

Univerzális programozás

Írd meg a saját programozás tankönyvedet!

Ed. BHAX, DEBRECEN,
2019. február 19, v. 0.0.4

Copyright © 2019 Dr. Bátfai Norbert

Copyright (C) 2019, Norbert Bátfai Ph.D., batfai.norbert@inf.unideb.hu, nbatfai@gmail.com,

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

<https://www.gnu.org/licenses/fdl.html>

Engedélyt adunk Önnek a jelen dokumentum sokszorosítására, terjesztésére és/vagy módosítására a Free Software Foundation által kiadott GNU FDL 1.3-as, vagy bármely azt követő verziójának feltételei alapján. Nincs Nem Változtatható szakasz, nincs Címlapszöveg, nincs Hátlapszöveg.

<http://gnu.hu/fdl.html>

COLLABORATORS

	<i>TITLE :</i> Univerzális programozás		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Bátfai, Norbert Ács Halász, Dávid	2019. október 1.	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.0.1	2019-02-12	Az iniciális dokumentum szerkezetének kialakítása.	nbatfai
0.0.2	2019-02-14	Inciális feladatlisták összeállítása.	nbatfai
0.0.3	2019-02-16	Feladatlisták folytatása. Feltöltés a BHAX csatorna https://gitlab.com/nbatfai/bhax repójába.	nbatfai
0.0.4	2019-02-19	Aktualizálás, javítások.	nbatfai

Ajánlás

„To me, you understand something only if you can program it. (You, not someone else!) Otherwise you don't really understand it, you only think you understand it.”

—Gregory Chaitin, *META MATH! The Quest for Omega*, [METAMATH]

Tartalomjegyzék

I. Bevezetés	1
1. Vízió	2
1.1. Mi a programozás?	2
1.2. Milyen doksikat olvassak el?	2
1.3. Milyen filmeket nézzek meg?	2
II. Tematikus feladatok	3
2. Helló, Turing!	5
2.1. Végtelen ciklus	5
2.2. Lefagyott, nem fagyott, akkor most mi van?	6
2.3. Változók értékének felcserélése	8
2.4. Labdapattogás	9
2.5. Szóhossz és a Linus Torvalds féle BogomIPS	11
2.6. Helló, Google!	13
2.7. 100 éves a Brun tétel	16
2.8. A Monty Hall probléma	17
3. Helló, Chomsky!	20
3.1. Decimálisból unárisba átváltó Turing gép	20
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	21
3.3. Hivatkozási nyelv	22
3.4. Saját lexikális elemző	23
3.5. l33t.1	24
3.6. A források olvasása	26
3.7. Logikus	27
3.8. Deklaráció	28

4. Helló, Caesar!	30
4.1. int *** háromszögmátrix	30
4.2. C EXOR titkosító	31
4.3. Java EXOR titkosító	34
4.4. C EXOR törő	35
4.5. Neurális OR, AND és EXOR kapu	38
4.6. Hiba-visszaterjesztéses perceptron	42
5. Helló, Mandelbrot!	44
5.1. A Mandelbrot halmaz	44
5.2. A Mandelbrot halmaz a <code>std::complex</code> osztállyal	47
5.3. Biomorfok	50
5.4. A Mandelbrot halmaz CUDA megvalósítása	52
5.5. Mandelbrot nagyító és utazó C++ nyelven	56
5.6. Mandelbrot nagyító és utazó Java nyelven	58
6. Helló, Welch!	61
6.1. Első osztályom	61
6.2. LZW	65
6.3. Fabejárás	70
6.4. Tag a gyökér	72
6.5. Mutató a gyökér	80
6.6. Mozgató szemantika	81
7. Helló, Conway!	83
7.1. Hangyaszimulációk	83
7.2. Java életjáték	89
7.3. Qt C++ életjáték	94
7.4. BrainB Benchmark	101
8. Helló, Schwarzenegger!	106
8.1. Szoftmax Py MNIST	106
8.2. Mély MNIST	110
8.3. Minecraft-MALMÖ	111

9. Helló, Chaitin!	115
9.1. Iteratív és rekurzív faktoriális Lisp-ben	115
9.2. Gimp Scheme Script-fu: króm effekt	116
9.3. Gimp Scheme Script-fu: név mandala	120
10. Helló, Gutenberg!	125
10.1. Programozási alapfogalmak	125
10.2. Programozás bevezetés	129
10.3. Programozás	131
III. Második felvonás	136
11. Helló, Arroway!	138
11.1. OO szemlélet	138
11.2. Homokózó	140
11.3. „Gagyí”	150
11.4. Yoda	152
11.5. Kódolás from scratch	152
12. Helló, Liskov!	155
12.1. Liskov helyettesítés sértése	155
12.2. Szülő-gyerek	159
12.3. Anti OO	161
12.4. Hello, Android!	161
12.5. Ciklomatikus komplexitás	162
13. Helló, Mandelbrot!	166
13.1.	166
14. Helló, Berners-Lee!	167
14.1. Nyékyné Gaizler Judit: Java 2 Útikalauz programozóknak 5.0	167
14.2. Forstner Bertalan, Ekler Péter, Kelényi Imre: Bevezetés a mobilprogramozásba. Gyors prototípus fejlesztés Python és Java nyelven	168

IV. Irodalomjegyzék	170
14.3. Általános	171
14.4. C	171
14.5. C++	171
14.6. Lisp	171
14.7. Bio Intelligence	171
14.8. Neurális hálózatok	171
14.9. Programozás Technika	171
14.10A Szoftver mérése	172

Ábrák jegyzéke

2.1. EXOR működése	9
2.2. Monty Hall probléma	17
4.1. C Exor titkosítás	33
4.2. C Exor törés	38
4.3. Neurális OR eredménye	39
4.4. Neurális OR, AND eredménye	40
4.5. Neurális EXOR eredménye	42
5.1. A mandelpngt.c++ elindítása	46
5.2. A mandel.png eredménye	47
5.3. A 3.1.2.cpp rogram elindítása	49
5.4. A 3.1.2.cpp eredménye	50
5.5. Biomorf képe	52
5.6. Makefile létrehozása	56
5.7. Mandelbrot halmaz alaphelyzetben	57
5.8. Mandelbrot halmaz nagyítva	58
6.1. A polargenteszt eredménye	63
6.2. A JAVA JDK kódcsipete	65
6.3. Az LZW fa C-ben	70
6.4. Fabejárás	70
6.5. Az LZW preorder bejárása	71
6.6. Az LZW postorder bejárása	72
6.7. Mutató a gyökér	81
7.1. A sejtek alakulása	89
7.2. A QT c++ program elindulása	101

7.3. BrainB Benchmark	104
8.1. Hálózat tesztelése	110
8.2. SMNIST Eredmény	111
8.3. Minecraft	114
9.1. Króm effekt	116
9.2. Mandala	124
11.1. A polárgenerátor eredménye	139
11.2. Java.random	140
11.3. Java Binfa	147
11.4. LZWJavaServlet.class megjelenítése	150
11.5. -128-as érték	151
11.6. -129-as érték	151
12.1. Az LSP megsértése	156
12.2. Az LSP betartása	158
12.3. A School.java program fordítása	160
12.4. Program futtatása	164
12.5. Ciklusos komplexitás eredménye	165

Táblázatok jegyzéke

12.1. BBP algoritmus eredménye	161
12.2. Ciklomatikus komplexitás	162

Előszó

Amikor programozónak terveztem állni, ellenezték a környezetemben, mondván, hogy kell szövegszerkesztő meg táblázatkezelő, de az már van... nem lesz programozói munka.

Tévedtek. Hogy egy generáció múlva kell-e még tömegesen hús-vér programozó vagy olcsóbb lesz alkálni igény szerint pár robot programozót a felhőből? A programozók dolgozók lesznek vagy papok? Ki tudhatná ma.

Mindenesetre a programozás a teoretikus kultúra csúcsa. A GNU mozgalomban látom annak garanciáját, hogy ebben a szellemi kalandban a gyerekeim is részt vehessenek majd. Ezért programozunk.

Hogyan forgasd

A könyv célja egy stabil programozási szemlélet kialakítása az olvasóban. Módszere, hogy hetekre bontva ad egy tematikus feladatcsokrot. Minden feladathoz megadja a megoldás forráskódját és forrásokat feldolgozó videókat. Az olvasó feladata, hogy ezek tanulmányozása után maga adja meg a feladat megoldásának lényegi magyarázatát, avagy írja meg a könyvet.

Miért univerzális? Mert az olvasótól (kvázi az írótól) függ, hogy kinek szól a könyv. Alapértelmezésben gyerekeknek, mert velük készítem az iniciális változatot. Ám tervezem felhasználását az egyetemi programozás oktatásban is. Ahogy szélesedni tudna a felhasználók köre, akkor lehetne kiadása különböző korosztályú gyerekeknek, családoknak, szakköröknek, programozás kurzusoknak, felnőtt és továbbképzési műhelyeknek és sorolhatnánk...

Milyen nyelven nyomjuk?

C (mutatók), C++ (másoló és mozgató szemantika) és Java (lebutított C++) nyelvekből kell egy jó alap, ezt kell kiegészíteni pár R (vektoros szemlélet), Python (gépi tanulás bevezető), Lisp és Prolog (hogyan lássuk mást is) példával.

Hogyan nyomjuk?

Ránts le a <https://gitlab.com/nbatfai/bhax> git repót, vagy méginkább forkolj belőle magadnak egy sajátot a GitLabon, ha már saját könyvön dolgozol!

Ha megvannak a könyv DocBook XML forrásai, akkor az alább látható **make** parancs ellenőrzi, hogy „jól formázottak” és „érvényesek-e” ezek az XML források, majd elkészíti a dlatex programmal a könyved pdf változatát, íme:

```
batfai@entropy:~$ cd glrepos/bhax/thematic_tutorials/bhax_textbook/
batfai@entropy:~/glrepos/bhax/thematic_tutorials/bhax_textbook$ make
rm -f bhax-textbook-fdl.pdf
xmllint --xinclude bhax-textbook-fdl.xml --output output.xml
xmllint --relaxng http://docbook.org/xml/5.0/rng/docbookxi.rng output.xml  ←
--noout
output.xml validates
rm -f output.xml
dlatex bhax-textbook-fdl.xml -p bhax-textbook.xls
Build the book set list...
Build the listings...
XSLT stylesheets DocBook - LaTeX 2e (0.3.10)
=====
Stripping NS from DocBook 5/NG document.
Processing stripped document.
Image 'dlatex' not found
Build bhax-textbook-fdl.pdf
'bhax-textbook-fdl.pdf' successfully built
```

Ha minden igaz, akkor most éppen ezt a legenerált `bhax-textbook-fdl.pdf` fájlt olvasod.



A DocBook XML 5.1 új neked?

Ez esetben forgasd a <https://tdg.docbook.org/tdg/5.1/> könyvet, a végén találsz az informatikai szövegek jelölésére használható gazdag „API” elemenkénti bemutatását.

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

1.2. Milyen doksikat olvassak el?

- Olvasgasd a kézikönyv lapjait, kezd a **man man** parancs kiadásával. A C programozásban a 3-as szintű lapokat fogod nézegetni, például az első feladat kapcsán ezt a **man 3 sleep** lapot
- [[KERNIGHANRITCHIE](#)]
- [[BMECPP](#)]
- Az igazi kockák persze csemegéznek a C nyelvi szabvány [ISO/IEC 9899:2017](#) kódcsipeteiből is.

1.3. Milyen filmeket nézzek meg?

- 21 - Las Vegas ostroma, <https://www.imdb.com/title/tt0478087/>, benne a **Monty Hall probléma** bemutatása.

II. rész

Tematikus feladatok

**Bátf41 Haxor Stream**

A feladatokkal kapcsolatos élő adásokat sugároz a <https://www.twitch.tv/nbatfai> csatorna, melynek permanens archívuma a <https://www.youtube.com/c/nbatfai> csatornán található.

DRAFT

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

Írj olyan C végtelen ciklusokat, amelyek 0 illetve 100 százalékban dolgoztatnak egy magot és egy olyat, amely 100 százalékban minden magot!

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/tree/master/attention_raising/Source/vegtelen

Végtelen ciklusnak nevezzük azt az utasítást, ami örökké futna, ha nem állítanánk le a programot. A végtelen ciklus nem minden esetben jelent rossz dolgot, csak akkor, ha olyan feladat során keletkezik, amelynek végrehajtása nem igényel ilyen ciklusos algoritmust. Ilyen ciklus lehet például az alábbi kódrészlet:

```
Program vegtelen.c {  
  
    int main() {  
        int i = 0;  
        while (i<=0) {  
            i--;  
            printf("\n vegtelen ciklus");  
        }  
    }  
}
```

A program a while utáni zárójelben lévő kifejezést értékeli ki. Mivel a az „i” változó értéke egyenlő vagy kisebb a nullával ezért ez a feltétel igaz lesz és a program lefut. Ennek eredményeképpen az i változó értéke eggyel kisebb lesz és újból indul az egész. Mivel az eredmény sose lesz nullánál nagyobb, ezért a feltétel mindig igaz lesz, azaz végtelen ciklusba kerül. Vannak esetek, amikor szükség van végtelen ciklusra, ekkor érdemes olyat programkódot használni, amiről lehet látni, hogy ezt direkt így akartuk és nem pedig szoftverhiba. Ilyen ciklus az alábbi példa, amikor nem töltjük ki a for ciklus fejlécét:

```
Program vegtelen_for {  
  
    int main() {  
        for(;;);  
    }  
}
```

```
}
```

Ez a végtelen ciklus 1 magot használ 100%-on. Ha több magot szeretnénk dolgoztatni, akkor hozzá kell adni a `#pragma omp parallel-t`, tehát a végleges kódrészlet így fog kinézni:

```
Program vegtelen_pragma.c {  
  
    int main() {  
        #pragma omp parallel  
        for(;;);  
    }  
  
}
```

Ezzel az elhelyezett OpenMP alapú parallel régiók segítségével olyan függvényt hívunk elő, amellyel könnyen kihasználhatjuk vele egy számítógép processzorainak teljes számítási kapacitását. Fontos megjegyezni, hogy fordítás során hozzá kell adni `-fopenmp` kódrészletet is, ezért végül így fog kinézni: `gcc vegtelen_pragma.c -o vegtelen_pragma -fopenmp` Ha olyan programra van szükség, ahol csak 0%-ban dolgoztatnak meg egy magot, akkor a `sleep(1)` függvényt kell használnunk:

```
Program vegtelen_sleep.c {  
  
    int main() {  
        for(;;) {  
            sleep(1);  
        }  
    }  
  
}
```

A `sleep()` függvény hatására a program a zárójelben megadott értékű másodpercre megáll, alszik. Ha a zárójelben nincs megadva argumentum, akkor meghatározatlan időre alszik. Ezzel érhető el, hogy a mag ne legyen 100%-ban kihasználva.

2.2. Lefagyott, nem fagyott, akkor most mi van?

Mutasd meg, hogy nem lehet olyan programot írni, amely bármely más programról eldönti, hogy le fog-e fagyni vagy sem!

Megoldás videó:

Megoldás forrása: tegyük fel, hogy akkora haxorok vagyunk, hogy meg tudjuk írni a `Lefagy` függvényt, amely tetszőleges programról el tudja dönteni, hogy van-e benne végtelen ciklus:

```
Program T100  
{  
  
    boolean Lefagy(Program P)  
    {  
        if(P-ben van végtelen ciklus)
```

```
    return true;
  else
    return false;
}

main(Input Q)
{
  Lefagy(Q)
}
}
```

A program futtatása, például akár az előző v. c ilyen pszeudókódjára:

```
T100(t.c.pseudo)
true
```

akár önmagára

```
T100(T100)
false
```

ezt a kimenetet adja.

A T100-as programot felhasználva készítsük most el az alábbi T1000-set, amelyben a Lefagy-ra épülő Lefagy2 már nem tartalmaz feltételezett, csak konkrét kódot:

```
Program T1000
{

  boolean Lefagy(Program P)
  {
    if(P-ben van végtelen ciklus)
      return true;
    else
      return false;
  }

  boolean Lefagy2(Program P)
  {
    if(Lefagy(P))
      return true;
    else
      for(;;);
  }

  main(Input Q)
  {
    Lefagy2(Q)
  }

}
```

Mit for kiírni erre a T1000 (T1000) futtatásra?

- Ha T1000 lefagyó, akkor nem fog lefagyni, kiírja, hogy true
- Ha T1000 nem fagyó, akkor pedig le fog fagyni...

akkor most hogy fog működni? Sehogy, mert ilyen Lefagy függvényt, azaz a T100 program nem is létezik.

Tanulságok, tapasztalatok, magyarázat...

2.3. Változók értékének felcserélése

Írj olyan C programot, amely felcseréli két változó értékét, bármiféle logikai utasítás vagy kifejezés használata nélkül!

Megoldás videó: https://bhaxor.blog.hu/2018/08/28/10_begin_goto_20_avagy_elindulunk

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/csere.c

Két változó értékének felcserélése segédváltozó nélkül a legegyszerűbben XOR-ral oldható meg:

```
Program csere.c {  
  
    int main(){  
  
        int x = 3;  
        int y = 5;  
  
        x ^= y; // x = 6  
        y ^= x; // y = 3  
        x ^= y; //x = 5  
  
        printf("x:%d y:%d\n", x, y);  
  
        return 0;  
    }  
}
```

Az XOR lényege, hogy a számok bináris alakjainak különbségét vizsgálja meg. Nézzük meg sorról sorra a kódot:

3 bináris alakja: 0011

5 bináris alakja: 0101

$x \oplus y$ esetén x értéke 0110 lesz, mert a 2 szám binárisan összehasonlítva, ha az adott biten a két szám értéke megegyezik, akkor 0-át kapunk, ha különbözőek, akkor pedig 1-et. Az eredményt az alábbi táblázat mutatja, ahol az eltérést pirossal, az egyezést zölddel jelöltem:

	BINÁRIS ALAK			
3	0	0	1	1
5	0	1	0	1
Összehasonlítás	0	1	1	0

2.1. ábra. EXOR működése

Ezután $y \hat{=} x$ esetén (ahol $y = 0101$ és $x = 0110$) y értéke 0011 lesz, ami a 3-as szám bináris alakjának felel meg. Ezután már csak a x értékét kell helyesen megadni, azaz $x \hat{=} y$ esetén (ahol $x = 0110$ és $y = 0011$) x értéke binárisan 0101 lesz, ezzel pedig meg is kapjuk az 5-ös számot, azaz y és x értékét sikeresen felcseréltük segédváltozó nélkül.

2.4. Labdapattogás

Először **if-ekkel**, majd bármiféle logikai utasítás vagy kifejezés használata nélkül írj egy olyan programot, ami egy labdát pattogtat a karakteres konzolon! (Hogy mit értek pattogtatás alatt, alább láthatod a videón.)

Megoldás videó: <https://bhaxor.blog.hu/2018/08/28/labdapattogas>

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/tree/master/attention_raising/Source/labdapattogas

Először nézzük meg a labdapattogatást if-ekkel. Az `initscr()` curses módba kapcsolja a terminált. Ez törli a képernyőt és fekete háttérrel ad, továbbá tárolja az ablak adataival kapcsolódó információkat. Először fontos, hogy az ablakkal és a lépésekkel kapcsolatos változókat meghatározása. Az `int x`-el és az `int y`-al adjuk meg az aktuális pozíciót, mely az x és az y tengelyen fog elhelyezkedni. Az `int xnov` és az `int ynov` lesznek a lépésközök, azaz hogy mennyit lépjen majd előre a labda az x és y tengelyen. Az `int mx`-ben és az `int my`-ben pedig az ablak méreteit fogjuk majd eltárolni.

Program `labdaif.c`

```
{  
  
    WINDOW *ablak;  
    ablak = initscr ();  
  
    int x = 0;  
    int y = 0;  
  
    int xnov = 1;  
    int ynov = 1;  
  
    int mx;  
    int my;  
    ...  
}
```

Mivel azt szeretnénk, hogy a program folyamatosan, megállás nélkül fusson, ezért végtelen ciklusra lesz szükségünk, amit a `for(;;)` ciklussal érhetünk el. Ezen belül kell program számára megadni, hogy pontosan mekkora lesz a képernyő. Ezt a `getmaxyx()` függvény segítségével érhetjük el, ahol az ablak-ban tárolt értékeket elmenti az `mx` és `my` változóba.

