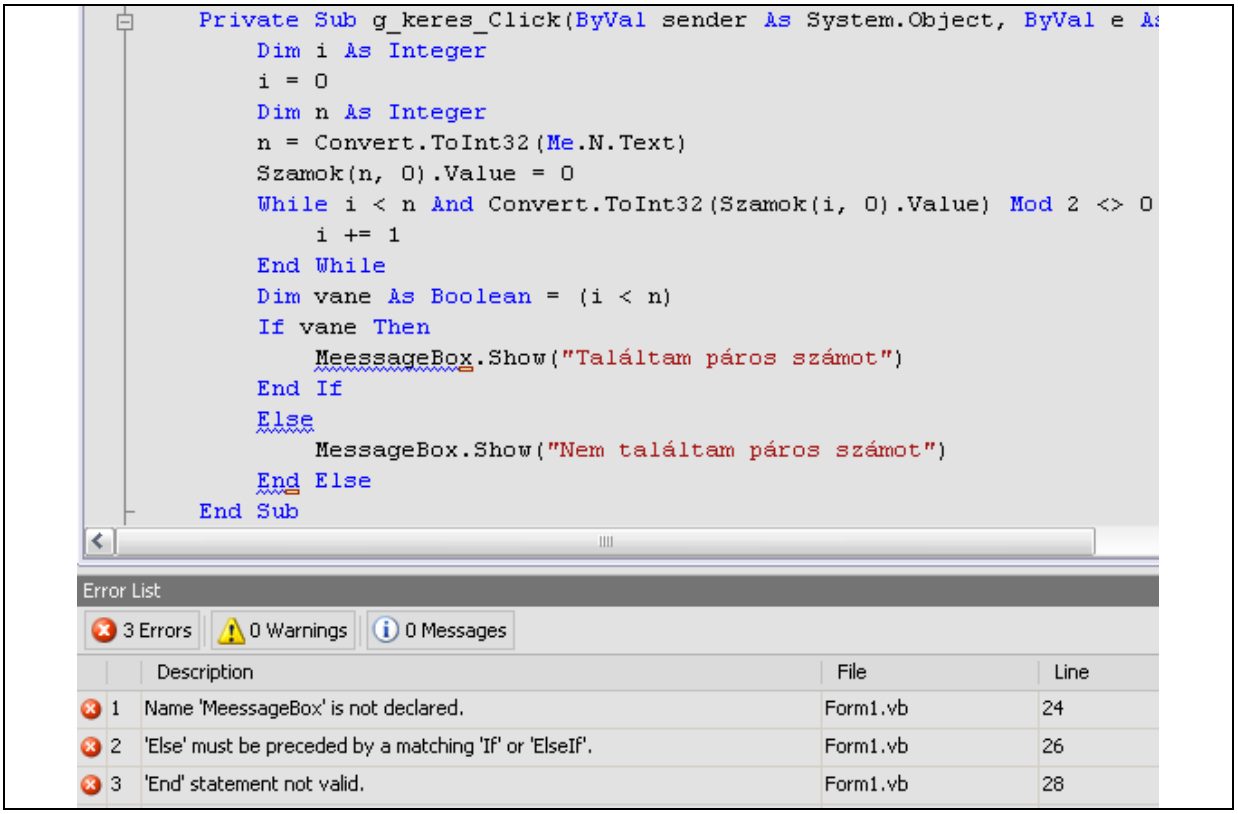
A Visual C# fordítója és hibakezelője – a nyelvhez hasonlóan – sokkal barátságosabb és lényegre törőbb. A 18. ábrán látható, hogy fordítás előtt ki fog derülni, hogy hiányzik a csukó kapcsos zárójel. Igaz, nem a hiány tényleges helyén jelzi, de közvetett módon arra sarkallja a programozót, hogy számolja össze a nyitó és csukó kapcsos zárójelek számát. Ebben a fázisban, nem derül ki az elgépelés, de fordítás után egyértelműen fény derül rá.

VB-ben nehéz „kész akarva” generálni ilyen fajta hibákat, ugyanis a fejlesztőkörnyezet kódkiegészítő szolgáltatása nagyon aktívan vigyázza, hogy csak helyes azonosítókat használjunk, illetve minden kezdő programblokknak meg legyen a vége is. Azaz, ha azt írja a tanuló, hogy „`If feltétel`”, és leüti az entert, az „`Then`” és az „`End If`” automatikusan meg fog jelenni.