Megadhatjuk továbbá, hogy a képernyő ciklusonként törölje az előző "labda" nyomát a `clear()` függvénnyel. Ha ezt kihagyjuk, akkor a képernyőn folyamatosan látni fogjuk a labda eddigi helyzetét is, azaz folyamatosan "csíkot húz" maga után. Ennek a függvénynek mindig az `mvprintw()` előtt kell szerepelnie, ellenkező esetben csak üres ablakot fogunk látni. Ez utóbbi függvény segítségével adhatjuk meg, hogy az "O" karaktert, hol jelenítse meg a képernyőn. Majd az a következő pár sorban az `x` és `y` értékeket 1-el növeljük.

```
Program labdaif.c
{
    ...
    getmaxyx ( ablak, my , mx );

    clear ();
    mvprintw ( y, x, "O" );

    x = x + xnov;
    y = y + ynov;
    ...
}
```

Ezután következik annak meghatározása, hogy a labda elérte-e a képernyő szélét. Ha igen, akkor az értékeket meg kell szorozni -1-el és így a labda "visszapattog". Itt kell használnunk az `if()` függvényeket. Ha itt megnézzük az első sort, akkor láthatjuk, hogy ha az `x` értéke nagyobb mint az ablak `mx-1` értéke, akkor a labda útját, azaz az `xnov` értékét meg kell szorozni mínusz 1-gyel. Értelemszerűen ugyanígy kell meghatározni a többit is a hozzá tartozó változókkal.

```
Program labdaif.c
{
    ...
    if ( x>=mx-1 ) { // elerte-e a jobb oldalt?
        xnov = xnov * -1;
    }
    if ( x<=0 ) { // elerte-e a bal oldalt?
        xnov = xnov * -1;
    }
    if ( y<=0 ) { // elerte-e a tetejet?
        ynov = ynov * -1;
    }
    if ( y>=my-1 ) { // elerte-e a aljat?
        ynov = ynov * -1;
    }
    ...
}
```

Végül az `usleep()` függvénnyel adjuk meg a labda sebességét. Jelenleg 1 másodperc van megadva, ami átszámolva 100000 mikroszekundum. Ha ez letelik, a ciklus újraindul.

Ha a **labdapattogást if nélkül** szeretnénk megoldani, akkor maradékos osztást kell használni, ami az alábbi kódrészletben látható:

```
Program labda.c
{
    ...

    for (;;)
    {
        getmaxyx (ablak, my, mx);
        xj = (xj - 1) % mx;
        xk = (xk + 1) % mx;

        yj = (yj - 1) % my;
        yk = (yk + 1) % my;

        clear ();

        mvprintw (abs (yj + (my - yk)),
                  abs (xj + (mx - xk)), "o");

        refresh ();
        usleep (100000);

    }

    ...
}
```

2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

Írj egy programot, ami megnézi, hogy hány bites a szó a gépeden, azaz mekkora az int mérete. Használd ugyanazt a while ciklus fejet, amit Linus Torvalds a BogoMIPS rutinjában!

Megoldás videó: https://youtu.be/9KnMqrkj_kU, <https://youtu.be/KRZlt1ZJ3qk>, .

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/tree/master/attention_raising/Source/bogomips

A szóhosszt az alábbi programmal tudjuk kiírni:

```
Program szohossz.c
{
    int h = 0;
    int n = 0x01;
    do
        ++h;
```



```
while (n <= 1);  
printf ("A szóhossz ezen a gépen: %d bites\n", h);  
return 0;  
  
}
```

A program elindítása után a terminálban olvashatjuk, hogy a szóhoz ezen a gépen 32 bites.

A `0x01` kifejezés hexadecimális (16-os számrendszer) számot jelent, mely `0x` prefixszel kezdődik. A `do` és `while` páros pedig azt jelenti, hogy "csináld ezt, amíg a feltétel teljesül". Van még egy fontos eleme a programnak, ami a `while` utáni zárójelben szerepel, ez pedig a bittologató (vagy más néven shiftelő) operátor. Ennek a jele: `<<` vagy `>>` lehet. Jelen példában a `<<` operátort használjuk, mert a biteket balra szeretnénk tolni egyesével, egészen addig, amíg csupa nullát nem kapunk. A bitek léptetését így kell elképzelni:

Bitek léptetése

```
00000001  
00000010  
00000100  
00001000  
00010000  
00100000  
01000000  
10000000  
00000000
```

A program végül azért ír 32 bitet végeredményül, mert ennyiszor fut le a ciklus, amíg csupa nulla nem lesz.

A **Bogomips** egy Linux operációs rendszeren nyújtott mérési program, amely relatív módon jelzi, hogy a számítógép processzor milyen gyorsan fut, azaz a processzor hányszor halad át egy adott cikluson egy meghatározott idő alatt. A programot Linus Torvalds írta. Most ennek a programnak a működését fogom bemutatni.

A program elején látható, hogy először 2 változót deklaráltunk: az egyik a `loops_per_sec`, melynek értéke 1, a másik pedig a `ticks`, ami a `while` cikluson belül a `clock()` függvényt rendeljük hozzá. Ez a `clock()` függvény méri, hogy eddig mennyi processzor idő telt el. A `while` ciklus zárójelében a `loops_per_sec` értéke a `<=` shiftelő operátor miatt 2 hatványaival fog számolni. Ezután meghívjuk a `delay()` függvényt, aminek mindig átadom a ciklus változó értékét. Nagyon gyorsan növekedik, mindig hatványodik és ez a `delay()` függvény azt csinálja, hogy 0-tól egyesével növelgetve elszámol a `loops` értékig. Ezt követően újra lekérem a `ticks` értékét, mégpedig úgy, hogy a processzoridőből kivonjuk az előző időt. Lényegében ezzel kapjuk meg, hogy mennyi idő telt el.

Program `bogomips.c`

```
void delay (unsigned long long loops)  
{  
    for (unsigned long long i = 0; i < loops; i++);  
}  
  
...
```

```
while (loops_per_sec <= 1)
{
    ticks = clock ();
    delay (loops_per_sec);
    ticks = clock () - ticks;

    ...
}
```

Az `if` függvénnyel megvizsgáljuk, hogy a `ticks` értéke nagyobb vagy egyenlő-e a `CLOCKS_PER_SEC` értékénél. Ez utóbbi értéke mindig 1.000.000. Ha ez a feltétel teljesül, akkor kiszámoljuk, `loops_per_sec` értékét. Itt arra vagyunk kíváncsiak, hogy milyen ciklusérték tartozna hozzá, ha nem 2 hatványaival mennénk. Tehát, hogy milyen hosszú ciklust képes végrehajtani a gép. Kiíratásnál még elosztjuk az adatokat, hogy jobban olvasható legyen a kapott eredmény.

Program `bogomips.c`

```
...
    if (ticks >= CLOCKS_PER_SEC)
    {
        loops_per_sec = (loops_per_sec / ticks) * CLOCKS_PER_SEC;

        printf ("ok - %llu.%02llu BogoMIPS\n", loops_per_sec / ↵
            500000,
            (loops_per_sec / 5000) % 100);

        return 0;
    }
}
...
```

2.6. Helló, Google!

Írj olyan C programot, amely egy 4 honlapból álló hálózatra kiszámolja a négy lap Page-Rank értékét!

Megoldás videó:

Megoldás forrása:

A PageRank a Google internetes keresőmotor alapja, amit a Google alapítói, Larry Page és Sergey Brin fejlesztettek ki 1998-ban a Stanford Egyetemen. A PageRank a Google bejegyzett védjegye, legfontosabb tulajdonsága, hogy képes elemezni az oldalak közötti kapcsolatokat és ezáltal relevánsabb találatokat tud visszaadni. A forrásban látható program egy 4 honlapból álló hálózatra számolja ki a négy lap PageRank-ét.

Először is szükségünk lesz egy 4x4-es mátrixra, mivel 4 honlapunk van. Ezt az `L` változóban deklaráljuk. Látható, hogy oszloponként vannak kiszámolva az eredmények. Minden oldalnak egy egységnyi szavazata van, amit szétoszt azok között az oldalak között, amikre hivatkozik. Tehát ha az `A` oszlopban csak 1

honlapra van hivatkozás, ezért oda 1-es kerül. A második oszlopban lévő honlap 2 oldalra is hivatkozik ezért ezt egyenlő arányban osztja szét, tehát mindkét hivatkozott oldal fél-fél szavazatot kap és ez így megy tovább a többi oszlopnál is. Ezután meghívjuk a `pagerank()` függvényt, aminek átadjuk az `L` változóban tárolt mátrixot.

Program `pagerank.c`

```
...

int main (void){
    double L[4][4] = {
        {0.0, 0.0, 1.0/3.0, 0.0},
        {1.0, 1.0/2.0, 1.0/3.0, 1.0},
        {0.0, 1.0/2.0, 0.0, 0.0},
        {0.0, 0.0, 1.0/3.0, 0.0}
    };

    pagerank(L);

    printf("\n");

    return 0;
}
```

A `pagerank()` függvényben a `PRv[]` tömbben lesznek eltárolva az `L` mátrixból kapott értékek. A `PR[]` tömbben pedig majd a számítás során kapott eredményeket. A `for` függvény négyszer fog lefutni, mert összesen ennyi honlapunk van, ezeket össze fogja adni, míg végül egy oszlopból álló 4 soros mátrixot kapunk, aminek eredménye bekerül a már említett `PR[]` tömbbe.

Program `pagerank.c`

```
...

void pagerank(double T[4][4]){
    double PR[4] = { 0.0, 0.0, 0.0, 0.0 };
    double PRv[4] = { 1.0/4.0, 1.0/4.0, 1.0/4.0, 1.0/4.0};

    int i, j;
    for(;;){
        for (i=0; i<4; i++){
            PR[i]=0.0;
            for (j=0; j<4; j++){
                PR[i] = PR[i] + T[i][j]*PRv[j];
            }
        }

        if (tavolsag(PR,PRv,4) < 0.0000000001)
            break;

        for (i=0; i<4; i++){
```

```
        PRv[i]=PR[i];
    }
}
kiir (PR, 4);
}

...
```

Ezután if függvénnyel megvizsgáljuk, hogy a távolság() függvény által adott eredmény kisebb-e 0.0000000001-val. Az ezt a függvényt az alábbi kódrészletben lehet látni:

Program pagerank.c

```
...

double
tavolsag (double PR[], double PRv[], int n){

    int i;
    double osszeg=0;

    for (i = 0; i < n; ++i)
        osszeg += (PRv[i] - PR[i]) * (PRv[i] - PR[i]);
    return sqrt(osszeg);
}

...
```

Ezután már csak kiíratjuk a PR[] tömbben tárolt eredményeket egyesével, amit a kiir() függvény segítségével tesszük meg.

Program pagerank.c

```
...

void
kiir (double tomb[], int db){
    int i;

    for (i=0; i<db; ++i){
        printf("%f\n",tomb[i]);
    }
}

...
```

2.7. 100 éves a Brun tétel

Írj R szimulációt a Brun tétel demonstrálására!

Megoldás videó: <https://youtu.be/xbYhp9G6VqQ>

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Primek_R/stp.r

A számelmélet egyik legfontosabb tétele a Brun-tétel, mely a prímszámokkal kapcsolatos elméleteket gondolja tovább. Prímszámoknak nevezzük azokat a számokat, amelyeknek trivális osztói vannak, azaz csak önmagával és 1-el oszthatók. A Brun-tétel szerint végtelen sok olyan p prím létezik, amire $p+2$ is prím (pl.: 3,5; 5,7; 7,9; stb). Az ilyen prímpárokat, amiknek a különbsége 2, ikerprímeknek nevezzük. Az ilyen ikerprímsejtések bizonyítása vagy cáfolata jelenleg meghaladja a matematika eszközeit. Az ikerprímek, ha végtelen sokan vannak is, nagyon ríktán fordulnak elő a prímek között. Ugyanis míg a prímek reciprokaiból álló sor divergens, addig az ikerprímek reciprokaik sora konvergens. Konvergensnek nevezzük azokat a sorokat, ha az elemek tartanak egy számhoz. Azt a számot pedig ahova eljut, a sor összegének nevezzük. A Brun tétel szerint ezt a sor összeget az ikerprímszámok reciprokaiknak összeadásával kapjuk meg $[(1/3+1/5)+(1/5+1/7)+(1/7+1/9)+...]$ és megkapunk egy határértéket. Ezt a határértéket nevezzük Brun konstansnak. Most nézzük meg az alábbi programot, ami megpróbálja közelíteni a Brun konstans értékét:

```
Program stp.r
library(matlab)

stp <- function(x){

  primes = primes(x)
  diff = primes[2:length(primes)]-primes[1:length(primes)-1]
  idx = which(diff==2)
  t1primes = primes[idx]
  t2primes = primes[idx]+2
  rt1plust2 = 1/t1primes+1/t2primes
  return(sum(rt1plust2))
}

x=seq(13, 1000000, by=10000)
y=sapply(x, FUN = stp)
plot(x,y,type="b")
```

A kód futtatásához Matlabra lesz szükség. A `primes()` függvény egy paramétert kér, az x helyére bármilyen számot írhatunk és az ott megadott számig fogja kiszámolni a prímeket.

A `diff` a `primes` vektorban lévő számok segítségével az egymást követő számok különbségét kapjuk meg egy vektorba rendezve.

Az `idx` változóban megkeressük azokat a számokat, ahol a `diff` egyenlő 2-vel. Ugyanis ezek lesznek majd az ikerprím párok.

A `t1primes` kiveszi az ikerprímpárok első tagját, majd a `t2primes` változóban az ikerprím pár első tagjához hozzáadunk 2-őt, és így megkapjuk az ikerprímpár második tagját is. Majd ezen ikerpárok reciprokösszegét a `rt1plust2` változóban tároljuk. Végül pedig a `sum()` függvénnyel ezeket a törteket összeadjuk.

Ezután már csak meg kell jelenítenünk a kapott eredményeket egy függvény segítségével. A jelen példában egy `seq()` függvényt fogjuk használni. Az `y` változóban az `sapply()` függvény használatával rendeljük hozzá az `x` érték adatait. Végül pedig a `plot()` függvénnyel rajzoljuk ki az eredményt.

2.8. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

Megoldás videó: https://bhaxor.blog.hu/2019/01/03/erdos_pal_mit_keresett_a_nagykonyvben_a_monty_hall-paradoxon_kapcsan

Megoldás forrása: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/MontyHall_R

A paradoxon egy az Egyesült Államokban az 1960-as években nagy sikerrel futott televíziós vetélkedő utolsó játékán alapszik. Nevét a show házigazdájáról kapta.

A probléma alaphelyzete a következő: A játékosnak mutatnak három zárt ajtót, melyek közül kettő mögött egy-egy kecske van, a harmadik pedig egy vadonatúj autót rejt. A hangulat fokozása érdekében a választást, illetve a felnyitást egy kicsit megbonyolították. A játékos kiválaszt egy ajtót, de mielőtt ezt kinyitná, a műsorvezető a másik két ajtó közül kinyit egyet, mely mögött biztos nem az autó van (természetesen a showman a kezdettől tisztában van azzal, hogy melyik ajtó rejt az autót). Ezt követően megkérdezi a már amúgy is ideges vendéget, hogy jól meggondolta-e a választását, vagyis nem akar-e váltani. A játékos dönt, hogy változtat, vagy sem, végül feltárul az így kiválasztott ajtó, mögötte a nyereménnyel. A paradoxon kérdése az, hogy érdemes-e változtatni, illetve van-e ennek egyáltalán jelentősége.

Vizsgáljuk meg a Monty Hall paradoxont: kezdetben tehát 1:3 az esélye, hogy bármelyik ajtó mögött ott lehet a főnyeremény, az autó. Ezután a műsorvezető kinyit egy olyan ajtót, ami mögött nincs ott az autó. Ekkor még mindig 1:3 lesz az esélye annak, hogy az általunk választott ajtó mögöttrejtőzik a kocs, és 2:3 hogy mégse ott. Így tehát 2:3 lesz annak is az esélye, hogy a másik ajtó mögött találjuk meg a főnyereményt, ezért jó, ha váltunk. Az alábbi táblázat bemutatja a nyerési esélyeinket:

1. ajtó	2. ajtó	3. ajtó	Monty kinyitja	Ha váltunk
kecske	kecske	autó	a 2. ajtót	nyerünk
kecske	autó	kecske	a 3. ajtót	nyerünk
autó	kecske	kecske	a 2. vagy a 3. ajtót	vesztünk

2.2. ábra. Monty Hall probléma

Láthatjuk, hogy ha mindig az első ajtót választjuk, és változtatunk a döntésünkön, akkor a három esetből kétszer nyerünk.

Most pedig nézzünk meg egy ezzel kapcsolatos R szimulációt. Először is megadjuk, hogy mennyi legyen a kísérletek száma (1000000). A kísérlet változóban deklaráljuk, hogy `sample()` függvénnyel, hogy

1-től 3-ig, generáljon random számokat, mégpedig annyiszor, ahány a `kiserletek_szama` változó értéke. A `replace=T` engedélyezi, hogy legyen ismétlődés a számok között. Ide kerül tehát, majd az, hogy hol lesz a nyeremény. ezután megint ugyanezt a kódrészletet adjuk meg a `jatekos` változónak is, mely a játékos választásait fogja tárolni. Majd a `musorvezeto` értékéhez azt adjuk meg, hogy mennyi a `kiserletek_szama` mérete.

Program mh.r

```
kiserletek_szama=10000000
kiserlet = sample(1:3, kiserletek_szama, replace=T)
jatekos = sample(1:3, kiserletek_szama, replace=T)
musorvezeto=vector(length = kiserletek_szama)
...
```

A következő `for` ciklus annyiszor fog lefutni, ahány a `kiserletek_szama`. Azon belül, ha a `kiserlet` adott eleme megegyezik a `jatekos` adott elemével, azaz eltalálta a helyes ajtót, akkor a `kiserlet` tömbből kivesszük az egyezést és így lesz elmentve az adat a `mibol` változóban. Ezt a `setdiff()` függvénnyel érhetjük el. Ellenkező esetben (`else`) mind a `kiserlet[]` és a `jatekos[]` tömbből ki kell venni az adott értéket. Ezután állítjuk össze a `musorvezeto[]` értékeit a `mibol` és a `sample()` függvény segítségével.

Program mh.r

```
...
for (i in 1:kiserletek_szama) {
  if(kiserlet[i]==jatekos[i]){
    mibol=setdiff(c(1,2,3), kiserlet[i])
  }else{
    mibol=setdiff(c(1,2,3), c(kiserlet[i], jatekos[i]))
  }
  musorvezeto[i] = mibol[sample(1:length(mibol),1)]
}
...
```

A `nemvaltoztatasesnyer` tömbbe kerülnek azok az adatok, hogy a `kiserlet` és a `jatekos` értékeit összehasonlítva hol találhatók egyezések. Ehhez a `which()` függvényt használjuk. A `valtoztat` vektorban megadjuk, hogy a változó értéke azonos legyen a `kiserletek_szama`-val.

Program mh.r

```
...
nemvaltoztatasesnyer= which(kiserlet==jatekos)
valtoztat=vector(length = kiserletek_szama)
```

...

Ezután megint for ciklus jön, megint ugyanannyiszor, ahány a `kiserletek_szama`. Dekráljuk a `holvalt` tömböt, mégpedig úgy, hogy a `setdiff()` függvénnyel kivesszük azokat az értékeket, amik egyeznek az adott indexben a `musorvezeto` és a `jatekos` értékeivel.

Program `mh.r`

...

```
valtoztatesnyer = which(kiserlet==valtoztat)
```

```
sprintf("Kiserletek szama: %i", kiserletek_szama)
```

```
length(nemvaltoztatesnyer)
```

```
length(valtoztatesnyer)
```

```
length(nemvaltoztatesnyer)/length(valtoztatesnyer)
```

```
length(nemvaltoztatesnyer)+length(valtoztatesnyer)
```

Ha ezzel megvagyunk, akkor a `valtoztatesnyer` változót ugyanúgy adjuk meg, ahogy eddig, majd kiíratjuk az adatokat.

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfiával megadva írd meg ezt a gépet!

Megoldás videó:

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/Chomsky/convert.c

A decimális számrendszer, vagy más néven tízes számrendszer egy 10 elemből álló halmaz, melynek első tagja a 0, az utolsó tagja pedig a 9. Ha ezt át szeretnénk váltani unáris, azaz egyes számrendszerbe, akkor a végeredményként egy csupa 1-esekből álló értéket kapunk. Az 1 csupán szimbólum, lehet helyettesíteni bármilyen más szimbólummal, a lényeg, hogy az N számot az általunk választott szimbólum N-szeri ismétlésével ábrázoljuk.

Ha egy decimálisból unárisba átváltó programot szeretnénk írni, akkor egy olyan ciklust kell létrehozni, ami mindig annyiszor írja ki az egyest, ahány számot adtunk meg. Ha az adott szám nulla, akkor eredményként nem fogunk kapni semmit, mivel a nullát nem képes egyes számrendszerben ábrázolni.

```
#include <stdio.h>

int main()
{
    int a;
    int count = 0;

    printf("Adj meg egy természetes számot\n");
    scanf("%d", &a);
    printf("A beírt szám unáris alakban:");

    for (int i = 0; i < a; ++i) {
        printf("1");
        count++;
    }

    printf("\n");
    return 0;
}
```

```
}
```

3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen

Mutass be legalább két környezetfüggetlen generatív grammatikát, amely ezt a nyelvet generálja!

Megoldás forrása az előadás fóliája

A Noam Chomsky nevéhez fűződő úgynevezett Chomsky-féle hierarchia 1-es típusa definiálja a környezetfüggetlen nyelvtanokat. Kétféle szabály létezik:

S, X, Y „változók”

a, b, c „konstansok”

$S \rightarrow abc, S \rightarrow aXbc, Xb \rightarrow bX, Xc \rightarrow Ybcc, bY \rightarrow Yb, aY \rightarrow aaX, aY \rightarrow aa$
S-ből indulunk ki

A szabályok betartása után megkapjuk:

```
S (S → aXbc)
aXbc (Xb → bX)
abXc (Xc → Ybcc)
abYbcc (bY → Yb)
aYbbcc (aY → aaX)
aaXbbcc (Xb → bX)
aabXbcc (Xb → bX)
aabbXcc (Xc → Ybcc)
aabbYbcc (bY → Yb)
aabYbbcc (bY → Yb)
aaYbbbcc (aY → aa)
aaabbbccc
```

A feladatban szereplő $a^n b^n c^n$ nyelvet az alábbi generálja:

A, B, C „változók”

a, b, c „konstansok”

$A \rightarrow aAB, A \rightarrow aC, CB \rightarrow bCc, cB \rightarrow Bc, C \rightarrow bc$
A-ból indulunk ki, ez lesz a kezdőszimbólum.

A szabályokat követve:

```
A (A → aAB)
aAB (A → aAB)
aaABB (A → aAB)
aaaABBB (A → aC)
aaaaCBBB (CB → bCc)
aaaabCcBB (cB → Bc)
aaaabCBcB (cB → Bc)
aaaabCBBc (CB → bCc)
aaaabbCcBc (cB → Bc)
aaaabbCBcc (CB → bCc)
aaaabbbCccc (C → bc)
aaaabbbbcccc
```

3.3. Hivatkozási nyelv

A [KERNIGHANRITCHIE] könyv C referencia-kézikönyv/Utasítások melléklete alapján definiáld BNF-ben a C utasítás fogalmát! Majd mutass be olyan kódcsipeteket, amelyek adott szabvánnyal nem fordulnak (például C89), mással (például C99) igen.

Megoldás videó:

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/Chomsky/szabvany.

A BNF (Backus-Naur Form) egy olyan metanyelv, melynek segítségével szabályok alkothatók meg. Legfőbb célja a programozási nyelvek szintaxisának leírása. Főbb részei:

- <név<

Olyan fogalmak tartoznak ide, mint pl.: azonosító, betű, törtszám, prízszám, stb. Tehát bármi lehet. Nem terminális elem.

- ::=

Ez fogja elválasztani a szabály bal- és jobb oldalát.

- A definiálandó nyelv karakterkészlete, azaz a terminálisok. Az elválasztó jel jobb oldalán helyezkedik el.

Ezek alapján néhány C utasítások BNF-ben:

```
//feltételes utasítás
if (<kifejezés>)
    <utasítás>
else if (<kifejezés>)
    <utasítás>
else (<kifejezés>)
    <utasítás>

//while utasítás
while (<kifejezés>)
    do <utasítás>
```

Most pedig jöjjön egy olyan kódcsipet, amely c89-cel nem, de c99-el működik:

```
int main()
{
    //comment
    return 0;
}
```

A fenti kódot ha c89-cel fordítjuk le, akkor az alábbi hibaüzenetet fogunk kapni:

```
gcc -std=c89 szabvany.c -o szabvany

szabvany.c: In function 'main':
szabvany.c:3:5: error: C++ style comments are not allowed in ISO C90
    //comment
    ^
szabvany.c:3:5: error: (this will be reported only once per input file)
```

Ha pedig c99-cel, akkor gond nélkül működik:

```
gcc -std=c99 szabvany.c -o szabvany
```

3.4. Saját lexikális elemző

Írj olyan programot, ami számolja a bemenetén megjelenő valós számokat! Nem elfogadható olyan megoldás, amely maga olvassa betűnként a bemenetet, a feladat lényege, hogy lexert használjunk, azaz óriások vállán álljunk és ne kispályázzunk!

Megoldás videó:

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/Chomsky/lexer.l

Az alábbi kódot .l végződésű fájlba kell menteni, mert a lex programmal fogunk dolgozni. A program segítségével lexikális szabályokból lexikális elemző programkódot fogunk generálni. Ehhez a következő parancsot kell megadnunk: `lex -o lexer.c lexer.l`, ami elkészíti számunka a c forráskódot. Ezután már csak a szokásos módon gcc-vel fordítjuk le, azzal a különbséggel, hogy a végéhez beírjuk az `-lfl` is: `gcc lexer.c -o lexer -lfl`.

```
%{
    #include <stdio.h>

    int realnumbers = 0;
}%
digit [0-9]
%%
{digit}*({digit}+)? {++realnumbers;
    printf("[realnum=%s %f]", yytext, atof(yytext));}
%%
int
main ()
{
    yylex ();
    printf("The number of real numbers is %d\n", realnumbers);
    return 0;
}
```

A fenti program 3 fő részből áll, amelyeket a %% szimbólum választja szét őket. Az első rész tartalmazza a headert és a változót. Itt bármilyen C kódot megadhatunk a %{ és %} jelek között. Ezt a Flex módosítás nélkül fogja átmásolni a generálandó kódba.

A második rész 2 részből álló szabályokat tartalmaz: Először a reguláris kifejezésekkel kezdjük, utána jön a kapcsos zárójelbe zárva a C kód. A reguláris kifejezéseknél határozzuk meg a keresési szabályokat. Jelen esetben számmal kezdődött keresünk (digit), ami akár többször is előfordulhat (*). Ezt követően van egy zárójeles rész is, ami azt jelenti, hogy ????? Az ezt követő C kódban adjuk meg, hogy majd a terminálba bevitt szöveget vizsgálja meg. Az atof() függvény segítségével a string-ből double lesz. Tehát ezekből a szabályokból létrejön majd a yylex nevű függvény és return esetén újra ide fog visszakérülni.

A program harmadik részében látható egy yylex() függvény, ami a fenti szabályokat hívja elő, továbbá kiírjuk az eredményt.

3.5. l33t.l

Lexelj össze egy l33t ciphert!

Megoldás forrása: https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/Chomsky/leet.l

A leet nyelv lényege, hogy bizonyos betűket számokkal vagy ASCII karakterekkel szokták helyettesíteni. Innen jön az, hogy leet helyett l33t írunk. A mostani példában olyan programot mutatok be, ami az adott szövegben ezeket a betűket kicseréli a megfelelő számokkal, vagy karakterekkel. A megadott forráskódot is lex programmal kell C-be fordítani. Vizsgáljuk meg a programot. Ahogy az előző feladatnál, úgy itt is 3 részből áll.

Az első részben definiáljuk a L337SIZE konstansszerű elemet. Ezt a #define segítségével oldjuk meg és a nevet mindig csupa nagybetűvel kell megadni. Ha a programban valahol beírjuk a megadott konstans nevet (L337SIZE), akkor a helyére fogja majd behelyettesíteni a hozzá tartozó szöveget. Jelen esetben ehhez a változóhoz meg van adva egy char c és egy hozzá tartozó char *leet[4] tömb, ebből a kettőből fog összeállni a l337d1c7[] tömb, ahol az előbbi lesz majd a kicserélendő betű, utóbbi pedig az, hogy mire lehet majd kicserélni (4 választási lehetőség van).

```
#define L337SIZE (sizeof l337d1c7 / sizeof (struct cipher))

struct cipher {
    char c;
    char *leet[4];
} l337d1c7 [] = {

    {'a', {"4", "4", "@", "/-\\"}},
    ...
}
```

Ha az egyes betűkhöz hozzárendeltük a megfelelő számokat vagy karaktereket. Akkor a második részben fogjuk megadni C-ben, hogy hogyan történjen a karaktercsere:

```
...
{
```

```
int found = 0;
for(int i=0; i<L337SIZE; ++i)
{
    if(l337d1c7[i].c == tolower(*yytext)) //Megegyezik-e a két karakter?
    {
        int r = 1+(int) (100.0*rand()/(RAND_MAX+1.0)); //random szám ←
        generálása 1 és 100 között

        if(r<91)
            printf("%s", l337d1c7[i].leet[0]);
        else if(r<95)
            printf("%s", l337d1c7[i].leet[1]);
        else if(r<98)
            printf("%s", l337d1c7[i].leet[2]);
        else
            printf("%s", l337d1c7[i].leet[3]);

        found = 1;
        break;
    }
}

if(!found)
    printf("%c", *yytext);
}
...
```

A kapcsos zárójel elején egy pont van, ami azt jelenti, hogy minden egyes karaktert vizsgáljon meg a szövegben. Ez lesz tehát a reguláris kifejezés. A C részben látható, hogy a `for` ciklusban előhívjuk a `L337SIZE` nevű konstanst, tehát ide fogja behellyettesíteni azt, amit már az első részben megadtunk.

Szükségünk lesz a `tolower()` függvényre is, mert ha nagybetű van a szövegben, akkor azt nem fogja számításba venni a program, ezért ezzel a függvénnyel minden betűt kisbetűvé alakítunk át. Ezután a kapott random szám alapján cseréljük ki az adott betűket. Tehát például az "a" betűt kell kicserélni, és `r` értéke 97, akkor ez azt jelenti, hogy az "a" betű "@"-ra lesz kicserélve, mivel az `if-else` utasításban itt fog teljesülni:

```
...
    else if(r<98)
        printf("%s", l337d1c7[i].leet[2]);
...
```

Ez azt jelenti, hogy `l337d1c7[i]` tömbön belül a `leet[2]` tömb 3. eleme legyen kiválasztva, ami a "@". Előfordul azonban, hogy a két karakter nem egyezik meg a vizsgálat során (`if(!found)`), ezért ilyenkor az eredeti karaktert írjuk ki.

3.6. A források olvasása

Hogyan olvasod, hogyan értelmezed természetes nyelven az alábbi kódcsipeteket? Például

```
if(signal(SIGINT, jelkezeslo)==SIG_IGN)
    signal(SIGINT, SIG_IGN);
```

Ha a SIGINT jel kezelése figyelmen kívül volt hagyva, akkor ezen túl is legyen figyelmen kívül hagyva, ha nem volt figyelmen kívül hagyva, akkor a jelkezeslo függvény kezelje. (Miután a **man 7 signal** lapon megismertem a SIGINT jelet, a **man 2 signal** lapon pedig a használt rendszerhívást.)



Bugok

Vigyázz, sok csipet kerülendő, mert bugokat visz a kódba! Melyek ezek és miért? Ha nem megváránzésre, elkapja valamelyiket esetleg a splint vagy a frama?

Mielőtt átnéznénk a kódcsipeteket, előbb fontosnak tartom megemlíteni, hogy a kódokat úgy vizsgáljuk meg, hogy feltételezzük, az adott változók, függvények értéke már előzetesen deklarálva vannak.

- i.
- ```
if(signal(SIGINT, SIG_IGN)!=SIG_IGN)
 signal(SIGINT, jelkezeslo);
```

Ez a kódcsipet az előző példa ellentettje. Azt jelenti, hogy ha a SIGINT jel kezelése nem volt figyelmen kívül hagyva, akkor ezután se legyen figyelmen kívül hagyva.

- ii.
- ```
for(i=0; i<5; ++i)
```

A for ciklus 5-ször hajtódik végre. Az i változó értéke 0-tól 4-ig fog változni. A ++i azt jelenti, hogy i értékét előbb növeljük és ez lesz az új érték.

- iii.
- ```
for(i=0; i<5; i++)
```

Ez a ciklus ugyanúgy fog végrehatjtódni, mint az előző. Itt is ugyanúgy 0-tól 4-ig fog számolni. Az i++ pedig azt jelenti, hogy i értéke marad a régi, majd utána növeli 1-el.

- iv.
- ```
for(i=0; i<5; tomb[i] = i++)
```

A for ciklus kicseréli a tömb elemeit. A tömb első eleme marad a régi, az azt követő elemek értéke azonban már rendre 0, 1, 2, 3 lesz. Ha a tömb mérete 5-nél nagyobb, akkor a többi értéket figyelmen kívül hagyja, mivel a for ciklus csak 5-ször fut le. Ezért tehát a program bár gond nélkül le fog futni, de bugos.

- v.
- ```
for(i=0; i<n && (*d++ = *s++); ++i)
```

A for ciklusnak itt 2 felétele van, ugyanis látható, hogy van egy && (ÉS) logikai operátor is. A ciklus addig fog futni, amíg i értéke kisebb mint n. A logikai operátor jobb oldalán nincs relációs operátor megadva, ezért ez a program bugos. A változók előtti \* jel azt jelenti, hogy egy adott tárterületre mutatnak a változók értékei.

vi.

```
printf("%d %d", f(a, ++a), f(++a, a));
```

Az `printf()` függvény segítségével írjuk ki a zárójelben megadott paraméterek alapján. Jelen esetben a cél az lenne, hogy kiíratjuk az `f()` függvények visszatérési értékét, de ez a kód bugos, ugyanis az `f()` függvény két paramétert kér, de ezek kiértékelési sorrendje nincs meghatározva.

vii.

```
printf("%d %d", f(a), a);
```

Ez a kód már helyes. Az `f()` függvény egy paramétert kér, ami jelen esetben az `"a"` lesz, így ennek az értéke kerül kiíratásra.

viii.

```
printf("%d %d", f(&a), a);
```

Ez a kód is le fog fordulni, csak itt az `f()` függvénynél `&a` argumentum van megadva, ami azt jelenti, hogy egy terület memóriacímére mutató értéket ad paraméterül.

### 3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

```
$(\forall \text{forall } x \ \exists \text{exists } y \ ((x < y) \wedge (y \ \text{text}\{ \text{prím}})))$
```

```
$(\forall \text{forall } x \ \exists \text{exists } y \ ((x < y) \wedge (y \ \text{text}\{ \text{prím}}) \wedge (\exists \text{exists } z \ \text{text}\{ \text{prím}})) \leftrightarrow)$
```

```
$(\exists \text{exists } y \ \forall \text{forall } x \ (x \ \text{text}\{ \text{prím}}) \supset (x < y)) \ $
```

```
$(\exists \text{exists } y \ \forall \text{forall } x \ (y < x) \supset \neg (x \ \text{text}\{ \text{prím}}))$
```

Megoldás forrása: [https://gitlab.com/nbatfai/bhax/blob/master/attention\\_raising/MatLog\\_LaTeX](https://gitlab.com/nbatfai/bhax/blob/master/attention_raising/MatLog_LaTeX)

Megoldás videó: <https://youtu.be/ZexiPy3ZxsA>, [https://youtu.be/AJSXOQFF\\_wk](https://youtu.be/AJSXOQFF_wk)

Végtelen sok prímszám van.

```
$(\forall \text{forall } x \ \exists \text{exists } y \ ((x < y) \wedge (y \ \text{text}\{ \text{prím}})))$
```

Végtelen sok ikerprímszám van.

```
$(\forall \text{forall } x \ \exists \text{exists } y \ ((x < y) \wedge (y \ \text{text}\{ \text{prím}}) \wedge (\exists \text{exists } z \ \text{text}\{ \text{prím}})) \leftrightarrow \text{text}\{ \text{prím}})))$
```

Véges sok prímszám van.

```
$(\exists \text{exists } y \ \forall \text{forall } x \ (x \ \text{text}\{ \text{prím}}) \supset (x < y)) \ $
```

Véges sok prímszám van.

```
$(\exists \text{exists } y \ \forall \text{forall } x \ (y < x) \supset \neg (x \ \text{text}\{ \text{prím}}))$
```



## 3.8. Deklaráció

Megoldás videó:

Megoldás forrása:

- egész

```
int egesz = 5;
```

- egészre mutató mutató

```
*mutato = &egesz;
```

- egész referenciája

```
int &r_egesz = egesz;
```

- egészek tömbje

```
int tomb[3] = {1,2,3};
```

- egészek tömbjének referenciája (nem az első elemé)

```
int (&r_tomb)[3] = tomb;
```

- egészre mutató mutatók tömbje

```
int *m_tomb[3];
```

- egészre mutató mutatót visszaadó függvény

```
int* e_mutato = f(mutato);
```

- egészre mutató mutatót visszaadó függvényre mutató mutató

```
int* (*f_mutato)(int*) = f;
```

- egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvény

```
int (*getInt (int a)) (int b, int c);
```

- függvénymutató egy egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényre

Mit vezetnek be a programba a következő nevek?

- ```
int a;
```

Integer, azaz egész típusú változót, melynek változóneve: a

- ```
int *b = &a;
```

Az első sorban bevezetett a változóra mutató mutató lesz a \*b.

- ```
int &r = a;
```

Az a változó referenciája lesz az &r

- ```
int c[5];
```

Integer típusú, c változónevű tömb deklarálása, melynek mérete 5.

- ```
int (&tr)[5] = c;
```

A c tömb referenciája.

- ```
int *d[5];
```

Ez egy integer típusú tömbre mutató mutató.

- ```
int *h ();
```

Integer típusú h függvényre mutató mutató.

- ```
int *(*l) ();
```

Egy egészre mutató mutatóval visszatérő függvényre mutató mutató.

- ```
int (*v (int c)) (int a, int b)
```

Egy egészszel visszatérő, 2 egész paramétert váró függvényre mutató mutató.

- ```
int (*(z) (int)) (int, int);
```

függvénymutató, mely egy egészet visszaadó, két egész paramétert váró függvényre mutató mutatót visszaadó, egészet kapó függvény.

## 4. fejezet

# Helló, Caesar!

### 4.1. `int ***` háromszögmátrix

Írj egy olyan `malloc` és `free` párost használó C programot, amely helyet foglal egy alsó háromszög mátrixnak a szabad tárban!

Megoldás videó:

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/blob/master/attention\\_raising/Source/caesar/tm.c](https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/caesar/tm.c)

Háromszögmátrixnak nevezünk egy négyzetes mátrixot, melynek főátlója alatt vagy felett minden elem nulla. Ez alapján tehát két típusa van: alsó háromszögmátrix és felső háromszögmátrix.

Az `int nr = 5` változóban deklaráljuk a sorok számát. A `print()` függvénnyel kiíratjuk a `tm`-nek lefoglalt memória címét. Itt a `&` szimbólum azt jelenti, hogy valaminek a memóriacímére mutat. Az `if()` függvényen belül a `malloc` visszaad nekünk egy pointert, ami a lefoglalt memóriára mutat. Pontosabban a `malloc` egy `void` típusú mutatót visszaadó függvény, ami csak egy paramétert vár, mégpedig a lefoglalandó tárterület mennyiségét bájtokban. A függvény használatához tudnunk kell az egyes adattípusok méretét, ezért kell használnunk a `sizeof()` operátort, amely az adott adattípus bájtokban megadott méretével tér vissza. Ennek az operátornak a paramétere lehet egy változó, egy tömb, egy kifejezés vagy egy adattípus. Mi most az utóbbit adjuk meg, azaz legyen `double` típusú, ami mondjuk legyen 8 bájt, amit megszorunk `nr` értékével, mivel nekünk 5 sorunk van. Ha az `if` függvényben lévő feltétel egyenlő `NULL`, akkor hiba van, ezért kilép és `-1`-et ad vissza. Ezután kiíratom, amit a `malloc()` függvény a `tm`-re visszaad.

```
...
int nr = 5;
double **tm;

printf("%p\n", &tm); //a tm memóriacím kiíratása. A & szimbólum azt jelenti ←
, hogy a memóriacímre mutat.

if ((tm = (double **) malloc (nr * sizeof (double *))) == NULL)
{
 return -1;
}

printf("%p\n", tm);
```

```
...
```

Ezután egy for ciklus következik, ami 5-ször fog lefutni. Az if() függvényen belül a malloc() visszaad egy pointert a lefoglalt memóriákra. Minden sorban egyre több 8 bájtnyi memóriát foglal le ((i + 1) \* sizeof (double)). Itt is ugyanúgy ellenőrizzük, hogy teljesül-e a feltétel, mint a fenti kódrészletben.

```
...
for (int i = 0; i < nr; ++i)
{
 if ((tm[i] = (double *) malloc ((i + 1) * sizeof (double))) == NULL)
 {
 return -1;
 }
}

printf("%p\n", tm[0]);
...
```

Végül kiíratjuk a tm tömb adatait egyenként, úgy, hogy alsó háromszögmátrixot kapjunk. Ezt úgy érhetjük el, hogy a mátrix minden elemét ezzel a képlettel számoljuk ki:  $tm[i][j] = i * (i + 1) / 2 + j$ ;

```
...
for (int i = 0; i < nr; ++i)
 for (int j = 0; j < i + 1; ++j)
 tm[i][j] = i * (i + 1) / 2 + j;

for (int i = 0; i < nr; ++i)
{
 for (int j = 0; j < i + 1; ++j)
 printf("%f, ", tm[i][j]);
 printf("\n");
}
...
```

## 4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/blob/master/attention\\_raising/Source/caesar/EXOR/-exor.c](https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/caesar/EXOR/-exor.c)

Ebben a feladatban XOR - magyarul: kizáró vagy - titkosítással foglalkozunk. Ennek az egyszerű titkosítási eljárásnak a lényege, hogy a szöveget egy kulcs segítségével titkosíthatjuk és visszafelé pedig ugyanezzel

a kulccsal tudjuk dekódolni a titkosított adatot. A maximális biztonság eléréséhez, amit egy külső támadó sem tud feltörni, az alábbi feltételeket kell figyelembe venni:

- A kulcs ugyanolyan hosszú legyen, mint a titkosítandó szöveg. Ellenkező esetben, ha a kulcs rövidebb, akkor az ismétlődni fog a kimeneti bitsorozatban, amiből már kinyirehető egy szövegrészlet.
- A kulcs véletlenszerűen generált legyen, ne használjuk ugyanazt a kulcsot minden titkosítás során.