Amennyiben az „`i`” változónak nincs kezdőértéke, a fordító ugyancsak hibát fog jelezni. (L. 18. ábra)



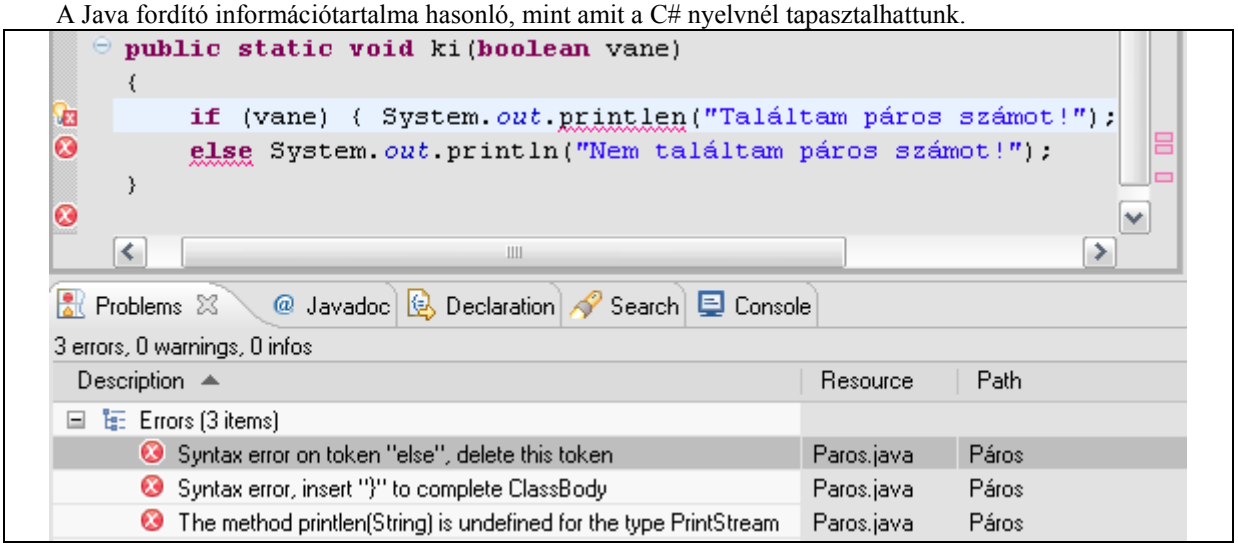
18. ábra. C# - a fordítás előtt már figyelmeztetést ad



19. ábra. Visual Basic: „Kierőszakolt” hiba, jó hibaüzenet

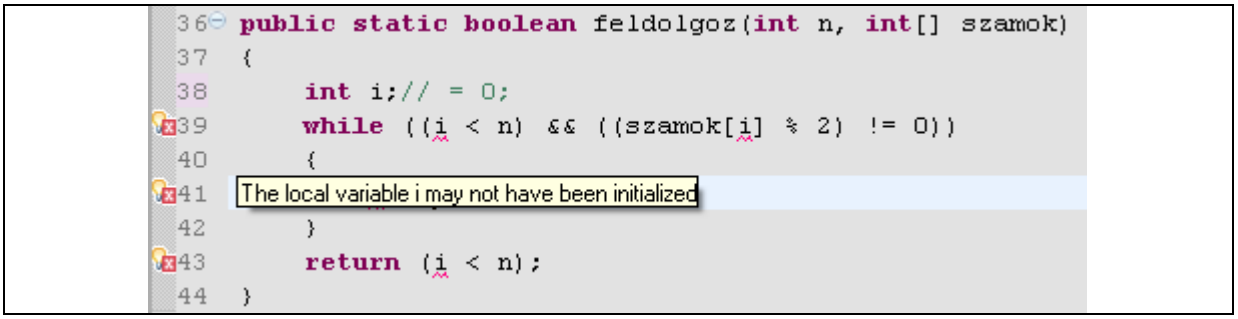
A fenti ábra egy nagyon valószínűtlen hibát mutat, viszont tisztán látszik, hogy fordítás nélkül, kék aláhúzással jelzi a környezet, illetve alul a hibalistában, hogy milyen szintaktikai hibát vétett a programozó.

A Java fordító információtartalma hasonló, mint amit a C# nyelvnél tapasztalhattunk.



20. ábra. Java – minden hibára fény derül

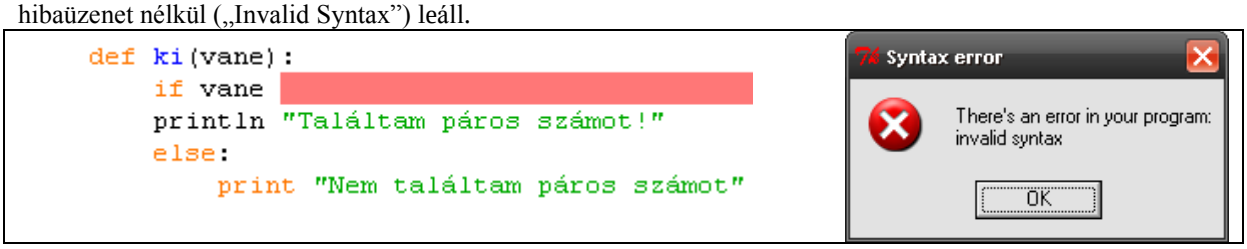
Fordítás előtt már aláhúzza a környezet a hibás szavakat, és a hibaüzenetek is rávilágítanak a valós problémára. Jó megoldás, hogy a hibás sorokat egy piros x-szel jelöli a rendszer. A gépelési hibákat könnyen lehet javítani a fejlesztőkörnyezet által felkínált „Quick fix” szolgáltatással. A környezet figyel a változók helyes definiálására (lásd 21. ábra), így a szemantikus hibák egy részére is figyelmeztet.



21. ábra. Java – Inicializálás hiányára fordítás előtt figyelmeztet

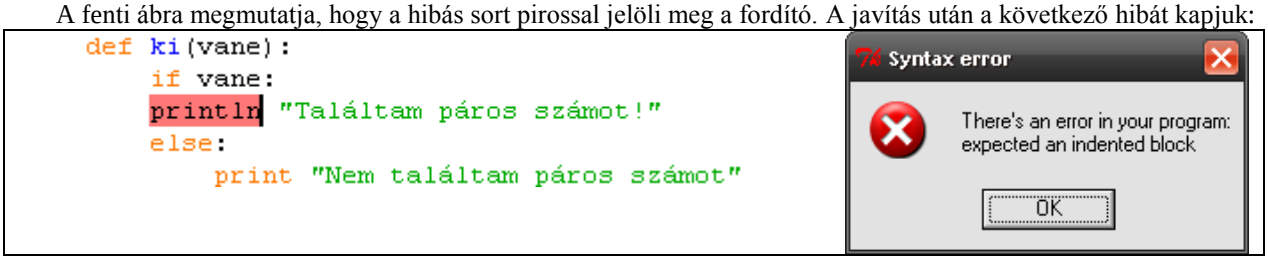
A Python és a Ruby abban különböznek az eddig tárgyalt nyelvektől, hogy ezek a nyelvek scriptnyelvek, azaz futási időben értelmezi őket a fordító.

A Python értelmezője, mielőtt lefuttatná a programot, ellenőrzi a kódot. Ha hibát talál, gyakorlatilag specifikus hibaüzenet nélkül („Invalid Syntax”) leáll.



22. ábra. Python: pontatlan, túl általános hibaüzenet

A fenti ábra megmutatja, hogy a hibás sort pirossal jelöli meg a fordító. A javítás után a következő hibát kapjuk:



23. ábra. Python: pontos hibaüzenet

Ez a hibaüzenet, az előzővel ellentétben, jól és pontosan határozza meg a problémát.

```
def ki(vane):
    if vane:
        println "Találtam páros számot!"
    else:
        print "Nem találtam páros számot"
```

24. ábra. Python: hol van a hiba?

A fenti ábrán látható állapotra is „érvénytelen szintakszis” üzenetet ad a fordító, ám a hiba helyét nem helyesen lokalizálja, és a hibaüzenetből semmilyen egyéb más információ nem derül ki, amely közelebb vinne a megoldáshoz.

A Ruby hibaüzeneteit hasonlóan nehéz – sőt, talán nehezebb értelmezni.

```
21  def ki(vane)
22    if (vane) then begin pputs 'Talaltam paros szamot'
23     else puts 'Nem talaltam paros szamot'
24    end
25  end
26  #Főprogram
27    szamok = Array.new
28    n = be(szamok)
29    vane = feldolgoz(n, szamok)
30    ki(vane)
paros.rb:24: warning: else without rescue is useless
paros.rb:30: syntax error, unexpected $end, expecting kEND
  ki(vane)
            ^
```

25. ábra. Ruby: Értelmetlen hibaüzenet

A két hibaüzenetből kiderül annyi, hogy a begin-end párokkal lehet valamilyen hiba, de arról egy utalás sincsen, hogy mit jelent a „$end” és a „kEND”. A 22. sorban látható gépelési hibára („`puts`” helyett „`pputs`”), futási időben fog fény derülni, és csak akkor, ha a „ki” eljárás bemeneti paramétere igaz.