A forrásban látható program is egy XOR titkosítási eljárást használ. Első lépésként a `#define` előfeldolgozó utasítás használatával definiálunk két változónevet, amiket csupa nagybetűvel adunk meg, majd ezekhez konstans értékeket rendelünk hozzá. Ezeket a változóneveket bárhol használhatjuk a programban, ahol az előfeldolgozó az ahhoz megadott értékeket helyezi majd el a fordítási folyamat előtt.

```
#define MAX_KULCS 100
#define BUFFER_MERET 256
```

Ezután a szokásos `main()` függvényt kiegészítjük két argumentummal. Erre azért van szükség, mert amikor a futtatási rendszer meghívja a `main()` függvényt, akkor az alábbi két paramétert fogja átadni:

- `argc` (argument count): A parancssorban kapott egész számot adja meg.
- `**argv` (argument vector): Lényegében egy tömb, amely a karakterláncok mutatóit tartalmazza.

Ezt a kódrészletet úgy kell elképzelni a gyakorlatban, hogy a program kérni fog tőlünk adatokat, amiket be kell majd írunk. Jelen esetben a kulcsot és a szöveget.

Az `main()` függvényen belül két `char` típusú tömböket deklarálunk (`kulcs[]`, `buffer[]`), amelyek méretét a már fent említett `#define` konstans változónevek használatával adjuk meg. Ezt követően két `integer` típusú változót is megadunk (`kulcs_index`, `olvasott_bajtok`), melyek értéke mindkét esetben 0 lesz. Az előbbi a `kulcs[]` tömb indexét fogja majd megadni. Deklarálunk még egy `kulcs_meret` változót is, ami az `argv[]` tömb második paraméterének karakterlánc hosszát fogja kiszámolni az `strlen()` függvény segítségével. Látható még egy `strncpy()` függvény is, ami azt fogja csinálni, hogy egyik helyről a másikba másolja át az adatokat és megadhatjuk a a karaktermásolás számát is. Jelen esetben a kulcs tömbbe másoljuk az `argv[1]` adatait és a másolás mérete `MAX_KULCS` konstans értéke lesz.

```
...
int main(int argc, char **argv)
{
 char kulcs[MAX_KULCS];
 char buffer[BUFFER_MERET];

 int kulcs_index = 0;
 int olvasott_bajtok = 0;

 int kulcs_meret = strlen (argv[1]);
 strncpy (kulcs, argv[1], MAX_KULCS);
 ...
}
```

A változók deklarálása után szükségünk van egy `while` ciklusra, ami addig fog futni, amíg a byte-ok olvasása tart. A `read()` függvény 3 paramétert kér:

- 0: A 0 szám itt azt jelenti, hogy a standard inputról fogja beolvasni az adatot. Ez a rész mindig egy int típusú fájlleíró lesz. 3 típusa van: a már megadott 0, 1 (standard output), 2 (standard error).
- (void\*) buffer: A beolvasott adatok a buffer tömbbe lesznek eltárolva.
- BUFFER\_MERET: Az olvasandó byte-ok száma.

```
...
while ((olvasott_bajtok = read (0, (void *) buffer, BUFFER_MERET)))
{
 for (int i = 0; i < olvasott_bajtok; ++i)
 {
 buffer[i] = buffer[i] ^ kulcs[kulcs_index];
 kulcs_index = (kulcs_index + 1) % kulcs_meret;
 }

 write (1, buffer, olvasott_bajtok);
}
```

A while cikluson belül egy for ciklus lesz, ami addig fog futni, amíg el nem éri az olvasott byte-ok számát. Az adott buffer[i] karakter ki lesz cserélve az adott kulccsal exor segítségével (^). Ezután a kulcs\_index értékét megnöveljük 1-el és ez addig fog futni, amíg tart a ciklus. A for cikluson kívül ezután előhívjuk a write() függvényt, mely hasonló a read() függvényhez, csak itt a standard outputra (1) írjuk ki a buffer-ben eltárolt adatokat és az írás mérete az olvasott\_bajtok lesz.

A program működését az alábbi képen lehet látni:

```
dave@dave-K501UB:~/gyakorlas$ more tiszta.txt
A titkosítás vagy rejtjelezés a kriptográfiának az az eljárása, amellyel az információt (nyílt szöveg) egy algoritmus
(titkosító eljárás) segítségével olyan szöveggé alakítjuk, ami olvashatatlan olyan ember számára, aki nem rendelkezi
k az olvasáshoz szükséges speciális tudással, amit általában kulcsnak nevezünk. Az eredmény a titkosított információ
(titkosított szöveg). Sok titkosító eljárás egy az egyben (vagy egyszerű átalakítással) használható megfejtésre is, a
zaz, hogy a titkosított szöveget újra olvashatóvá alakítsa.
dave@dave-K501UB:~/gyakorlas$ gcc exor.c -o exor -std=c99
dave@dave-K501UB:~/gyakorlas$./exor 00012345 <tiszta.txt >titkos.txt
dave@dave-K501UB:~/gyakorlas$ more titkos.txt
q[0]XFX[F00D000[0]QWI[0]V^AZU\TH00F[0]0[0]ZDA_WB00U]00^0Z[0]BN[0]J[0]^Y00B00BS[0]U\]KVX[0]J[0]U[G]00R[0]00A[0]H00XA[0]J00EQR[0]00VK
00YW BXF^AF[0]0XFX[F0
0D00[0]YZ00C00G[0]0UV00@F00H00EQY[0]HS]00J00GWTs00[0]SX00DZEZ[0]0XY[0]DRG]QDQE^RZ[0]IP\00XRUB[0]I00]00CS[0]0Y[0]W^00U^TT^X
QOY[0]PH[0]YFQC00@Z
J[0]K00 F00NTA[0]EUSY00_]F[0]EU00GFQ\000^]A[0]00]FRX00RQ_00AYSC^PY[0]PFUJ00_]000J[0]HAQQ]00_K[0]00YDZ]@00D_DE[0]ZS_B]00P]00[0]0[
G ZC00E]G00J00DV
S[0]000^Y[0]0D[_B000000]X00G00C[0]TM[0]J[0]UJVP^000STM[0]WIBHVF00[0]FRXT[00E00GFQ\000ZRG0^00]ZR000[0]TUUQ_D00B@V[0]C[0]HHRN[0]00_
VK[0]00YDZ]@00D_DE[0]N00F
UVWG[0]0ZBP[0]XCQCXPF00C00[0]^R_00DCP[0]0
dave@dave-K501UB:~/gyakorlas$
```

4.1. ábra. C Exor titkosítás

## 4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/blob/master/attention\\_raising/Source/caesar/EXOR/-/javaexor.java](https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/caesar/EXOR/-/javaexor.java)

Javában először mindig egy osztályt kell létrehozni, ami jelen esetben az `ExorTitkosító` lesz. Ezen belül a `public ExorTitkosító()` zárójelében deklaráljuk a `kulcsSzöveg`, ami egy `String` típusú változó lesz, továbbá java függvényeket hívunk elő, mint ahogy C-nél tettük `include` használatával, csak itt mindig `java.io.`-val kezdődik és a függvény nevével fejeződik be. Mivel ezek általában hosszúak, a függvény után szóközzel elválasztva adhatunk neki, egy számunkra sokkal olvashatóbb nevet, mint ahogy a példában a `bejövőCsatorna` és a `kimenőCsatorna` neveket adtuk meg.

```
public class ExorTitkosító {

 public ExorTitkosító(String kulcsSzöveg,
 java.io.InputStream bejövőCsatorna,
 java.io.OutputStream kimenőCsatorna)
 throws java.io.IOException {

 byte [] kulcs = kulcsSzöveg.getBytes();
 byte [] buffer = new byte[256];
 int kulcsIndex = 0;
 int olvasottBájtok = 0;

 while((olvasottBájtok =
 bejövőCsatorna.read(buffer)) != -1) {

 for(int i=0; i<olvasottBájtok; ++i) {

 buffer[i] = (byte)(buffer[i] ^ kulcs[kulcsIndex]);
 kulcsIndex = (kulcsIndex+1) % kulcs.length;

 }

 kimenőCsatorna.write(buffer, 0, olvasottBájtok);

 }

 }

 ...
}
```

A változók deklarálása után a `while()` ciklust hozunk létre, ami addig fog futni, amennyi a buffer mérete. A `for` ciklusban pedig megtörténik a már C változatban ismertett EXOR-ozás. Itt is karakterenként cseréljük ki az adott kulcssal a szöveg tartalmát. Eddig ez a program önmagában nem fog csinálni semmit, mert ez csak egy segédfüggvény volt. Szükség van egy `public static void main()` függvényre is, ami a C program `int main()` függvényre hasonlít.

```
...
public static void main(String[] args) {

 try {
 new ExorTitkosító(args[0], System.in, System.out);

 } catch (java.io.IOException e) {

 e.printStackTrace();

 }

}
```

Itt a try és catch páros azt jelenti, hogy vizsgáljuk meg, hogy az ExorTitkosító() function-be bevitt adatok helyesen lettek-e megadva. Ha igen, akkor hajtsa végre a feladatot, ellenkező esetben jön a catch() függvény, hogy kiírassuk a hibaüzenetet.

## 4.4. C EXOR törő

Írj egy olyan C programot, amely megtöri az első feladatban előállított titkos szövegeket!

Megoldás videó:

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/blob/master/attention\\_raising/Source/caesar/EXOR/-/tores.c](https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/caesar/EXOR/-/tores.c)

A forrásban található programnak az a célja, hogy az első feladatban titkosított szöveget addig próbáljuk a program által generált kulcsokkal feltörni, amíg tiszta szöveget nem kapunk. Jelen programot leszűkítettük, 8 számjegyű kulcsokra és minden egyes számjegy 0-tól 9-ig terjed. Erre azért van szükség, mert ha az összes lehetséges karaktert és méretet megvizsgáljuk akkor nagyon sokáig tartana, mire a program feltöri a titkos szöveget. A generált kulcsokat a kulcs[] tömbben tároljuk el.

```
...
for (int ii = '0'; ii <= '9'; ++ii)
 for (int ji = '0'; ji <= '9'; ++ji)
 for (int ki = '0'; ki <= '9'; ++ki)
 for (int li = '0'; li <= '9'; ++li)
 for (int mi = '0'; mi <= '9'; ++mi)
 for (int ni = '0'; ni <= '9'; ++ni)
 for (int oi = '0'; oi <= '9'; ++oi)
 for (int pi = '0'; pi <= '9'; ++pi)
 {
 kulcs[0] = ii;
 kulcs[1] = ji;
 kulcs[2] = ki;
 kulcs[3] = li;
 kulcs[4] = mi;
```



```
 kulcs[5] = ni;
 kulcs[6] = oi;
 kulcs[7] = pi;

 if (exor_tores (kulcs, KULCS_MERET, titkos, ←
 p - titkos))
 printf
 ("Kulcs: [%c%c%c%c%c%c%c%c]\nTiszta szoveg: ←
 [%s]\n",
 ii, ji, ki, li, mi, ni, oi, pi, titkos);

 // ujra EXOR-ozunk, így nem kell egy ←
 // masodik buffer
 exor (kulcs, KULCS_MERET, titkos, p - ←
 titkos);
 }

 ...
```

Ha egy kulcsot legenerált a program, akkor `if()` függvénnel megvizsgáljuk, hogy működik-e az adott kulcs. Látható, hogy itt az `exor_tores()` függvényt hívjuk elő, ami 4 paramétert kér. Ha a kulcs helyes, akkor kiíratjuk az eredményt, ellenkező esetben újra EXOR-ozunk. Az `exor_tores()`-en belül 2 függvény látható. Az egyik az `exor()` függvény, ahol ténylegesen történik az exor-ozás, a másik pedig a `tiszta_lehet()` ez fogja majd az értéket visszaadni a `return` függvénnel.

```
...
void
exor (const char kulcs[], int kulcs_meret, char titkos[], int titkos_meret)
{
 int kulcs_index = 0;

 for (int i = 0; i < titkos_meret; ++i)
 {
 titkos[i] = titkos[i] ^ kulcs[kulcs_index];
 kulcs_index = (kulcs_index + 1) % kulcs_meret;
 }
}

int
exor_tores (const char kulcs[], int kulcs_meret, char titkos[], int ←
 titkos_meret)
{
 exor (kulcs, kulcs_meret, titkos, titkos_meret);

 return tiszta_lehet (titkos, titkos_meret);
}
```

```
}
...
```

A `tiszta_lehet()` függvényen belül van egy `double` típusú `szohossz` változó, ami a megszámlolt titkos szavak méretének átlagát tartalmazza. Ezalatt a `return` függvénnyel megvizsgáljuk, hogy a `szohossz` mérete nagyobb-e, mint 6 és kisebb mint 9 továbbá, `strcasestr()` függvénnyel megvizsgáljuk, hogy tartalmazza-e az adott szavakat ("hogy", "nem", "az", "ha"). Azért pont ezeket a szavakat, mert ezek a leggyakoribb magyar szavak, amik előfordulhatnak. Ezeket akár bővíthatjuk további szavakkal is. Ha ezek teljesülnek, akkor nagy valószínűséggel a generált kulcs helyes, így a `return true` értéket ad vissza és megtörténik a `main()` függvényben a kiíratás.

```
...
double
atlagos_szohossz (const char *titkos, int titkos_meret)
{
 int sz = 0;
 for (int i = 0; i < titkos_meret; ++i)
 if (titkos[i] == ' ')
 ++sz;

 return (double) titkos_meret / sz;
}

int
tiszta_lehet (const char *titkos, int titkos_meret)
{
 // a tiszta szoveg valszeg tartalmazza a gyakori magyar szavakat
 // illetve az átlagos szóhossz vizsgálatával csökkentjük a
 // potenciális töréseket

 double szohossz = atlagos_szohossz (titkos, titkos_meret);

 return szohossz > 6.0 && szohossz < 9.0
 && strcasestr (titkos, "hogy") && strcasestr (titkos, "nem")
 && strcasestr (titkos, "az") && strcasestr (titkos, "ha");
}
...
```

Ha a program helyesen működik, akkor a terminálban az alábbi kellene kapnunk a program elindítása után:

```
sar/EXOR$./tores <titkos.txt
Kulcs: [00012345]
Tiszta szoveg: [A titkosítás vagy rejtjelezés a kriptográfiának az az eljárása,
amellyel az információt (nyílt szöveg) egy algoritmus (titkosító eljárás) segíts
égével olyan szöveggé alakítjuk, ami olvashatatlan olyan ember számára, aki nem
rendelkezik az olvasáshoz szükséges speciális tudással, amit általában kulcsnak
nevezünk. Az eredmény a titkosított információ (titkosított szöveg). Sok titkosí
tó eljárás egy az egyben (vagy egyszerű átalakítással) használható megfejtésre i
s, azaz, hogy a titkosított szöveget újra olvashatóvá alakítsa.
]
```

4.2. ábra. C Exor törés

## 4.5. Neurális OR, AND és EXOR kapu

Megoldás videó: <https://youtu.be/Koyw6IH5ScQ>

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/blob/master/attention\\_raising/Source/caesar/neutralis.c](https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/caesar/neutralis.c)

Ebben a feladatban a gépet tanítjuk meg arra, hogy az általunk megadott adatokból végezzen számítást, úgy hogy a végeredmény ugyanannyi legyen. A program elindításához R szimulációra lesz szükségünk.

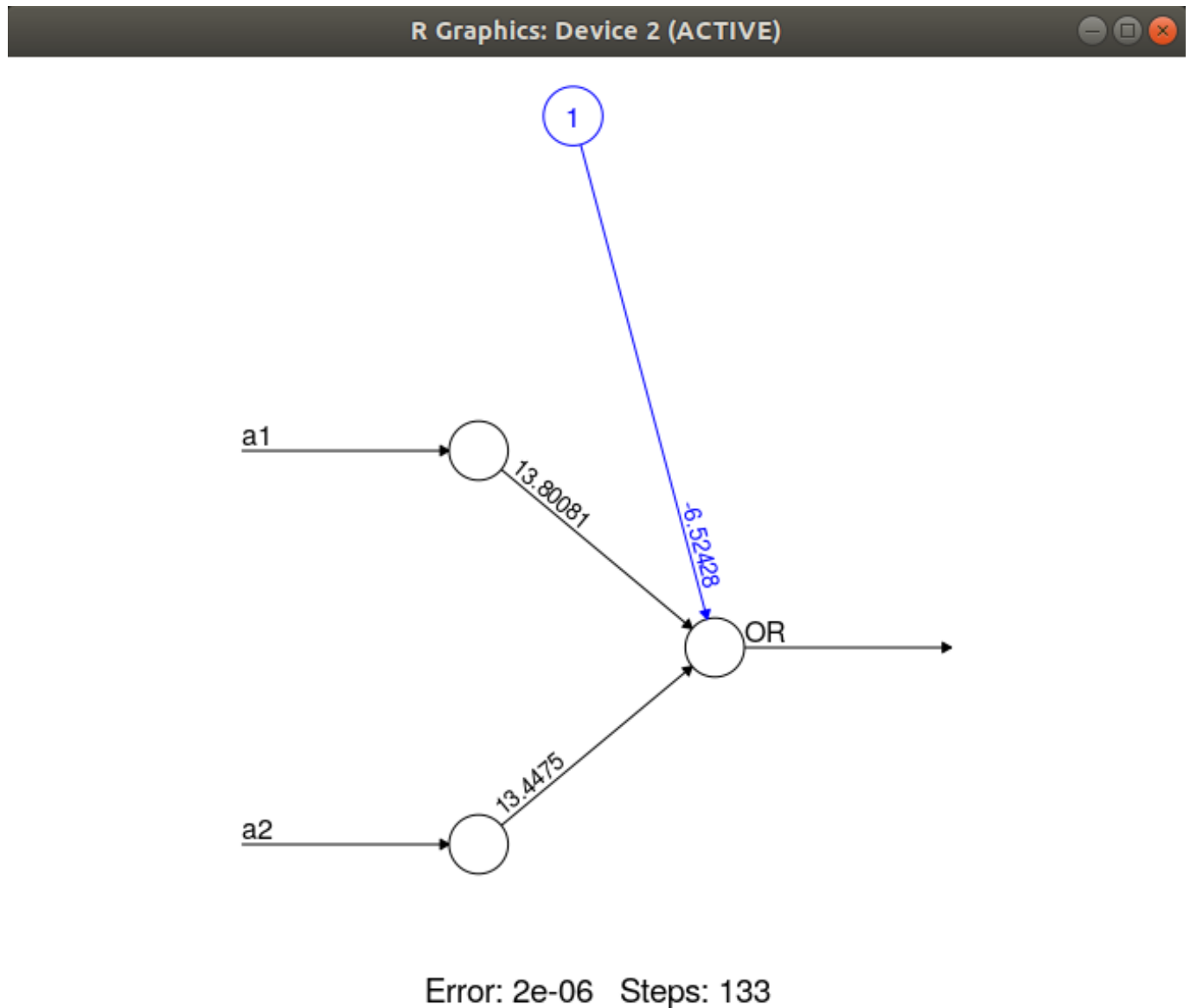
Az első példában 2 inputunk van (a1 és a2) és egy kimenet, amit OR-nak neveztünk el. Az a1 és az a2 tartalmazza az adatokat és az OR sorral pedig azt mondjuk neki, hogy mit kell kapnunk. Tehát ezzel fixáljuk le a szabályokat. Ezen adatok összeségét elnevezzük `or.data`-nak amit a `data.frame()` függvénnyel oldhatunk meg. Itt kell megadni paraméterként az adatokat. Ezután deklaráljuk az `nn.or` változót is, amihez a `neuralnet()` függvényt használjuk. Az `OR=a1+a2` azt jelenti, hogy az OR a kimenet, majd kötőjellel elválasztjuk a bemenettől és a bemeneteket vessző helyett `+` jellel adjuk hozzá. A `hidden=0` pedig azt jelenti, hogy itt most nem lesz rejtett réteg. Majd a `plot()` függvénnyel írjuk ki az eredményt.

```
a1 <- c(0,1,0,1)
a2 <- c(0,0,1,1)
OR <- c(0,1,1,1)

or.data <- data.frame(a1, a2, OR)
nn.or <- neuralnet(OR~a1+a2, or.data, hidden=0, linear.output=FALSE, ←
 stepmax = 1e+07, threshold = 0.000001)

plot(nn.or)
```

Lényegében ezeket kell majd bemásolni az R szimulációba és megjelenik az alábbi ábra a képernyőn. Itt látható, hogy az ERROR: 2e-06, ami azt jelenti, hogy a számítási hibahatár 0.0000002, ami nagyon kicsi, így ez az érték elfogadható.