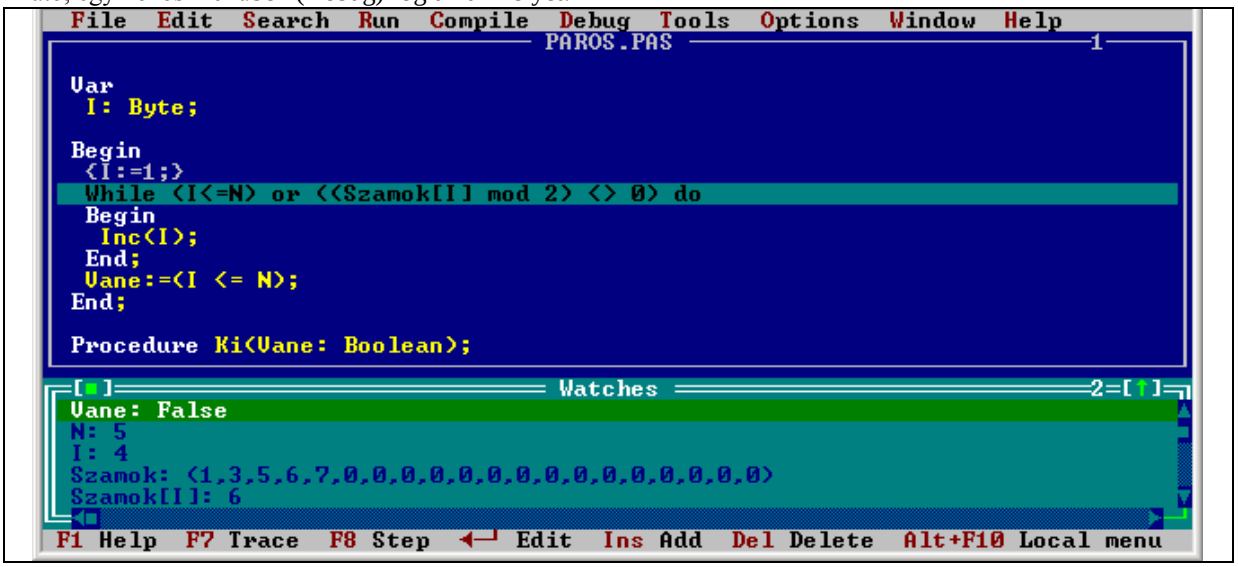
A nyelv tulajdonsága miatt (scriptnyelv), a szintaktikai és szemantikai hibák is futási időben derülnek ki.

2.3. Tesztelés, szemantikai hibák javítása

A dinamikus szemantikai hibák sokkal kényelmetlenebbek, mint a szintaktikusak vagy a statikus szemantikusak, ugyanis a program fut, csak hibásan, nem azt csinálja, amit kellene. Hosszabb időbe telik megtalálni és kijavítani őket. Ezek felderítésében segíti a programozót a nyomkövető (debug-) rendszer.

Mit az elvárás egy nyomkövető rendszerrel szemben? Mikor mondható jónak és használhatónak? Előnyös, ha a nyomkövető funkciók egy helyen, azaz egy menüben foglalnak helyet, jó, ha a megjelenítés lehetővé teszi, hogy egyszerre legyen látható és vizsgálható a kód, a változók tartalma, a program kimenete, illetve lehetőség legyen töréspontok létrehozására és soronkénti vagy eljárásonkénti végrehajtásra.

A BP egyszerűen kezelhető felületet bocsát a programozó rendelkezésére (lásd 26. ábra). Külön ablakban láthatóak a megfigyelésre felvett változók értékei, s az éppen szükséges funkciók (az ábra alján). Töréspontokat lehet elhelyezni a kódban, hogy csak a hibás részt futassuk soronként. A nyomkövetés összes kelléke könnyen megtalálható, egy közös menüben (Debug) foglalnak helyet.