4.3. ábra. Neurális OR eredménye

A következő példánál már egy új sort adunk hozzá (AND) és ugyanúgy futtatjuk a programot, ahogy a fenti kódcsipetnél csináltuk. Itt az OR és az AND két külön adat lesz, tehát a program nem veszi őket figyelembe számításnál, csak az a1 és az a2 adataival számol.

```
a1 <- c(0, 1, 0, 1)
a2 <- c(0, 0, 1, 1)
OR <- c(0, 1, 1, 1)
```

```
AND <- c(0,0,0,1)

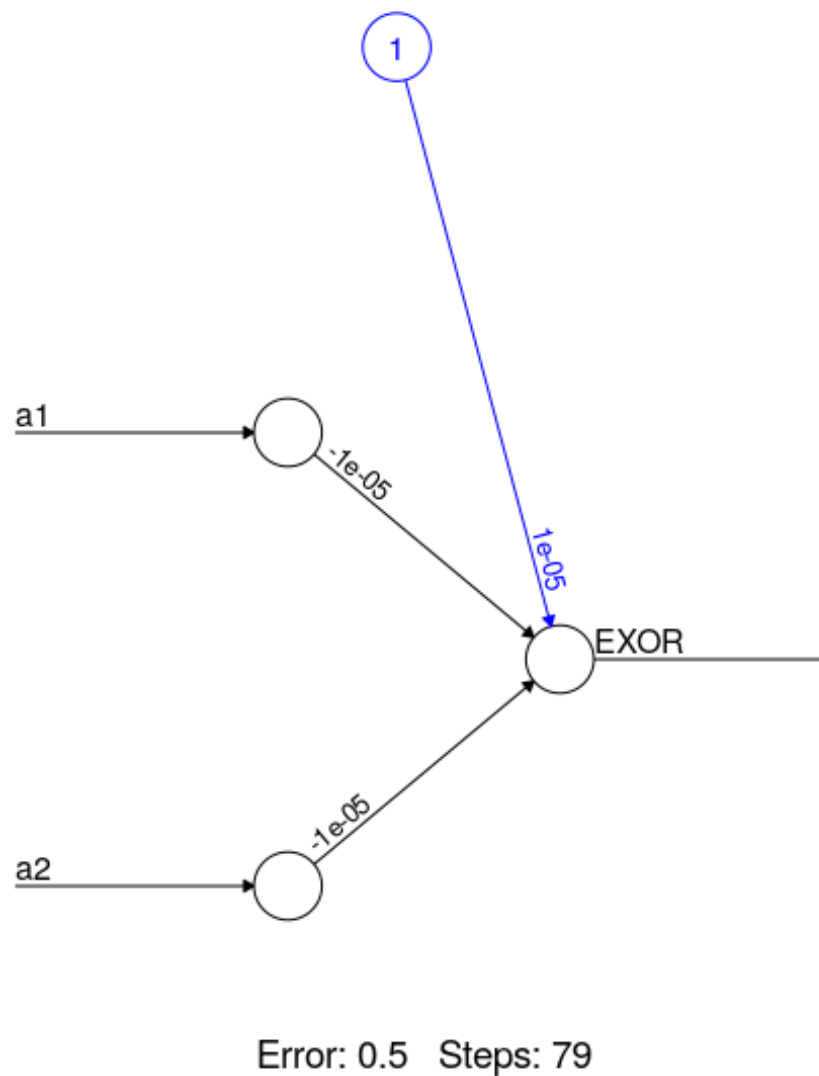
orand.data <- data.frame(a1, a2, OR, AND)

nn.orand <- neuralnet(OR+AND~a1+a2, orand.data, hidden=0, linear.output= <-
 FALSE, stepmax = 1e+07, threshold = 0.000001)

plot(nn.orand)

compute(nn.orand, orand.data[,1:2])
```

És most jön az EXOR. Ha a kódcsipetet bemásoljuk az R szimulációba, akkor az alábbiakat kapjuk.



4.4. ábra. Neurális OR, AND eredménye

Figyeljük meg a hibahatárt. A 0.5-es hiba nagyon nagy, a program nem tud egyértelmű választ adni, ezért szükségünk lesz töbrétegű neuronokra a probléma megoldására, amit az alábbi kódnál mutatom be:

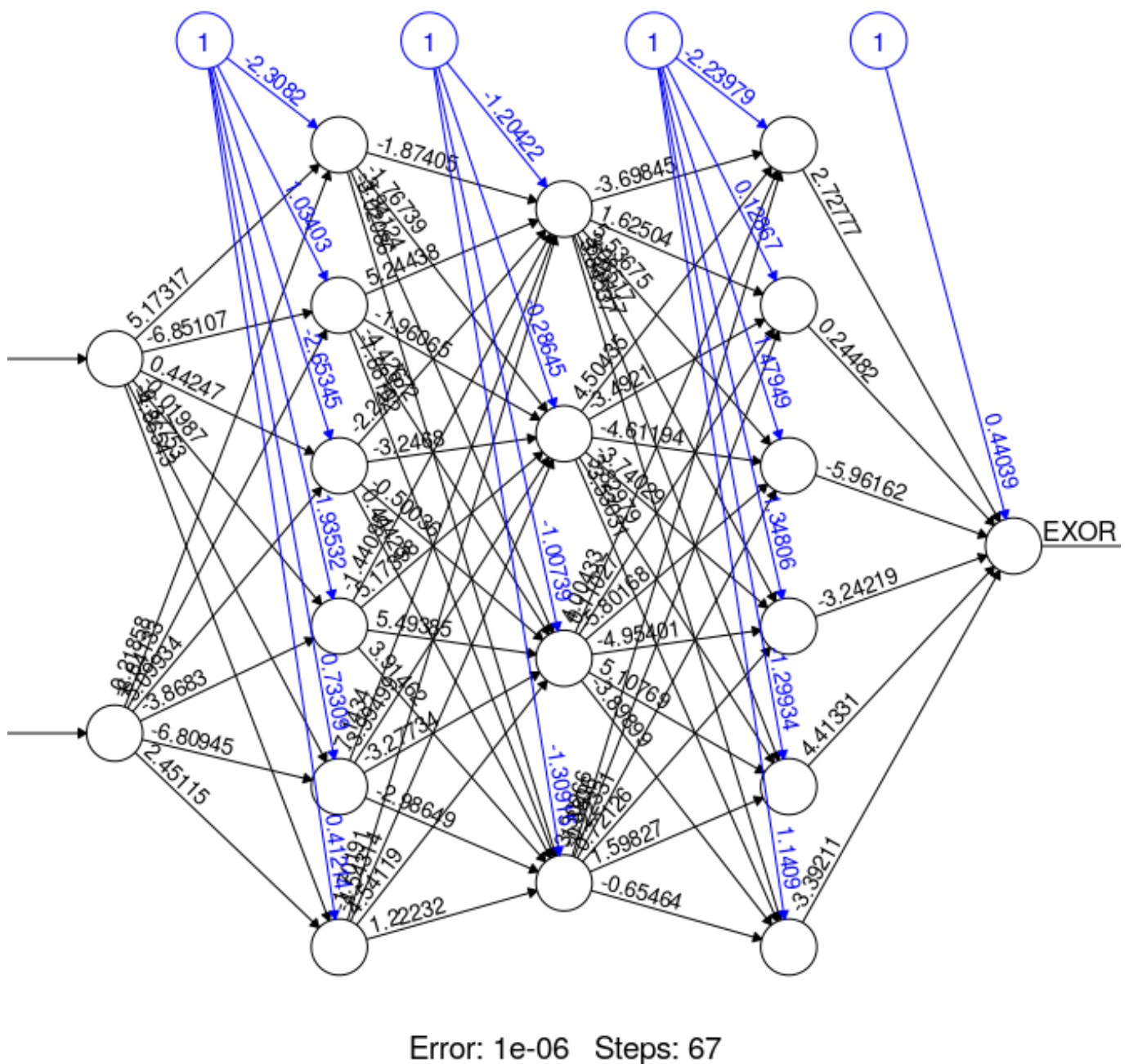
```
a1 <- c(0,1,0,1)
a2 <- c(0,0,1,1)
EXOR <- c(0,1,1,0)

exor.data <- data.frame(a1, a2, EXOR)

nn.exor <- neuralnet(EXOR~a1+a2, exor.data, hidden=c(6, 4, 6), linear. <-
 output=FALSE, stepmax = 1e+07, threshold = 0.000001)
plot(nn.exor)

compute(nn.exor, exor.data[,1:2])
```

Az előző kódrészlethez képest annyi a változás, hogy a `hidden=c(6, 4, 6)`-nál hozzáírtunk 6 neuront, tartalmaz további 4 neuront és megint 6-ot. Ezzel a töbrétegű neuronok hozzáadásával a hibahatár 0.000001 lesz. További különbség az eddigi ábrához képest, hogy a 2 input és az output között megjelennek a neuronok, azaz 6, 4 és 6 kör lesz oszloponként.



4.5. ábra. Neurális EXOR eredménye

## 4.6. Hiba-visszaterjesztéses perceptron

C++

Megoldás videó: <https://youtu.be/XpBnR31BRJY>

Megoldás forrása: <https://github.com/nbatfai/nahshon/blob/master/ql.hpp#L64> [https://gitlab.com/davidhalasz/-/tree/master/attention\\_raising/Source/caesar/perceptron](https://gitlab.com/davidhalasz/-/tree/master/attention_raising/Source/caesar/perceptron)

Ez a program több fájlból áll össze, de mi most csak a main.cpp-t vizsgáljuk meg. Onnan lehet tudni, hogy más programot is használunk, hogy headerben hozzáadjuk őket ( `#include "ml.hpp"` és `#include <png++/png.hpp>` ) A `main()` függvényen belül a `png::` névterű sorban mondjuk meg, hogy mi legyen az importált kép amit szeretnénk beadni a tanulo programnak. A `int size` változóban deklaráljuk, hogy mennyi a kép szélessége és hosszúsága pixelben megadva. Ezek elvégzéséhez szüksége lesz a programnak a headerben megadott `png++/png.hpp` fájlra.

```
png::image<png::rgb_pixel> png_image(argv[1]);

int size = png_image.get_width() * png_image.get_height();

Perceptron *p = new Perceptron(3, size, 256, 1);
double *image = new double[size];
```

A `Perceptron *p` változóban újperceptront hozunk létre. Ez a függvény az `ml.hpp`-ben van felépítve. A függvény 4 paramétert vár: a 3-as szám a layerek száma lesz, a `size` azt jelenti, hogy az első rétegre `size` darab neuront akarunk, a második réteg 256 legyen, a harmadik pedig legyen 1. Alatta az `image` mutató "size" méretű tömb lesz, de egyelőre itt csak a hely lesz neki foglalva, amit majd a `for` ciklusban feltöltjük őket.

```
for (int i = 0; i < png_image.get_width(); ++i)
 for (int j = 0; j < png_image.get_height(); ++j)
 image[i * png_image.get_width() + j] = png_image[i][j].red;

double value = (*p)(image);

std::cout << value << std::endl;

delete p;
delete[] image;
```

Az első `for` ciklussal végigmegyek a kép szélességén, a második ciklussal pedig a magasságon. Az `image` tömböt feltöltjük a `red` componensekkel. Majd a `delete` függvénnyel töröljük a `Perceptron* p`-t mivel előzőleg foglaltunk neki memóriát, továbbá az `image` tömböt is a memóriából.



## 5. fejezet

# Helló, Mandelbrot!

### 5.1. A Mandelbrot halmaz

A mandelbrot-halmaz

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/blob/master/attention\\_raising/Source/mandelpngt.c++](https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/mandelpngt.c++)

A forrásban megfigyelhető, hogy az eddigiekhez képest két új könyvtárat is használunk a headrünkben. Az egyik a `#include "png++/png.hpp"`, amivel png kép készíthető el, a másik könyvtár pedig a `#include < sys/times.h >`, amivel mérhetjük az időt, jelen esetben azt, hogy meddig tart a amíg a program elkészíti a képet. Ez utóbbit kihagyhatjuk, ez csak akkor érdekes, ha meg szeretnénk vizsgálni, hogy mennyi idő alatt készíti el a képet a processzor és mennyi ideig a videokártya.

A main függvény 2 argumentumot tartalmaz, amit az if függvénnyel ellenőrizünk le, azaz, ha a terminálban megadott paraméterek száma nem egyenlő a kettővel, akkor kiíratjuk a kép elkészítéséhez szükséges parancssor használat módját és egy hibaüzenetet. Az első paraméternek a program nevének, a másodiknak pedig az elmenteni kívánt png típusú fájl nevének kell lennie.

```
int
main (int argc, char *argv[])
{
 // Mérünk időt (PP 64)
 clock_t delta = clock ();
 // Mérünk időt (PP 66)
 struct tms tmsbuf1, tmsbuf2;
 times (&tmsbuf1);

 if (argc != 2)
 {
 std::cout << "Hasznalat: ./mandelpng fajlnev";
 return -1;
 }

}
```

Következő lépésként deklaráljuk a kép számításához szükséges adatokat (a, b, c, d, szélesség és magasság). Amit a következő négy sorban használjuk fel. A `png::image < png::rgb_pixel > kep (szélesség, magasság);` kódsorral létrehozunk egy 600x600 pixel méretű üres képet, továbbá deklaráljuk a dx és a dy változókat, melyek értékei az x és y tengely lépésközeit adja meg.

```
double a = -2.0, b = .7, c = -1.35, d = 1.35;
int szélesség = 600, magasság = 600, iteraciosHatar = 32000;

png::image < png::rgb_pixel > kep (szélesség, magasság);

double dx = (b - a) / szélesség;
double dy = (d - c) / magasság;
double reC, imC, reZ, imZ, ujreZ, ujimZ;

}
```

Ezt követően két for ciklussal zongorázzuk végig az adatokat, melyek addig tartanak, amennyi a szélesség és a magasság mérete. Miközben egyenként végigmegyünk a komplex síkon, a while ciklussal ellenőrizzük, hogy a a reZ és a imZ négyzeteinek összege kisebb-e, mint 4 ÉS hogy az iteráció nem éri-e el az iterációs határt. Ha a feltétel teljesül, akkor az adotrácspont nem lesz a Mandelbrot-halmaz eleme és számítással új értékeket rendelünk a reZ és az imZ változókhoz. Ettől a while ciklustól függ a pixelek kiíratása, azaz, ha a feltétel teljesül, akkor nem lesz pixel, ellenkező esetben előhívjuk a `kep.set_pixel()` függvényt a fekete képpont rajzolásához, ahol a k lesz a sor, a j pedig az oszlop.

```
for (int j = 0; j < magasság; ++j)
{
 for (int k = 0; k < szélesség; ++k)
 {
 reC = a + k * dx;
 imC = d - j * dy;
 reZ = 0;
 imZ = 0;
 iteracio = 0;

 while (reZ * reZ + imZ * imZ < 4 && iteracio < iteraciosHatar)
 {
 ujreZ = reZ * reZ - imZ * imZ + reC;
 ujimZ = 2 * reZ * imZ + imC;
 reZ = ujreZ;
 imZ = ujimZ;

 ++iteracio;
 }

 kep.set_pixel (k, j,
 png::rgb_pixel (255 -
 (255 * iteracio) / iteraciosHatar,
 255 -
 (255 * iteracio) / iteraciosHatar,
 255 -
```

```
 (255 * iteracio) / iteraciosHatar));
 }
 std::cout << "." << std::flush;
}
}
```

A fenti képet végül lementjük abba a fájlba, amit még a program elindítása során adtuk meg és kiíratjuk, hogy a kép mentése elkészült. Továbbá a kép elkészítésének idejét is kiíratjuk, amihez előbb számításokat kell végeznünk.

```
kep.write (argv[1]);
std::cout << argv[1] << " mentve" << std::endl;

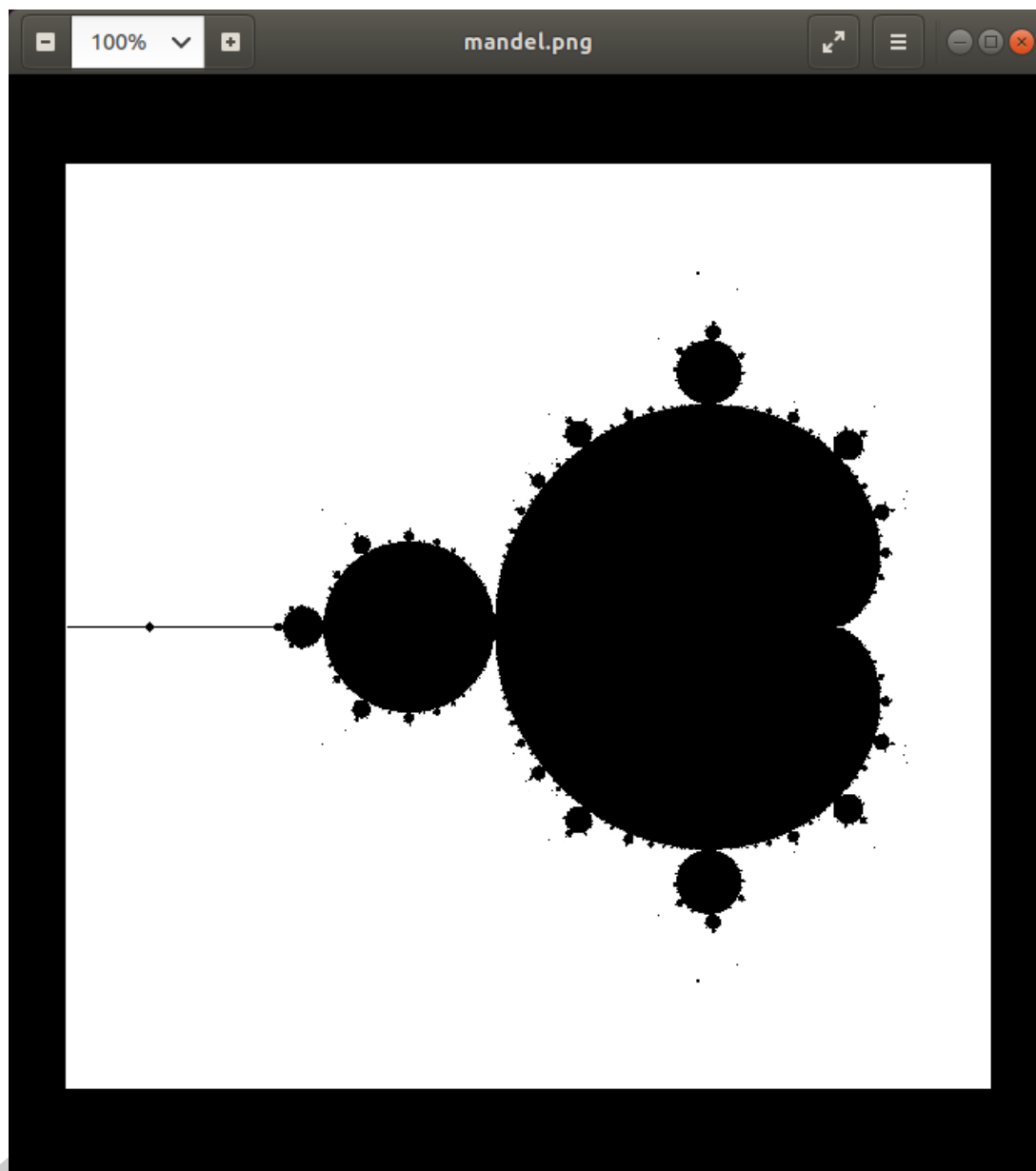
times (&tmsbuf2);
std::cout << tmsbuf2.tms_utime - tmsbuf1.tms_utime
 + tmsbuf2.tms_stime - tmsbuf1.tms_stime << std::endl;

delta = clock () - delta;
std::cout << (double) delta / CLOCKS_PER_SEC << " sec" << std::endl;
}
```

A program elindítása és a végeredmény az alábbi két képen látható:

```
dave@dave-K501UB:~/gyakorlas$ g++ mandelpngt.c++ -lpng16 -o3 -o mandelpngt
dave@dave-K501UB:~/gyakorlas$./mandelpngt mandel.png
2182
21.8237 sec
mandel.png mentve
dave@dave-K501UB:~/gyakorlas$
```

5.1. ábra. A mandelpngt.c++ elindítása



5.2. ábra. A mandel.png eredménye

## 5.2. A Mandelbrot halmaz a `std::complex` osztállyal

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/blob/master/attention\\_raising/Source/mandelbrot/3.1.2.cpp](https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/mandelbrot/3.1.2.cpp)

Az `std::complex` standard könyvtár segítségével a komplex számokat (azaz a valós és imaginárius egységeket) a `complex()` függvénnyel egy helyen kezelhetjük, anélkül, hogy további változókat hoznánk létre. Most az előző Mandelbrot programot készítjük el a `complex` könyvtár használatával. Az időmérést ebből a programból most kihagyjuk.

A headerbe beillesztjük a `<complex>` könyvtárat. A `main` függvényen belül megadjuk a szélességet és a

hosszúságot is, jelen esetben most egy fullhd értékeivel megegyező méretet adtunk meg. Az if() függvényben megvizsgáljuk, hogy a program elindításához szükséges parancssor paramétereinek száma egyenlő-e 9-cel, ha ige, akkor a változókhoz értékeket rendelünk hozzá. Az atoi() függvény azt csinálja, hogy a bevitt adatot integer típusú értékke alakítja, míg az atof() függvény pedig double lesz. Ellenkező esetben, ha a feltétel nem teljesül, akkor kiíratjuk a program helyes elindításához szükséges módot, továbbá egy hibaüzenetet.

```
#include <iostream>
#include "png++/png.hpp"
#include <complex>

int
main (int argc, char *argv[])
{

 int szelesseg = 1920;
 int magassag = 1080;
 int iteraciosHatar = 255;
 double a = -1.9;
 double b = 0.7;
 double c = -1.3;
 double d = 1.3;

 if (argc == 9)
 {
 szelesseg = atoi (argv[2]);
 magassag = atoi (argv[3]);
 iteraciosHatar = atoi (argv[4]);
 a = atof (argv[5]);
 b = atof (argv[6]);
 c = atof (argv[7]);
 d = atof (argv[8]);
 }
 else
 {
 std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg magassag n a b c d ←"
 << std::endl;
 return -1;
 }
}
```

További különbség az előző programunkhoz képest, hogy a második for cikluson belül használjuk az std::complex osztályt. Jellemzője, hogy double típusú paraméterek lesznek és két részből áll: egy valós (reC) és egy imaginárius (imC) egységből. Ez lesz a c változó. Ugyanígy adjuk meg a z\_n változót is, csak itt nullákat adunk meg, amik majd a while ciklusban fog változni, ha z\_n értéke kisebb mint 4 és az iteráció kisebb mint az iterációs határ. Ha azadott érték eleme a Mandelbrot-Halmaznak, akkor pixeleket rajzolunk, továbbá kiíratjuk, hogy hány százaléknál jár a kép feldolgozása, majd az előző programból ismert módon mentünk.

```
png::image < png::rgb_pixel > kep (szelesseg, magassag);
```

```
double dx = (b - a) / szelesseg;
double dy = (d - c) / magassag;
double reC, imC, reZ, imZ;
int iteracio = 0;

std::cout << "Szamitas\n";

for (int j = 0; j < magassag; ++j)
{
 for (int k = 0; k < szelesseg; ++k)
 {
 reC = a + k * dx;
 imC = d - j * dy;
 std::complex<double> c (reC, imC);

 std::complex<double> z_n (0, 0);
 iteracio = 0;

 while (std::abs (z_n) < 4 && iteracio < iteraciosHatar)
 {
 z_n = z_n * z_n + c;

 ++iteracio;
 }

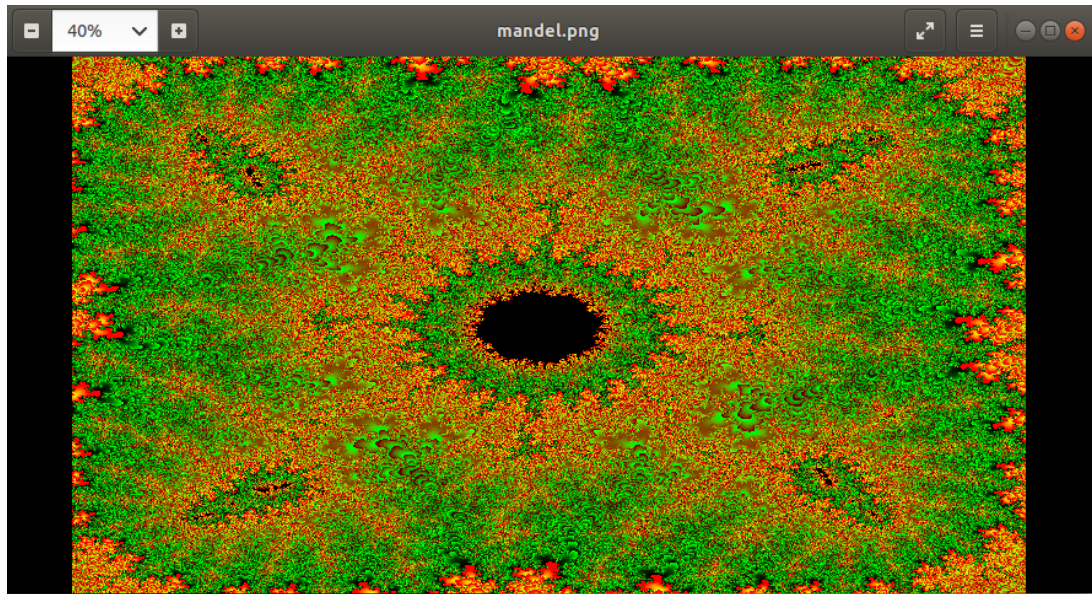
 kep.set_pixel (k, j,
 png::rgb_pixel (iteracio%255, (iteracio*iteracio)%255, 0));
 }

 int szazalek = (double) j / (double) magassag * 100.0;
 std::cout << "\r" << szazalek << "%" << std::flush;
}
kep.write (argv[1]);
std::cout << "\r" << argv[1] << " mentve." << std::endl;
}
```

A program elindítása után egy nem egy fekete-fehér képet, hanem sokkal színebb eredményt kapunk:

```
dave@dave-K501UB:~/gyakorlas$ g++ 3.1.2.cpp -lpng -O3 -o 3.1.2
dave@dave-K501UB:~/gyakorlas$./3.1.2 mandel.png 1920 1080 2040 -0.01947381057309366392260585598705802112818 -0.01947381057254134184564264842
26540196687 0.7985057569338268601555341774655971676111 0.798505756934379196110285192844457924366
Szamitas
mandel.png mentve.
```

5.3. ábra. A 3.1.2.cpp rogram elindítása



5.4. ábra. A 3.1.2.cpp eredménye

## 5.3. Biomorfok

Megoldás videó: <https://youtu.be/IJMbgRzY76E>

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/blob/master/attention\\_raising/Biomorf/3.1.3.cpp](https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Biomorf/3.1.3.cpp)

Vmi a Julia halmazról. Az előző feladathoz képest az a különbség, hogy itt 9 helyett 12 paramétert kell megadni a program elindítása során. Ugyanis itt a valós és az imaginárius egységek továbbá az  $R$  is állandó érték lesz.

```
#include <iostream>
#include "png++/png.hpp"
#include <complex>

int
main (int argc, char *argv[])
{

 int szelesseg = 1920;
 int magassag = 1080;
 int iteraciosHatar = 255;
 double xmin = -1.9;
 double xmax = 0.7;
 double ymin = -1.3;
 double ymax = 1.3;
 double reC = .285, imC = 0;
 double R = 10.0;

 if (argc == 12)
```

```

{
 szelesseg = atoi (argv[2]);
 magassag = atoi (argv[3]);
 iteraciosHatar = atoi (argv[4]);
 xmin = atof (argv[5]);
 xmax = atof (argv[6]);
 ymin = atof (argv[7]);
 ymax = atof (argv[8]);
 reC = atof (argv[9]);
 imC = atof (argv[10]);
 R = atof (argv[11]);

}
else
{
 std::cout << "Hasznalat: ./3.1.3 fajlnev szelesseg magassag n a b c ↔
 d reC imC R" << std::endl;
 return -1;
}

png::image < png::rgb_pixel > kep (szelesseg, magassag);

double dx = (xmax - xmin) / szelesseg;
double dy = (ymax - ymin) / magassag;

std::complex<double> cc (reC, imC);

std::cout << "Szamitas\n";
}

```

A másik lényeges különbség a for ciklusban figyelhető meg. Itt nincs while ciklus, helyette for-t használunk. Az első két ciklussal végigmegyünk a rácson, a harmadik pedig addig fog futni, amíg el nem éri az iterációs határt. Az utóbbin belül a  $z_n$  értéke az aktuális  $z_n$  változó értékének 3. hatványa lesz, amit a `std::pow()` osztállyal számítjuk ki, majd ezt az értéket vizsgáljuk meg egy `if()` függvénnyel. Az `std::real()` és azt `std::imag()` értelemszerűen azt jelenti, hogy az adott  $z_n$  változó valós és imaginárius értékét vegye figyelembe. Tehát ha a valós szám vagy az imaginárius szám értéke nagyobb, mint  $R$  akkor a feltétel teljesül és az iteracio értéke az adott  $i$  változó értékével lesz megegyező. Ezután elkészítjük a Julia-Halmazos képet.

```

for (int y = 0; y < magassag; ++y)
{
 for (int x = 0; x < szelesseg; ++x)
 {
 double reZ = xmin + x * dx;
 double imZ = ymax - y * dy;
 std::complex<double> z_n (reZ, imZ);

 int iteracio = 0;
 for (int i=0; i < iteraciosHatar; ++i)
 {

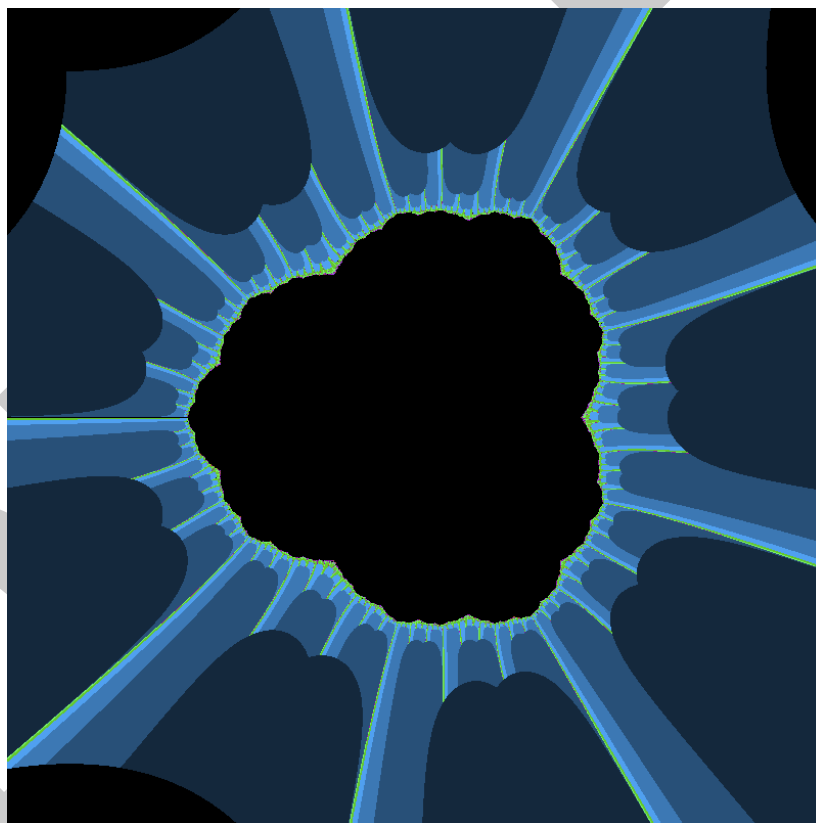
```



```
 z_n = std::pow(z_n, 3) + cc;
 if(std::real (z_n) > R || std::imag (z_n) > R)
 {
 iteracio = i;
 break;
 }
 }
 kep.set_pixel (x, y,
 png::rgb_pixel ((iteracio*20)%255, (iteracio*40)%255, (←
 iteracio*60)%255));
}

int szazalek = (double) y / (double) magassag * 100.0;
std::cout << "\r" << szazalek << "%" << std::flush;
}
}
```

Ezután elkészítjük a Juliaa-Halmazos képet:



5.5. ábra. Biomorf képe

## 5.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/blob/master/attention\\_raising/CUDA/mandelpngc\\_60x60](https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/CUDA/mandelpngc_60x60)

A CUDA az NVIDIA által fejlesztett programozási környezet, amely az NVIDIA grafikus processzorainak párhuzamos programozására használható. Legfontosabb célja, hogy az adott feladatokat a lehető legrövidebb idő alatt oldja meg. Amíg a CPU-k csak néhány szál tudnak egymástól függetlenül futtatni, addig a GPU-k sokkal nagyobb számú feladatokat is képesek elvégezni, és a CUDA ezt az előnyt használja ki, nem csak képalkotási feladatoknál, hanem akár általános célú programok megvalósítása során is felhasználhatjuk.

Jelen példánkban a .cu fájl típusú programot fogjuk használni. Ehhez előzetesen szükségünk lesz az NVIDIA cuda toolkit programot telepítenünk, hogy tudjuk használni az `nvcc` parancsot a program fordítása során. A programunk az első feladatban megismert alkalmazás továbbfejlesztése. A legszembetűnőbb változtatás a programunk elején láthatóak, amelyek mindig valamilyen függvény minősítővel kezdődnek. Ilyen függvény minősítők lehetnek például: `__global__`, `__device__` `__host__` és a `__noinline__`, `__forceinline__`. Jelen esetünkben van egy `__device__` függvényünk. Ez a GPU-n futó függvény kódja, ami csak a GPU kódból hívható. Ide rakjuk be az első feladtból megismert változókkal, számításokkal és a rács csomópontjaival kapcsolatos kódcsipetet. A magasság és a meret most 600x600 pixel lesz.

```
__device__ int
mandel (int k, int j)
{

 // számítás adatai
 float a = -2.0, b = .7, c = -1.35, d = 1.35;
 int szelesseg = MERET, magassag = MERET, iteraciosHatar = ITER_HAT;

 // a számítás
 float dx = (b - a) / szelesseg;
 float dy = (d - c) / magassag;
 float reC, imC, reZ, imZ, ujreZ, ujimZ;
 // Hány iterációt csináltunk?
 int iteracio = 0;

 // c = (reC, imC) a rács csomópontjainak
 // megfelelő komplex szám
 reC = a + k * dx;
 imC = d - j * dy;
 // z_0 = 0 = (reZ, imZ)
 reZ = 0.0;
 imZ = 0.0;
 iteracio = 0;

 while (reZ * reZ + imZ * imZ < 4 && iteracio < iteraciosHatar)
 {
 // z_{n+1} = z_n * z_n + c
 ujreZ = reZ * reZ - imZ * imZ + reC;
 ujimZ = 2 * reZ * imZ + imC;
 reZ = ujreZ;
 imZ = ujimZ;

 ++iteracio;
 }
}
```

```
 }
 return iteracio;
}
```

Továbbá van még két `__global__` függvényminősítőnk, és mindkettő a `mandelkernel()` függvényhez kapcsolódik. Ez egy kernel függvény lesz, ami azt jelenti, hogy a GPU-n futó kódból, azaz a gazda kódból hívható. A `blockIdx` a CUDA saját beépített változója, ami `uint3` típusú, a futó blokk számát adja meg. A `threadIdx` is egy `uint3` típusú változó, csak ez a futó szál számát fogja megadni.

```
/*
__global__ void
mandelkernel (int *kepadat)
{

 int j = blockIdx.x;
 int k = blockIdx.y;

 kepadat[j + k * MERET] = mandel (j, k);

}
*/

__global__ void
mandelkernel (int *kepadat)
{

 int tj = threadIdx.x;
 int tk = threadIdx.y;

 int j = blockIdx.x * 10 + tj;
 int k = blockIdx.y * 10 + tk;

 kepadat[j + k * MERET] = mandel (j, k);

}
```

Ez a `cudamandel()` függvény két paramétert vár: integer típusú `kepadat` tömb értékeit, ami értéke most egyenként 600 lesz. A `cudaMalloc` működése hasonló a C++ `malloc()` függvényéhez, amit szintén memó-riafoglaláshoz használunk. A `dim3` típusú `grid` blokk mérete 60x60 lesz, mivel a méretet elosztjuk 10-el és blokkonként 100 szállal fog dolgozni, ezt a `tgrid()` függvénnyel adtuk meg.

```
void
cudamandel (int kepadat[MERET][MERET])
{

 int *device_kepadat;
 cudaMalloc ((void **) &device_kepadat, MERET * MERET * sizeof (int));
```

```
dim3 grid (MERET / 10, MERET / 10);
dim3 tgrid (10, 10);
mandelkernel <<< grid, tgrid >>> (device_kepadat);

cudaMemcpy (kepadat, device_kepadat,
 MERET * MERET * sizeof (int), cudaMemcpyDeviceToHost);
cudaFree (device_kepadat);

}
```

A programunk main() részét nem tartom szükségesnek bemutatni, ugyanis ha megvizsgáljuk, akkor nem találunk semmi újdonságot az eddigi programunkhoz képest. Van benne egy cudamandel() függvény, ami megfogja kapni a fenti adatokat, így fogjuk elkészíteni a képet.

```
int
main (int argc, char *argv[])
{

 // MÉRÜNK IDŐT (PP 64)
 clock_t delta = clock ();
 // MÉRÜNK IDŐT (PP 66)
 struct tms tmsbuf1, tmsbuf2;
 times (&tmsbuf1);

 if (argc != 2)
 {
 std::cout << "Hasznalat: ./mandelpngc fajlnev";
 return -1;
 }

 int kepadat[MERET][MERET];

 cudamandel (kepadat);

 png::image < png::rgb_pixel > kep (MERET, MERET);

 for (int j = 0; j < MERET; ++j)
 {
 //sor = j;
 for (int k = 0; k < MERET; ++k)
 {
 kep.set_pixel (k, j,
 png::rgb_pixel (255 -
 (255 * kepadat[j][k]) / ITER_HAT,
 255 -
 (255 * kepadat[j][k]) / ITER_HAT,
 255 -
 (255 * kepadat[j][k]) / ITER_HAT));
 }
 }
}
```

```
 }
 kep.write (argv[1]);

 std::cout << argv[1] << " mentve" << std::endl;

 times (&tmsbuf2);
 std::cout << tmsbuf2.tms_utime - tmsbuf1.tms_utime
 + tmsbuf2.tms_stime - tmsbuf1.tms_stime << std::endl;

 delta = clock () - delta;
 std::cout << (float) delta / CLOCKS_PER_SEC << " sec" << std::endl;
}

}
```

## 5.5. Mandelbrot nagyító és utazó C++ nyelven

Építs GUI-t a Mandelbrot algoritmusra, lehessen egérrel nagyítani egy területet, illetve egy pontot egérrel kiválasztva vizualizálja onnan a komplex iteráció bejárta  $z_n$  komplex számokat!