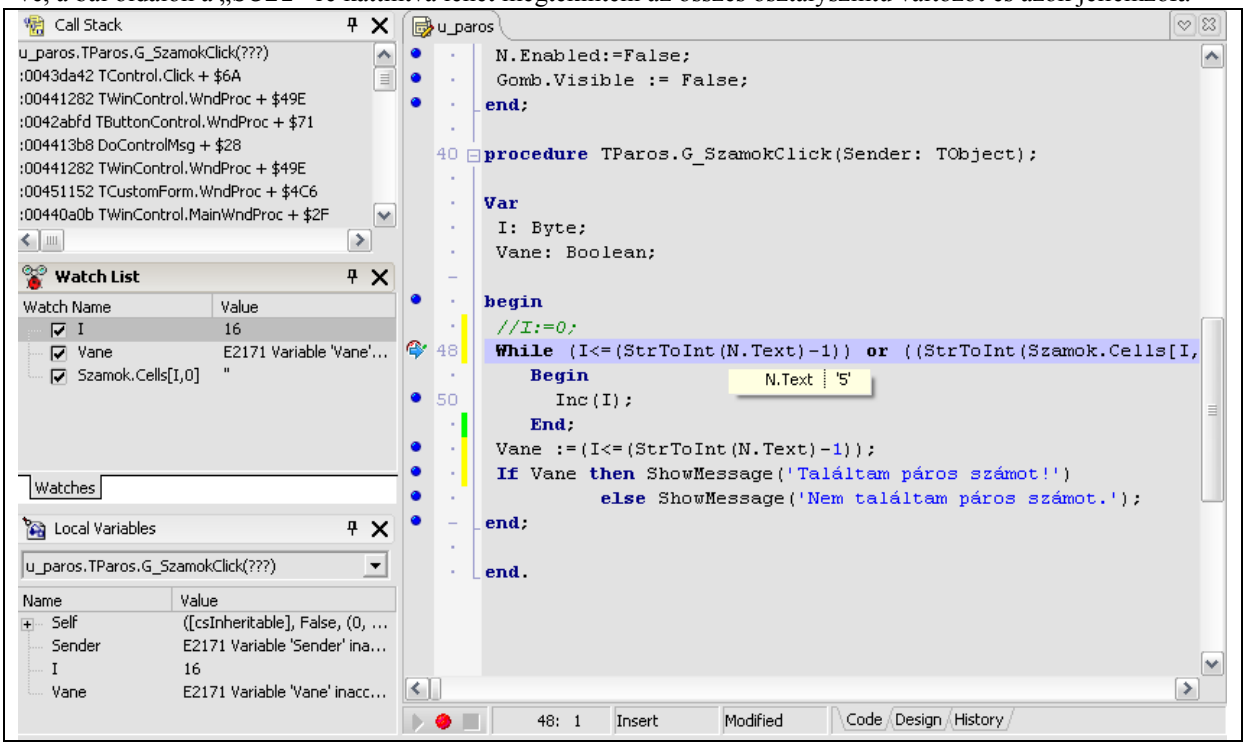
26. ábra. Nyomkövetés BP-ban

Ezekkel az eszközökkel könnyen detektálni lehet a kódolás közben elkövetett hibákat. Jelen példában a változók kezdőértékének hiányát és a hibás ciklusfeltételt.

A VS és az Eclipse egyik hasznos közös tulajdonsága – kivéve a két scriptnyelvet és a Dev C++-t –, hogy egy időben és helyen elég sok információt tudnak megmutatni. Nagyon hasznos, hogy az összes helyi változó értéke automatikusan látható, illetve, a kódban, mikor a programozó soronként hajtja végre a programot, a változó fölé helyezve a kurzort megtudhatja annak értékét, így a kód vizsgálata közben tisztában lesz a program belső állapotával. Ez sokkal kényelmesebbé teszi a hibakeresést, ugyanis valóban, minden egy helyen van.

A 27. ábra egy igényes megoldást mutat. A bal oldalon láthatóak a programozó által figyelt változók és az automatikusan megmutatottak is. A kép közepén látszik, ahogyan az egérkurzor „felfedi” az egyik komponens értékét. A nyomkövető rendszer magától értetődően működik, könnyű a kezelése.

A hibakeresésen kívül a vizuális fejlesztőkörnyezetek nyomkövető rendszere alkalmas arra, hogy a hibakeresésen túl segítsen bemutatni és megérteni az aktuális objektum szerkezetét és belső működését. Az alábbi ábrát tekintve, a bal oldalon a „`Self`”-re kattintva lehet megtekinteni az összes osztályszintű változót és azok jellemzőit.

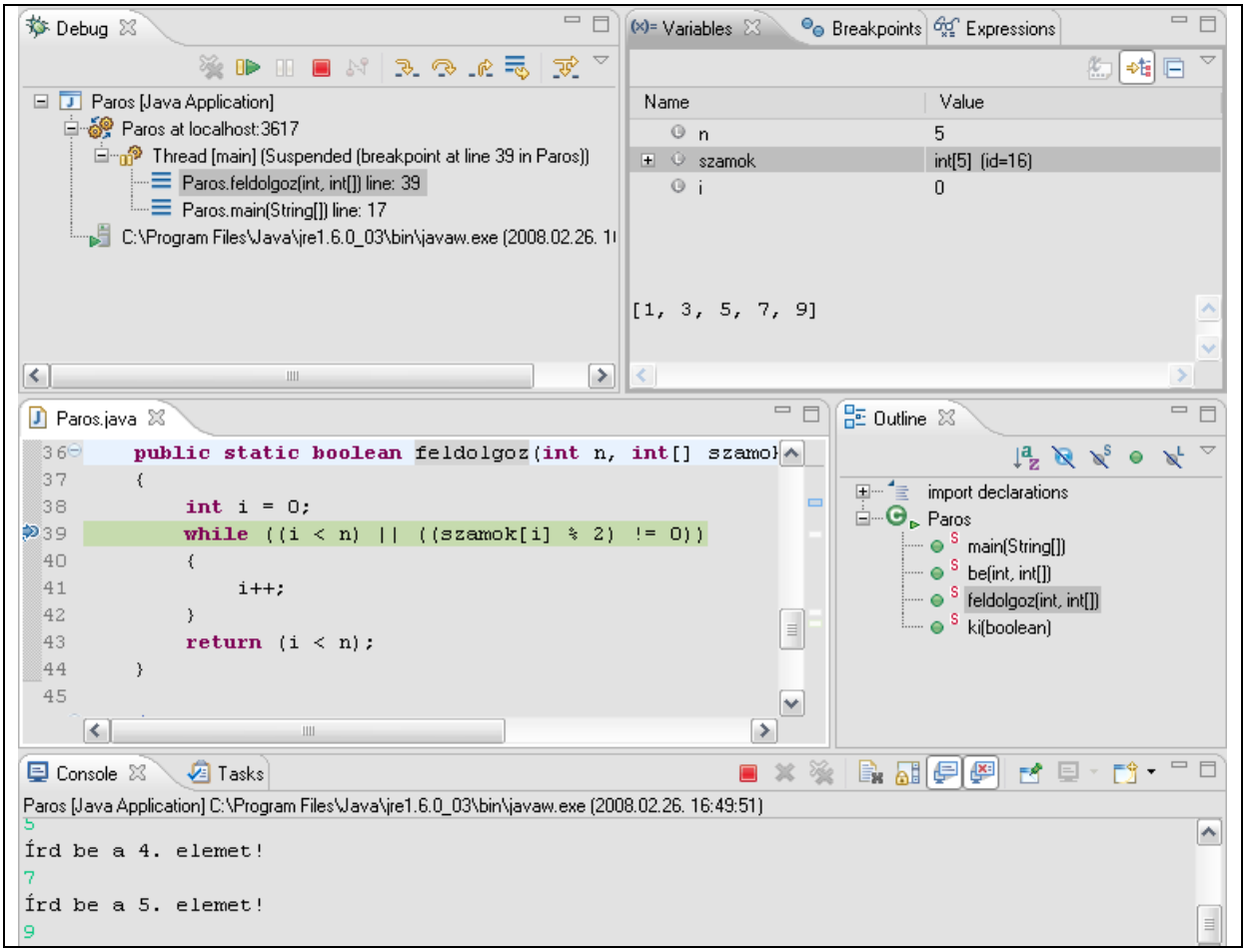


27. ábra. A Delphi nyomkövető rendszere

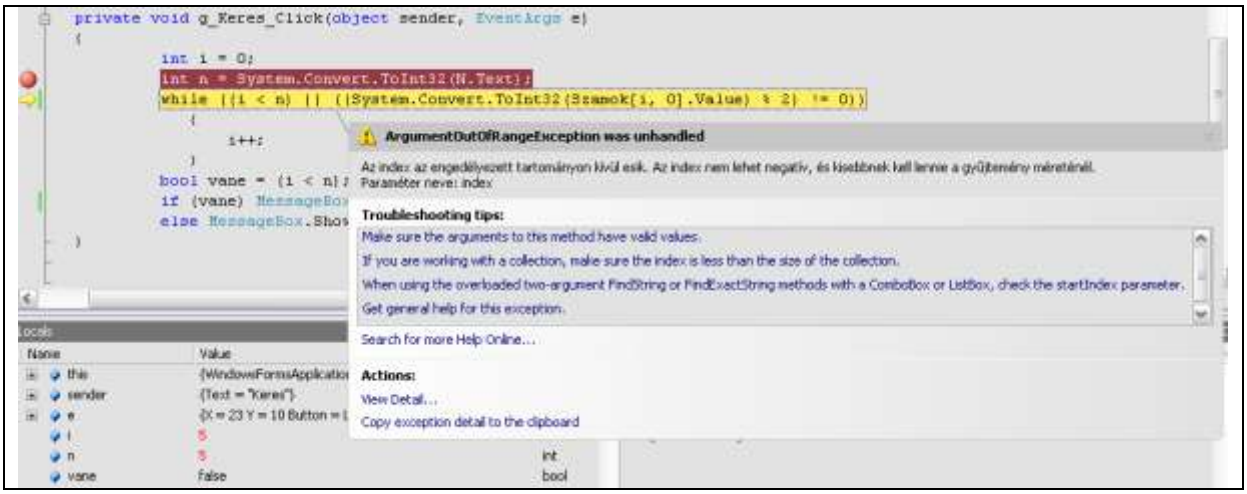
Az Eclipse debuggere (28. ábra) az elrendezés szempontjából a legjobb. Valóban megmutat minden információt, még a program kimenete is kényelmesen elfér a képernyőn. A nyomkövető rendszernek logikai kifejezéseket is meg lehet adni, így külön lehetséges figyelni a ciklusfeltételt, illetve azt, hogy milyen értéket fog kapni a „feldolgoz” függvény. Utóbbiakat az „Expressions” fülre kattintva lehet megtekinteni.

A Visual C# a színek kezelésére és a nyomlövető rendszer üzeneteiben jó példa. A 29. ábrán jól elkülönül a töréspont (piros színnel) és az aktuális (jelen esetben a hibás) sortól (sárga színnel). A hibaüzenet teljes körű, nem csak a hibát közli, hanem megoldási javaslatokkal is szolgál.