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/tree/master/attention\\_raising/Source/mandelbrot/nagyitas](https://gitlab.com/davidhalasz/bhax/tree/master/attention_raising/Source/mandelbrot/nagyitas)



### Megjegyzés

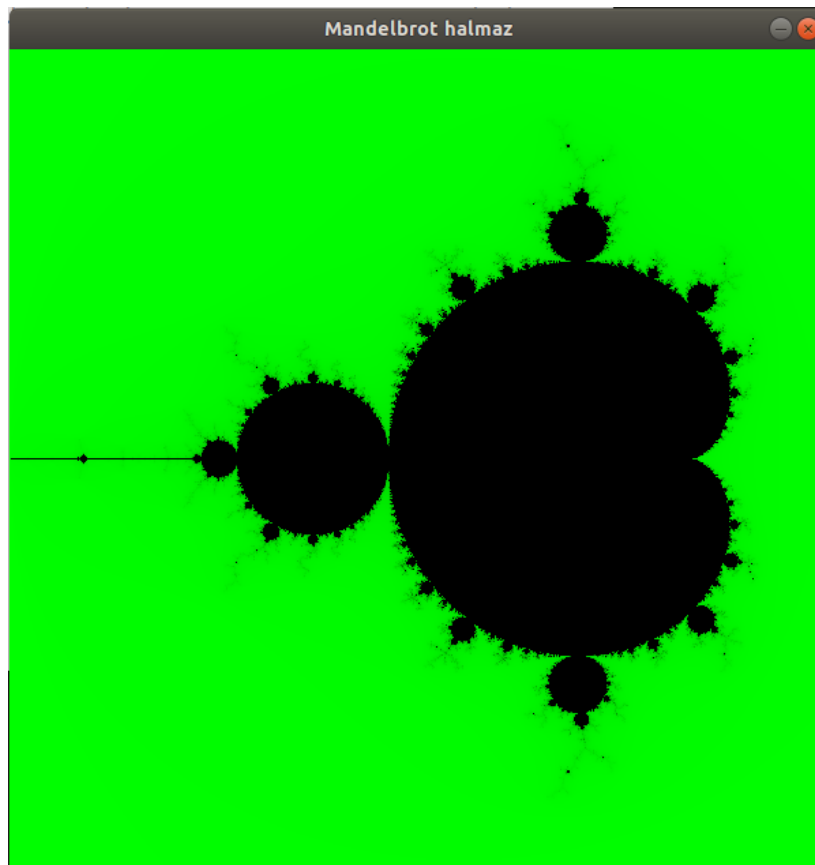
A feladat megoldásában tutorként részt vett: Fürjes-Beke Péter

Mielőtt elkezdenénk a programot, először szükséges telepíteni a libqt4-dev csomagot, mert enélkül nem fog elindulni a program. A mappában egy helyen kell szerepelnie az alábbi négy fájlnek, ami a képen is látható, miután elindítottam az `ls -l` parancsot. Továbbá a `qmake -project` létrehoz egy `.pro` végződésű fájlt, a fájl neve megegyezik az aktuális mappa nevével. Miután ez megtörtént, ezt a `.pro` végződésű fájl-ba bele kell másolni ezt a sort: `QT += widgets`. Ezután lehet a parancssorba beírni, hogy `qmake *.pro`

```
dave@dave-K501UB:~/gyakorlas/nagyitas$ ls -l
összesen 24
-rw-rw-r-- 1 dave dave 2634 márc 23 22:22 frakablak.cpp
-rw-rw-r-- 1 dave dave 1348 márc 23 22:22 frakablak.h
-rw-rw-r-- 1 dave dave 2609 márc 23 22:22 frakszal.cpp
-rw-rw-r-- 1 dave dave 866 márc 23 22:22 frakszal.h
-rw-rw-r-- 1 dave dave 600 márc 23 22:22 main.cpp
-rw-r--r-- 1 dave dave 976 márc 25 20:25 nagyitas.pro
dave@dave-K501UB:~/gyakorlas/nagyitas$ qmake -project
dave@dave-K501UB:~/gyakorlas/nagyitas$ qmake *.pro
```

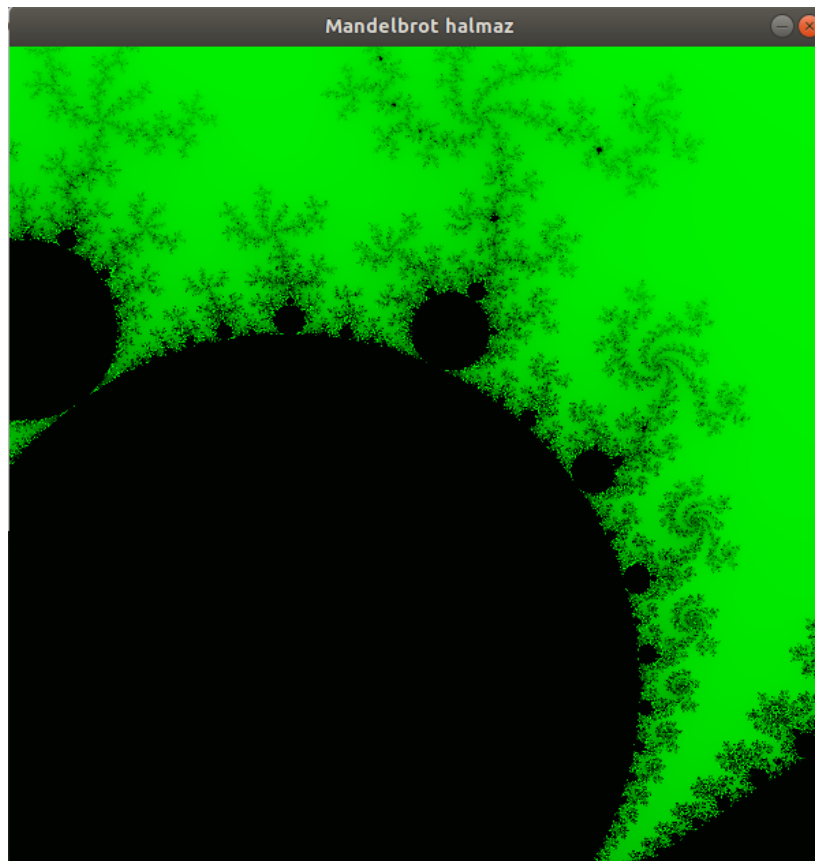
5.6. ábra. Makefile létrehozása

Ez a parancs létrehozza a Makefile-t, amit a make parancs használatával egy bináris fájlt hozunk létre, aminek a neve megegyezik az aktuális mappánk nevével. Ezt kell elindítanunk, ugyanúgy ahogy bármely más programot szoktunk. Haminden jól megy akkor az alábbi Mandelbrot-halmazt kell kapnunk:



5.7. ábra. Mandelbrot halmaz alaphelyzetben

Nagyítani úgy tudunk, hogy az egérrel kijelölünk egy részt, majd a program automatikusan oda zoomol. Ha a kép életlennek tűnik, akkor meg kell nyomnunk az "n" billentyűt, ugyanis ilyenkor a program számítást végez és élesíti a képet.



5.8. ábra. Mandelbrot halmaz nagyítva

## 5.6. Mandelbrot nagyító és utazó Java nyelven

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/tree/master/attention\\_raising/Source/mandelbrot/java](https://gitlab.com/davidhalasz/bhax/tree/master/attention_raising/Source/mandelbrot/java)

Ebben a feladatban a az előzőt csináljuk meg ugyanazt, csak java-ban. A fordítás a `javac` paranccsal történik, és a program elindításához pedig a `java`-t használjuk. Ezt az alábbi kódsor szemlélteti:

```
javac MandelbrotHalmazNagyító.java
java MandelbrotHalmazNagyító
```

A nagyítást végző kódsorokat `MandelbrotHalmazNagyító.java` programon belül a `MandelbrotHalmazNagyító` osztályban hozzuk létre. A `MouseAdapter` implementálja a `MouseListener`-t. A `MousePressed` osztály a `MouseEvent` paramétereit várja, ami akkor lép működésbe, ha egy pontra történik az egér klikkelés. Ilyenkor az osztályon belül lekéri a nagyítandó kijelölt terület bal felső sarkának helyét. Alaphelyzetben a szélesség (`mx`) és a magasság (`my`) értékei 0, ami megváltozik ha az egérrel kijelölünk egy területet.

```
// Egér kattintó események feldolgozása:
addMouseListener(new java.awt.event.MouseAdapter() {
 // Egér kattintással jelöljük ki a nagyítandó területet
 // bal felső sarkát:
 public void mousePressed(java.awt.event.MouseEvent m) {
```

```
// A nagyítandó kijelölt területet bal felső sarka:
x = m.getX();
y = m.getY();
mx = 0;
my = 0;
repaint();
}
}
```

Ha kijelöltünk egy területet, akkor a `MouseReleased()` osztály elvégz egy számítást, és egy új nagyító objektumot hoz létre.

```
public void mouseReleased(java.awt.event.MouseEvent m) {
 double dx = (MandelbrotHalmazNagyító.this.b
 - MandelbrotHalmazNagyító.this.a)
 /MandelbrotHalmazNagyító.this.szélesség;
 double dy = (MandelbrotHalmazNagyító.this.d
 - MandelbrotHalmazNagyító.this.c)
 /MandelbrotHalmazNagyító.this.magasság;

 new MandelbrotHalmazNagyító(MandelbrotHalmazNagyító.this.a+x*dx,
 MandelbrotHalmazNagyító.this.a+x*dx+mx*dx,
 MandelbrotHalmazNagyító.this.d-y*dy-my*dy,
 MandelbrotHalmazNagyító.this.d-y*dy,
 600,
 MandelbrotHalmazNagyító.this.iterációsHatár);
}
}
```

A `mouseDragged()` osztállyal figyeljük, hogy kijelölt területet. Ekkor kapjuk meg a szélességet és a magasságot értékül, amik rendre az `mx` és az `my` változókbán lesznek eltárolva.

```
addMouseListener(new java.awt.event.MouseMotionAdapter() {
 public void mouseDragged(java.awt.event.MouseEvent m) {
 mx = m.getX() - x;
 my = m.getY() - y;
 repaint();
 }
});
}
```

Itt is lehet élesíteni a képet, ha nagyítás után homályos lesz. Ehhez a `KeyAdaptert()` osztályt hívjuk segítségül, ami a `MandelbrotHalmaz.java` programban található. A `KeyListener()` figyel a billentyűzetleütéseket. Jelen esetben az 's', 'n' és 'm' gombokat figyel. Ha az a kapott billentyűzet (`e.getKeyCode()`) egyenlő az 'n' billentyűzettel (`KeyEvent.VK_N`), akkor újraszámítjuk az adatokat, kivéve, ha már előtte ki volt számítva.

```
...
addKeyListener(new java.awt.event.KeyAdapter() {
 // Az 's', 'n' és 'm' gombok lenyomását figyeljük
 public void keyPressed(java.awt.event.KeyEvent e) {
```



```
 if(e.getKeyCode() == java.awt.event.KeyEvent.VK_S)
 pillanatfelvétel();
 // Az 'n' gomb benyomásával pontosabb számítást végzünk.
 else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_N) {
 if(számításFut == false) {
 MandelbrotHalmaz.this.iterációsHatár += 256;
 // A számítás újra indul:
 számításFut = true;
 new Thread(MandelbrotHalmaz.this).start();
 }
 } else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_M) {
 if(számításFut == false) {
 MandelbrotHalmaz.this.iterációsHatár += 10*256;
 // A számítás újra indul:
 számításFut = true;
 new Thread(MandelbrotHalmaz.this).start();
 }
 }
 }
}
...
}
```

## 6. fejezet

# Helló, Welch!

### 6.1. Első osztályom

Valósítsd meg C++-ban és Java-ban az módosított polártranszformációs algoritmust! A matek háttér teljesen irreleváns, csak annyiban érdekes, hogy az algoritmus egy számítása során két normálist számol ki, az egyiket elspájzolod és egy további logikai taggal az osztályban jelzed, hogy van vagy nincs eltérő kiszámolt szám.

Megoldás C++ forrása: [https://gitlab.com/davidhalasz/bhax/tree/master/attention\\_raising/Source/welch/polargen/cpp](https://gitlab.com/davidhalasz/bhax/tree/master/attention_raising/Source/welch/polargen/cpp)

Megoldás Java forrása: [https://gitlab.com/davidhalasz/bhax/blob/master/attention\\_raising/Source/welch/-polargen/java/PolarGenerator.java](https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/welch/-polargen/java/PolarGenerator.java)

Ebben a feladatban a polártranszformációs algoritmust fogom bemutatni. Először vizsgáljuk meg a C++ programot. Ez a program 3 részből áll: `polargenteszt.cpp`, `polargen.cpp` és `polargen.h`. A `polargen.h` felépítése az alábbi:

```
#ifndef POLARGEN__H
#define POLARGEN__H

#include <cstdlib>
#include <cmath>
#include <ctime>

class PolarGen
{
public:
 PolarGen ()
 {
 nincsTarolt = true;
 std::srand (std::time (NULL));
 }
 ~PolarGen ()
 {
 }
 double kovetkezo ();
};
```

```
private:
 bool nincsTarolt;
 double tarolt;

};

#endif
```

Első lépésként deklaráljuk a `nincsTarolt` változót, aminek értéke igaz lesz. Ezután a véletlenszám-generátor függvényt hívjuk segítségül. Azért használjuk az `srand()` függvényt a `rand()` helyett, mert folyamatosan változó, megjósolhatatlan véletlen számra van szükségünk, ezért a függvénynek paraméterül a `std::time()` függvényt adjuk meg, azaz a számítógépünk órája által jelzett idő bitje lesz. Figyeljük meg, hogy a véletlenszám készítőnk egy `PolarGen` osztályban van felépítve. Így használhatóbb lesz a generátorunk és könnyebb lesz különböző értéktartományokhoz véletlen számokat előállítani. A `nincsTarolt` egy boolean típusú változó lesz, a `tarolt` pedig `double` típusú. Ezt a programban más-hol nem fogjuk tudni megváltoztatni, mivel ez egy `private` osztályban vannak elhelyezve. Van a programban egy `double` `kovetkezo()` függvényünk is, mely a `polargen.cpp`-ben van felépítve:

```
#include "polargen.h"

double
PolarGen::kovetkezo ()
{
 if (nincsTarolt)
 {
 double u1, u2, v1, v2, w;
 do
 {
 u1 = std::rand () / (RAND_MAX + 1.0);
 u2 = std::rand () / (RAND_MAX + 1.0);
 v1 = 2 * u1 - 1;
 v2 = 2 * u2 - 1;
 w = v1 * v1 + v2 * v2;
 }
 while (w > 1);

 double r = std::sqrt ((-2 * std::log (w)) / w);

 tarolt = r * v2;
 nincsTarolt = !nincsTarolt;

 return r * v1;
 }
 else
 {
 nincsTarolt = !nincsTarolt;
 return tarolt;
 }
}
```

```
}
```

Gyakorlatilag ez maga a polártranszformációs függvényünk. A számításokat itt végezzük el. Először is megvizsgáljuk hogy a `nincsTarolt` változó értéke hamis, vagy igaz. Ha igaz, akkor azt jelenti, hogy a `tarolt` változóban el van tárolva a visszaadandó lebegőpontos szám. A matematikai háttér most nem fontos, ezért a polártranszformációs eljárással kapcsolatos kódcsipetet átlépjük.

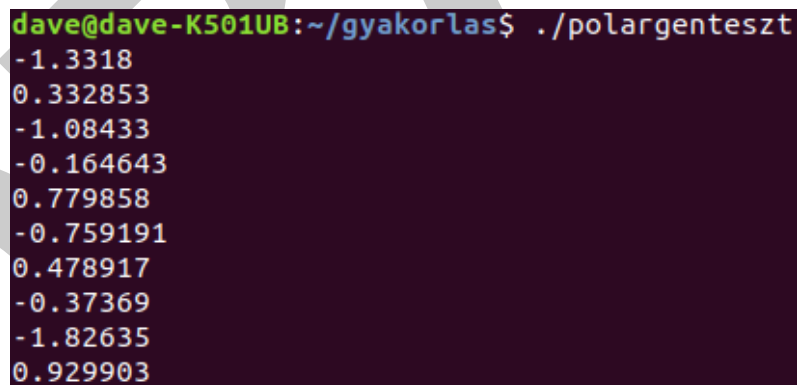
```
#include <iostream>
#include "polargen.h"
#include "polargen.cpp"

int
main (int argc, char **argv)
{
 PolarGen pg;

 for (int i = 0; i < 10; ++i)
 std::cout << pg.kovetkezo () << std::endl;

 return 0;
}
```

A fenti kódcsipet a `polargenteszt.cpp` fájlban található. Ez lesz a fő fájl, azaz majd ezt kell lefordítanunk és futtatni. For ciklussal megadjuk, hogy a program tízszer fusson le és írassa ki a kapott számításokat. Így tehát a program futtatása után megkapjuk a terminálban a következő kimenetet:



```
dave@dave-K501UB:~/gyakorlas$./polargenteszt
-1.3318
0.332853
-1.08433
-0.164643
0.779858
-0.759191
0.478917
-0.37369
-1.82635
0.929903
```

6.1. ábra. A polargenteszt eredménye

Most nézzük meg ugyanezt a programot Java-ban. Ha összehasonlítjuk az előző programmal, akkor láthatjuk, hogy szembetűnő változás nincs. Talán csak annyi, hogy a Java verzió sokkal letisztultabbnak tűnik. A Java alapból osztályokkal dolgozik, tehát itt programozni csak az objektum orientált paradigma mentén lehet.

```
public class PolarGenerator {
 boolean nincsTarolt = true;
```

```
double tárolt;

public PolarGenerator() {
 nincsTárolt = true;
}

public double következő() {
 if(nincsTárolt) {
 double u1, u2, v1, v2, w;
 do {
 u1 = Math.random();
 u2 = Math.random();
 v1 = 2*u1 - 1;
 v2 = 2*u2 - 1;
 w = v1*v1 + v2*v2;
 } while(w > 1);
 double r = Math.sqrt((-2*Math.log(w))/w);
 tárolt = r*v2;
 nincsTárolt = !nincsTárolt;
 return r*v1;
 } else {
 nincsTárolt = !nincsTárolt;
 return tárolt;
 }
}

public static void main(String[] args) {

 PolarGenerator g = new PolarGenerator();

 for(int i=0; i<10; ++i)
 System.out.println(g.következő());
}
```

A programban megfigyelhető következő() függvény megtalálható a Sun programozói által készített Java JDK forrásaiban, pontosabban az src.zip állományában a java/util/Random.java forrásban. Ennek a kódcsipetnek a megoldása nagyon hasonlít a miénkhez, amit az alábbi képen is megfigyelhetjük:

```
public synchronized double nextGaussian() {
 // See Knuth, ACP, Section 3.4.1 Algorithm C.
 if (haveNextNextGaussian) {
 haveNextNextGaussian = false;
 return nextNextGaussian;
 } else {
 double v1, v2, s;
 do {
 v1 = 2 * nextDouble() - 1; // between -1 and 1
 v2 = 2 * nextDouble() - 1; // between -1 and 1
 s = v1 * v1 + v2 * v2;
 } while (s >= 1 || s == 0);
 double multiplier = StrictMath.sqrt(-2 * StrictMath.log(s)/s);
 nextNextGaussian = v2 * multiplier;
 haveNextNextGaussian = true;
 return v1 * multiplier;
 }
}
```

6.2. ábra. A JAVA JDK kódcsipete

A JDK-ban a `haveNextNextGaussian` és a `nextNextGaussian` a fenti `nextGaussian()` függvény felett van definiálva, mégpedig úgy, hogy a változók `private` módban vannak megadva, azaz csak a saját osztálya számára lesz látható, más osztály azonban nem fog tudni hozzáférni az adathoz:

```
private double nextNextGaussian;
private boolean haveNextNextGaussian = false;
```

## 6.2. LZW

Valósítsd meg C-ben az LZW algoritmus fa-építését!

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/blob/master/attention\\_raising/Source/welch/binfa/-lzw.c](https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/welch/binfa/-lzw.c)

Az LZW (Lempel-Ziv-Welch) az LZ77 tömörítőprogram továbbfejlesztése, amit Terry Welch publikált 1984-ben. Ez lényegében egy veszteségmentes tömörítési eljárás. Mivel könnyű implementálni, ezért nagyon elterjedt. Jelen programunk lényege az, hogy egy binárisan megadott karakter-sorozatot részekre tördeljük rekurzíven. Ha egy darabot letördeltünk, akkor utána addig vizsgáljuk a sorozat további részeit, ami még nincs a "szótárunkban", ugyanis ha ilyen találat van, akkor azt szintén letördeljük és felvesszük a szótárunkba. Ha a sorozat végéhez érünk, akkor ebből a részekből egy bináris fa állítható elő. Jelen esetünkben úgy kell elképzelni, hogy a gyökér elemnek van egy 1-es és egy nullás gyermeke, majd ezekhez is fog tartozni további nullás és egyes gyermeke és így tovább.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
```

```
typedef struct binfa
{
 int ertek;
 struct binfa *bal_nulla;
 struct binfa *jobb_egy;
} BINFA, *BINFA_PTR;

BINFA_PTR
uj_elem ()
{
 BINFA_PTR p;

 if ((p = (BINFA_PTR) malloc (sizeof (BINFA))) == NULL)
 {
 perror ("memoria");
 exit (EXIT_FAILURE);
 }
 return p;
}
```

Ennél a kódcsipetnél az `uj_elem()` függvényen belül építjük fel a memóriafoglalási alprogramunkat. A `malloc` függvényről már volt szó, ugye ez fogja számunkra lefoglalni a memóriát, paraméterként a `BINFA` változót adjuk meg. Ezután a `main()` függvényben a gyöker értéket `'/'` karakterre állítjuk be, tehát ez lesz majd a bináris fánk tetején és 0-val fog kezdődni. A While ciklusban megvizsgáljuk, hogy a fa bal oldala tartalmaz-e nullát, ha nem létrehozunk egyet, `NULL` értéket adunk az egyes gyermeknek, és visszakerülünk a gyökérbe. Ezután megint egy `if` függvény jön, ugye mivel `NULL` értéket adtunk az egyes gyermeknek ezért a feltétel igaz lesz, és létrehozuk ez egyes gyermeket is.