A Dev C++-t és az Eclipse-t külön hibakereső módban kell fordítani és futtatni, hogy nyomon tudjuk követni a program belső életét. Előbbi hátránya az utóbbival szemben, hogy a beállításokon kell változtatni ehhez, míg az Eclipse-ben egy egyszerű kattintásra nyomkövető módba kerül a fejlesztői környezet.



28. ábra. Java – minden információ kéznél van



29. ábra. A C# nyomkövető rendszere – információdús hibaüzenet

Vizsgálatunkban szereplő két scriptnyelv hátránya – amely a nyelv alaptulajdonságából fakad –, hogy nem rendelkeznek nyomkövető rendszerrel. Ha szemantikus hibát találunk, a programírónak kell kiíratnia a fontos változók értékeit (lásd 30. ábra). Ez a megoldás jóval munka- és időigényesebb, és egy bonyolultabb problémát nehezebben lehet felderíteni.

```
13  def feldolgoz(n, szamok)
14    i = 1
15    #Nyomkövetés
16    puts 'n: ' + n.to_s
17    puts 'Belepunk a ciklusba...'
18    while (i <= n) or ((szamok[i] % 2) != 0)
19      puts 'i: ' + i.to_s
20      puts 'szamok[i]: ' + szamok[i].to_s
21      puts '(i <= n): ' + (i <= n).to_s
22      #Nyomkövetés vége
23      i = i + 1
24    end
25    #Nyomkövetés
26    puts 'Kileptunk a ciklusbol'
27    puts 'i: ' + i.to_s
28    puts '(i <= n): ' + (i <= n).to_s
29    #Nyomkövetés vége
30    return (i <= n)
31  end
```

```
n: 5
Belepunk a ciklusba...
i: 1
szamok[i]: 1
(i <= n): true
i: 2
szamok[i]: 3
(i <= n): true
i: 3
szamok[i]: 5
(i <= n): true
i: 4
szamok[i]: 7
(i <= n): true
i: 5
szamok[i]: 9
(i <= n): true
paros.rb:18:in `feldolgoz': undefined method `%' for nil:NilClass (NoMethodError
)
        from paros.rb:41
```

30. ábra. A Ruby „debuggere”

3. Összefoglalás

A fenti összehasonlításból kitűnik, hogy nincsen tökéletes nyelv és környezet, mindegyikük rendelkezik előnyökkel és hátrányokkal.

Oktatási szempontból előnyös, ha a választott nyelv könnyen megjegyezhető alapszavakból áll és egyszerű programszerkezetet használ. Fontos megvizsgálni, hogy milyen könnyű megírni az első *értelmes* programot, tehát az első olyan programot, amelynek gyakorlati haszna is van. Egy olyan nyelvre van szükség, amely közel áll az algoritmikus nyelvünkhöz, a tanuló megtanulja és megérti, a programozás alapvető elemeinek helyét, szerepét és használatát. A Pascal erős (és szigorú) típusossága, átlátható és memorizálható programszerkezete „rászoktatja” a tanulót, hogy ne felejtsen el változót deklarálni, típust definiálni, hogy programírás közben mindig gondolja végig, miket és hogyan akar használni az éppen aktuális eljárás megvalósításakor. A többi tárgyalt nyelv ad hoc módon elhelyezhető változódeklarációja nem támogatja ezt a gondolkodásmódot. A Pascal jó alap a továbblépéshez az objektum orientált programozás és/vagy egy 4GL fejlesztőrendszer felé. Utóbbi, vizualitása miatt, segítheti az objektum orientált paradigma megértését.

A C++, C#, Java nyelvek objektum orientáltsága új távlatokat nyit nyelvi szinten, a Pascalhoz képest. A C++ nyelv túl bonyolult első imperatív nyelvként, és továbblépés esetén is jobb választásnak bizonyul nála a C# és a Java, kiforrottságuk és letisztultságuk miatt.

A VB rengeteg előnyös tulajdonsággal rendelkezik kezdő programozók számára: olvasható, jól strukturált kódot ad, viszont a nyelvi gyökerek és az erős kötődés a fejlesztői környezethez megnehezíti a továbblépést.

A scriptnyelvek gyenge típusossága nem támogatja a haladó programozási stílus kialakulását, ráadásul rontja a program olvashatóságát, mivel több idő szükséges a kód megértéséhez, mint az erősen típusos nyelveknél. Ezeken felül rengeteg hibalehetőséget rejt magában. Első nyelvként tanítani scriptnyelvet ellenjavallt. Középiskolában való tanításának oktatási haszna is kérdéses.

A kód olvashatóságát segítő elemek szempontjából nem voltak lényeges különbségek az előbbiekben megismert környezetek között. Megjegyzendő, hogy a kulcsszavak kiemelésén túl, az intellisence szolgáltatással bíró környezetek jóval több segítséget nyújtanak a programozó számára.

A fejlesztő környezetek nyomkövető szolgáltatásai között – leszámítva a két scriptnyelv környezetét és a Dev C++-t – nem voltak jelentős különbségek. Hibaüzenetek terén a C++, a Python és a Ruby fordítója vizsgázott a leggyengébben, a nem specifikus, lényeget nélkülöző és néha érthetetlen hibaüzenetek miatt. A Delphi, a Visual Basic és C#, illetve az Eclipse kifejezetten segítik a hibajavítás menetét hibaüzeneteikkel.

Többször felmerült kérdésként, hogy érdemes-e objektum orientált nyelvet/környezetet használni? Az objektumság fogalmát be lehet vezetni egyszerű módon, viszont ez esetben az algoritmikus nyelvünket is alkalmasá kell tenni az algoritmikus gondolatok e paradigmába illő módon történő kifejtésére. További vizsgálatok szükségesek annak megválaszolására, hogy ez hogyan hat a programozás-oktatás hatékonyságára?

Egyszerűbbnek és a tanulók számára érthetőbbnek látszik az a módszer, hogy az oktatás első szakaszában, a programozási tételek tanítása idején az objektum orientáltság megismertetése nélkül, csakis az algoritmus szempontjából fontos nyelvi elemek használatára koncentráljunk.

Ebben a tanítási szakaszban nem hasznos, ha 4GL rendszert vagy egy scriptnyelvet alkalmazunk a programozási feladatok megoldására, igaz: más-más ok miatt. A C++ nyelv túl bonyolult első imperatív nyelvként, így a Borland Pascal környezete látszik jónak a kezdeti lépések megértéséhez, megtanításához.

A későbbiekben érdemes 4GL rendszerre váltani, megismertetni az objektum orientált programozást, illetve – akár – egy új nyelvet alkalmazni bonyolultabb feladatok megoldására. Új nyelvként a C nyelvcsalád valamelyik letisztultabb változatát, a C#-ot vagy a Javat javaslom, a fentebb megismert előnyös tulajdonságaik miatt.

Irodalomjegyzék

1. R. J. Sternberg, T. Ben-Zeev: *A matematikai gondolkodás természete.* Budapest: Vincze Kiadó. (1998.)
2. Szlávi Péter, Zsakó László: *Programozás tanítási módszerek.* http://digo.inf.elte.hu/~szlavi/ProgModsz/SzlaviZsako.pdf
3. Szlávi Péter: *Szoftverek értékelése iskolai szempontok szerint* http://digo.inf.elte.hu/~szlavi/InfoOkt/SzoftErt/IndSzoftErt.html
4. Kathleen Jensen and Niklaus Wirth: *PASCAL – User Manual and Report* http://www.cs.inf.ethz.ch/~wirth/books/Pascal/
5. Törley Gábor: *Középiskolai programozás oktatás vizuális környezetben* (Szakdolgozat). Budapest: ELTE-IK, Informatikai Szakmódszertani Csoport. (2005.)
6. Szlávi Péter: *Programozási tételek szerepe a gyakorlati programozásban.* Budapest: ELTE TTK Informatika Szakmódszertani Csoport. (1996.).
7. B. Stroustrup: *The C++ Programming Language* (3rd Edition). Addison-Wesley Longman. Reading Mass. USA. 1997. ISBN 0-201-88954-4.
8. *Visual C++ Developer Center* – http://msdn2.microsoft.com/hu-hu/visualc/default(en-us).aspx
9. *Visual C# Developer Center* – http://msdn2.microsoft.com/hu-hu/visualc/default(en-us).aspx
10. *Java™ 2 Platform Standard Edition 5.0 API Specification* – http://java.sun.com/j2se/1.5.0/docs/api
11. *Visual Basic Developer Center* – http://msdn2.microsoft.com/hu-hu/vcsharp/default(en-us).aspx
12. Nagy Gusztáv: *Programozási nyelvek az oktatásban* Informatika a felsőoktatásban, Programozási nyelvek az oktatásban 2005. (Konferencia kiadvány) – http://agrinf.agr.unideb.hu/if2005/kiadvany/papers/E73.pdf

13. *Programming Ruby – The Pragmatic Programmer's* Guide – http://www.ruby-doc.org/docs/ProgrammingRuby
14. Guido van Rossum: *Python Tutorial* – http://docs.python.org/tut/tut.html
15. Bölecz Mónika: *Felhasználói felületek tervezése.* http://www.szt.vein.hu/~bolecz/szf/felhasznaloiFeluletek.doc
16. E. Horowitz: *Magasszintű programnyelvek*, Műszaki Könyvkiadó, 1987.
17. N. Wirth: *Algoritmusok + adatszerkezetek = programok*, Műszaki Könyvkiadó, 1982.