```
extern void kiir (BINFA_PTR elem);
extern void ratlag (BINFA_PTR elem);
extern void rszoras (BINFA_PTR elem);
extern void szabadit (BINFA_PTR elem);

int
main (int argc, char **argv)
{
 char b;

 BINFA_PTR gyoker = uj_elem ();
 gyoker->ertek = '/';
 gyoker->bal_nulla = gyoker->jobb_egy = NULL;
 BINFA_PTR fa = gyoker;

 while (read (0, (void *) &b, 1))
 {
 // write (1, &b, 1);
 if (b == '0')
 {
```

```
 if (fa->bal_nulla == NULL)
 {
 fa->bal_nulla = uj_elem ();
 fa->bal_nulla->ertek = 0;
 fa->bal_nulla->bal_nulla = fa->bal_nulla->jobb_egy = NULL;
 fa = gyoker;
 }
 else
 {
 fa = fa->bal_nulla;
 }
}

else
{
 if (fa->jobb_egy == NULL)
 {
 fa->jobb_egy = uj_elem ();
 fa->jobb_egy->ertek = 1;
 fa->jobb_egy->bal_nulla = fa->jobb_egy->jobb_egy = NULL;
 fa = gyoker;
 }
 else
 {
 fa = fa->jobb_egy;
 }
}

printf ("\n");
kiir (gyoker);
```

Az alábbi kódrészletekben csak számításokat végzünk, arra vonatkozóan, hogy mennyi lesz majd az elkészült binfánk mélysége, átlaga stb. Majd a végén kiíratjuk a kiir (BINFA\_PTR elem) függvénnyel a kapott eredményt.

```
extern int max_melyseg, atlagosszeg, melyseg, atlagdb;
extern double szorasosszeg, atlag;

printf ("melyseg=%d\n", max_melyseg-1);

atlag = ((double)atlagosszeg) / atlagdb;

/* Ághosszak szórásának kiszámítása */
atlagosszeg = 0;
melyseg = 0;
atlagdb = 0;
szorasosszeg = 0.0;
```



```
rszoras (gyoker);

double szoras = 0.0;

if (atlagdb - 1 > 0)
 szoras = sqrt(szorasosszeg / (atlagdb - 1));
else
 szoras = sqrt (szorasosszeg);

printf ("atlag=%f\nszoras=%f\n", atlag, szoras);

szabadit (gyoker);
}

int atlagosszeg = 0, melyseg = 0, atlagdb = 0;

void
ratlag (BINFA_PTR fa)
{
 if (fa != NULL)
 {
 ++melyseg;
 ratlag (fa->jobb_egy);
 ratlag (fa->bal_nulla);
 --melyseg;

 if (fa->jobb_egy == NULL && fa->bal_nulla == NULL)
 {
 ++atlagdb;
 atlagosszeg += melyseg;
 }
 }
}

double szorasosszeg = 0.0, atlag = 0.0;

void
rszoras (BINFA_PTR fa)
{
 if (fa != NULL)
 {
 ++melyseg;
 rszoras (fa->jobb_egy);
 rszoras (fa->bal_nulla);
 }
}
```

```
--melyseg;

 if (fa->jobb_egy == NULL && fa->bal_nulla == NULL)
 {

 ++atlagdb;
 szorasosszeg += ((melyseg - atlag) * (melyseg - atlag));

 }

}

}

//static int melyseg = 0;
int max_melyseg = 0;

void
kiir (BINFA_PTR elem)
{
 if (elem != NULL)
 {
 ++melyseg;
 if (melyseg > max_melyseg)
 max_melyseg = melyseg;
 kiir (elem->jobb_egy);
 // ez a postorder bejáráshoz képest
 // 1-el nagyobb mélység, ezért -1
 for (int i = 0; i < melyseg; ++i)
 printf ("---");
 printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem->ertek ↔
 ,
 melyseg-1);
 kiir (elem->bal_nulla);
 --melyseg;
 }
}

void
szabadit (BINFA_PTR elem)
{
 if (elem != NULL)
 {
 szabadit (elem->jobb_egy);
 szabadit (elem->bal_nulla);
 free (elem);
 }
}
```

A program futtatása és a kapott eredményt az alábbi képen láthatjuk. Eszerint a b.txt-ben található 11 karakterből álló bináris sorból végül 6-ot tároltunk el és a fánk mélysége 3 lett.

```
dave@dave-K501UB:~/gyakorlas$ more b.txt
00011101110
dave@dave-K501UB:~/gyakorlas$ gcc lzw.c -o lzw -lm
dave@dave-K501UB:~/gyakorlas$./lzw < b.txt > output.txt
dave@dave-K501UB:~/gyakorlas$ more output.txt

-----1(2)
-----0(3)
-----1(1)
---/(0)
-----1(2)
-----0(1)
-----0(2)
melyseg=3
altag=2.333333
szoras=0.577350
dave@dave-K501UB:~/gyakorlas$
```

6.3. ábra. Az LZW fa C-ben

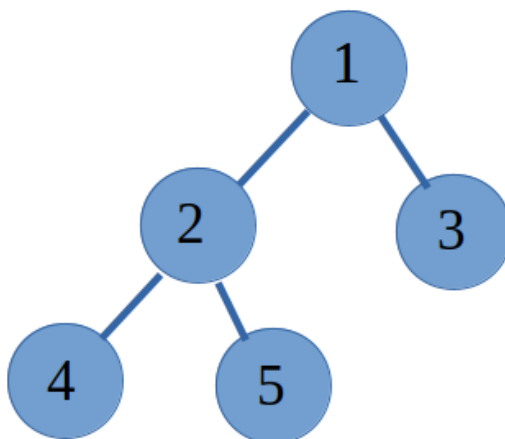
## 6.3. Fabejárás

Járd be az előző (inorder bejárású) fát pre- és posztorder is!

Preorder Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/blob/master/attention\\_raising/Source/welch/binfa/binfa%20c/pre/lzwpre.c](https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/welch/binfa/binfa%20c/pre/lzwpre.c)

Postorder Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/blob/master/attention\\_raising/Source/welch/binfa/binfa%20c/post/lzwpost.c](https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/welch/binfa/binfa%20c/post/lzwpost.c)

Ellentétben a lineáris adatstruktúrákkal, amelyeknek csak egy logikai módja van, a fabejárás történhet pre- és postorder módon. Az alábbi kis ábra és a hozzá tartozó szöveges magyarázat szemlélteti az in-, post és preorder közötti különbséget:



6.4. ábra. Fabejárás

- Inorder esetén a fenti fa a következő módon lesz bejárva: 4 2 5 1 3. Azaz először alulról a bal oldali gyermeket, majd a gyökérelemet, utána annak jobb oldali gyermekét.
- Preorder: 1 2 4 5 3. Tehát először a legelső gyökérelemet, onnan a bal, majd a jobb oldali gyermeket.
- Postorder-nél pedig: 4 5 2 3 1. Alulról a bal oldali gyermeket, jobb oldali gyermeket, majd végül a gyökérelemet.

Most a forráskódból csak a megváltozott kódcsípeteket másolom be, mivel a többi teljesen megegyezik az előző feladat forráskódjával. Először vizsgáljuk meg a preorder esetet. Itt is az előző feladatban használt b.txt fájlban tárolt bináris kódsort használjuk tesztelésre.

```
...
void
kiir (BINFA_PTR elem)
{
 if (elem != NULL)
 {
 ++melyseg;
 if (melyseg > max_melyseg)
 max_melyseg = melyseg;
 for (int i = 0; i < melyseg; ++i)
 printf ("---");
 printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem-> ←
 ertek,
 melyseg);
 kiir (elem->bal_nulla);
 kiir (elem->jobb_egy);
 --melyseg;
 }
}
...
```

A program futtatása után az alábbi lesz az eredmény:

```
dave@dave-K501UB:~/gyakorlas/pre$ more b.txt
00011101110
dave@dave-K501UB:~/gyakorlas/pre$./lzwpre < b.txt >output.txt
dave@dave-K501UB:~/gyakorlas/pre$ more output.txt

---/(1)
-----0(2)
-----0(3)
-----1(3)
-----1(2)
-----1(3)
-----0(4)
melyseg=3
altag=2.333333
szoras=0.577350
dave@dave-K501UB:~/gyakorlas/pre$
```

6.5. ábra. Az LZW preorder bejárása

Végül pedig nézzük meg a postorder eredményét:

```
...
void
kiir (BINFA_PTR elem)
{
 if (elem != NULL)
 {
 ++melyseg;
 if (melyseg > max_melyseg)
 max_melyseg = melyseg;
 kiir (elem->bal_nulla);
 kiir (elem->jobb_egy);
 for (int i = 0; i < melyseg; ++i)
 printf ("---");
 printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem-> ←
 ertek,
 melyseg);
 --melyseg;
 }
}
...
```

```
dave@dave-K501UB:~/gyakorlas/post$ gcc lzwpost.c -o lzwpost -lm
dave@dave-K501UB:~/gyakorlas/post$./lzwpost < b.txt >output.txt
dave@dave-K501UB:~/gyakorlas/post$ more output.txt

-----0(3)
-----1(3)
-----0(2)
-----0(4)
-----1(3)
-----1(2)
---/(1)
melyseg=3
altag=2.333333
szoras=0.577350
dave@dave-K501UB:~/gyakorlas/post$
```

6.6. ábra. Az LZW postorder bejárása

## 6.4. Tag a gyökér

Az LZW algoritmust ültess át egy C++ osztályba, legyen egy Tree és egy beágyazott Node osztálya. A gyökér csomópont legyen kompozícióban a fával!

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/blob/master/attention\\_raising/Source/welch/binfa/-binfa%20c++/z3a7.cpp](https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/welch/binfa/-binfa%20c++/z3a7.cpp)

**Megjegyzés**

A feladat megoldásában tutorként részt vett: Fürjes-Beke Péter

Ebben a feladatban a C binfás programunkat fogjuk átírni C++ verzióba. Jelen esetünkben van egy LZWBinFa osztályunk, azon belül egy Csomopont osztály továbbá egy void usage (void) és az elmaradhatatlan int main() függvényünk is. Az LZWBinFa fa egy pointer, amely mindig az épülő LZW fa azon csomópontjára mutat, amit az input feldolgozása során az LZW algoritmus logikája diktál. A kódcsipetben láthatunk egy void operator-t, aminek célja, hogy a bemenetként kapott elemeket beépítsük a fába. Itt döntjük el, hogy a kapott paraméter 0 vagy 1-es-e és attól függően foglalunk le a new paranccsal új tárterületet.

```
#include <iostream>
#include <cmath>
#include <fstream>

class LZWBinFa
{
public:

 LZWBinFa ():fa (&gyoker)
 {
 }
 ~LZWBinFa ()
 {
 szabadit (gyoker.egyGyermek ());
 szabadit (gyoker.nullasGyermek ());
 }

 void operator<< (char b)
 {
 if (b == '0')
 {
 if (!fa->nullasGyermek ())
 {
 Csomopont *uj = new Csomopont ('0');
 fa->ujNullasGyermek (uj);
 fa = &gyoker;
 }
 else
 {
 fa = fa->nullasGyermek ();
 }
 }
 else
 {
 if (!fa->egyGyermek ())
 {
 Csomopont *uj = new Csomopont ('1');
```

```

 fa->ujEgyesGyermekek (uj);
 fa = &gyoker;
 }
 else
 {
 fa = fa->egyenesGyermekek ();
 }
}

void kiir (void)
{
 melyseg = 0;
 kiir (&gyoker, std::cout);
}

int getMelyseg (void);
double getAtlag (void);
double getSzoras (void);

friend std::ostream & operator<< (std::ostream & os, LZWBinFa & bf)
{
 bf.kiir (os);
 return os;
}
void kiir (std::ostream & os)
{
 melyseg = 0;
 kiir (&gyoker, os);
}

```

A Csomopont osztályba implementáljuk a Csomopont konstruktort, ami paraméter nélküli lesz és itt hozza létre a '/' karakterrel kezdődő gyöker csomópontját. Ez lesz tehát az alapértelmezett, de, ha valami betűvel hívjuk, akkor '/' helyett azt teszi be és a két gyermekre mutató mutatót pedig mindkét esetben nullára állítjuk. Ezután a nullasGyermekek és egyesGyermekek függvényekben kérdezzük le az aktuális csomópont bal és jobb oldali gyermekét, ujNullasGyermekek és a ujEgyesGyermekek függvényekben. Pontosan ugyanúgy ahogy már a C programunknál megismertük. Ami számunkra most újdonság, az, hogy a Csomopont osztályban van egy private típusú kódrészlet is. Ez azt jelenti, hogy az LZWBinfa ezekhez az adatokhoz nem közvetlenül fér hozzá, hanem csak lekérdező üzenetekkel érheti el őket.

```

private:
 class Csomopont
 {
 public:
 Csomopont (char b = '/'):betu (b), balNulla (0), jobbEgy (0)
 {
 };
 ~Csomopont ()
 {

```

```
};

Csomopont *nullasGyermekek () const
{
 return balNulla;
}

Csomopont *egykesGyermekek () const
{
 return jobbEgy;
}

void ujNullasGyermekek (Csomopont * gy)
{
 balNulla = gy;
}

void ujEgykesGyermekek (Csomopont * gy)
{
 jobbEgy = gy;
}

char getBetu () const
{
 return betu;
}

private:

 char betu;
 Csomopont *balNulla;
 Csomopont *jobbEgy;
 Csomopont (const Csomopont &);
 Csomopont & operator= (const Csomopont &);
};
```

Ezután létrehozunk egy fa mutatót, ami mindig a csomópontra mutat. Van még nekünk egy protected osztályunk is, aminek szerepe, hogy az LZWBinFa osztályai el tudják érni az itt lévő adatokat, de ezen kívül más nem fér hozzá. Ez tartalmazza a mélység, átlag és szórás kiszámításához szükséges függvényeket.

```
Csomopont *fa;
int melyseg, atlagosszeg, atlagdb;
double szorasosszeg;
LZWBinFa (const LZWBinFa &);
LZWBinFa & operator= (const LZWBinFa &);

void kiir (Csomopont * elem, std::ostream & os)
{
```



```

 if (elem != NULL)
 {
 ++melyseg;
 kiir (elem->egyGyermek (), os);
 for (int i = 0; i < melyseg; ++i)
 os << "---";
 os << elem->getBetu () << "(" << melyseg - 1 << ")" << std::endl;
 kiir (elem->nullasGyermek (), os);
 --melyseg;
 }
 }
 void szabadit (Csomopont * elem)
 {
 if (elem != NULL)
 {
 szabadit (elem->egyGyermek ());
 szabadit (elem->nullasGyermek ());
 delete elem;
 }
 }
}

```

protected:

```

 Csomopont gyoker;
 int maxMelyseg;
 double atlag, szoras;

 void rmelyseg (Csomopont * elem);
 void ratlag (Csomopont * elem);
 void rszoras (Csomopont * elem);

};

```

Az LZWBinFa osztályon kívül meghatározzuk a mélységet, átlagot, a szórást és a kiír függvényeket, amik megegyeznek a C programból ismertekkel. Ezeket a függvényeket rekurzívan valósítjuk meg. Az osztályon belüli függvényeket az LZWBinFa:: előtaggal érhetjük el.

```

int
LZWBinFa::getMelyseg (void)
{
 melyseg = maxMelyseg = 0;
 rmelyseg (&gyoker);
 return maxMelyseg - 1;
}

double

```

```
LZWBinFa::getAtlag (void)
{
 melyseg = atlagosszeg = atlagdb = 0;
 ratlag (&gyoker);
 atlag = ((double) atlagosszeg) / atlagdb;
 return atlag;
}

double
LZWBinFa::getSzoras (void)
{
 atlag = getAtlag ();
 szorasosszeg = 0.0;
 melyseg = atlagdb = 0;

 rszoras (&gyoker);

 if (atlagdb - 1 > 0)
 szoras = std::sqrt (szorasosszeg / (atlagdb - 1));
 else
 szoras = std::sqrt (szorasosszeg);

 return szoras;
}

void
LZWBinFa::rmelyseg (Csomopont * elem)
{
 if (elem != NULL)
 {
 ++melyseg;
 if (melyseg > maxMelyseg)
 maxMelyseg = melyseg;
 rmelyseg (elem->egyenesGyermekek ());
 rmelyseg (elem->nullasGyermekek ());
 --melyseg;
 }
}

void
LZWBinFa::ratlag (Csomopont * elem)
{
 if (elem != NULL)
 {
 ++melyseg;
 ratlag (elem->egyenesGyermekek ());
 ratlag (elem->nullasGyermekek ());
 --melyseg;
 if (elem->egyenesGyermekek () == NULL && elem->nullasGyermekek () == NULL ↔
)
 }
}
```

```
 {
 ++atlagdb;
 atlagosszeg += melyseg;
 }
 }
}

void
LZWBinFa::rszoras (Csomopont * elem)
{
 if (elem != NULL)
 {
 ++melyseg;
 rszoras (elem->egyenesGyermek ());
 rszoras (elem->nullasGyermek ());
 --melyseg;
 if (elem->egyenesGyermek () == NULL && elem->nullasGyermek () == NULL <=
)
 {
 ++atlagdb;
 szorasosszeg += ((melyseg - atlag) * (melyseg - atlag));
 }
 }
}
```

Ezután megvizsgáljuk, hogy a user által megadott argumentumok megfelelnek-e a feltételeknek. Ha nem felel meg, akkor hibaüzenetet íratunk ki. Ellenkező esetben while ciklussal beolvassuk a bemenetet és a második while ciklusban dolgozzuk fel a fát. Ha a beolvasott karakter a 0x3e, azaz akkor kommentben igazgá tesszük, ha sortörés (0x0a) akkor pedig hamissá.

```
void
usage (void)
{
 std::cout << "Usage: lzwtree in_file -o out_file" << std::endl;
}

int
main (int argc, char *argv[])
{
 if (argc != 4)
 {
 usage ();
 return -1;
 }

 char *inFile = *++argv;
```

```
if ((*++argv) + 1) != 'o')
{
 usage ();
 return -2;
}

std::fstream beFile (inFile, std::ios_base::in);

if (!beFile)
{
 std::cout << inFile << " nem letezik..." << std::endl;
 usage ();
 return -3;
}

std::fstream kiFile (*++argv, std::ios_base::out);

unsigned char b;
LZWBinFa binFa;

while (beFile.read ((char *) &b, sizeof (unsigned char)))
 if (b == 0x0a)
 break;

bool kommentben = false;

while (beFile.read ((char *) &b, sizeof (unsigned char)))
{
 if (b == 0x3e)
 {
 kommentben = true;
 continue;
 }

 if (b == 0x0a)
 {
 kommentben = false;
 continue;
 }

 if (kommentben)
 continue;

 if (b == 0x4e)
 continue;

 for (int i = 0; i < 8; ++i)
 {
 if (b & 0x80)
```

```
 binFa << '1';
 else
 binFa << '0';
 b <= 1;
}

}

kiFile << binFa;

kiFile << "depth = " << binFa.getMelyseg () << std::endl;
kiFile << "mean = " << binFa.getAtlag () << std::endl;
kiFile << "var = " << binFa.getSzoras () << std::endl;

kiFile.close ();
beFile.close ();

return 0;
}
```

A program futtatásához bemenetként egy tömörített fájlra van szükség, ha sima txt-t adunk meg akkor nem lesz bináris fáánk. Mivel a tömörített fájl eredménye nagyon hosszú lett (13-as mélységű), ezért ezt most nem teszem be.

## 6.5. Mutató a gyökér

Írd át az előző forrást, hogy a gyökér csomópont ne kompozícióban, csak aggregációban legyen a fával!

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/blob/master/attention\\_raising/Source/welch/binfa-binfa%20c++/z3a8.cpp](https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/welch/binfa-binfa%20c++/z3a8.cpp)

Ez a feladat az előző példánkkal teljesen megegyezik, azzal a különbséggel, hogy a destruktorba hozzáadtunk egy új sort: `delete gyoker`

```
~LZWBinFa ()
{
 szabadit (gyoker->egyenesGyermekek ());
 szabadit (gyoker->nullasGyermekek ());
 delete gyoker;
}
```

```

ch/binfa/binfa c++$ more output.txt
-----1(2)
-----1(1)
-----0(2)
-----1(5)
-----1(4)
-----0(3)
-----0(4)
---/(0)
-----1(3)
-----0(4)
-----0(5)
-----0(6)
-----1(2)
-----0(3)
-----0(1)
-----1(4)
-----1(3)
-----0(4)
-----0(2)
-----1(4)
-----0(5)
-----0(3)
-----1(5)
-----0(4)
-----0(5)

depth = 6
mean = 4.3
var = 1.1595

```

6.7. ábra. Mutató a gyökér

## 6.6. Mozgató szemantika

Írj az előző programhoz mozgató konstruktort és értékadást, a mozgató konstruktor legyen a mozgató értékadásra alapozva!

Megoldás videó:

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/blob/master/attention\\_raising/Source/welch/binfa-binfa%20c++/z3a9.cpp](https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/welch/binfa-binfa%20c++/z3a9.cpp)

A Mozgató szemantikát elsősorban nagyobb méretű objektumok esetén szoktuk használni, segítségével kevesebb helyet foglalunk, mely gyorsabb programot eredményez. Lényege, hogy egy változó értékeit másolás helyett áthelyezzük egy másik vektorba, ekkor a régi objektum törlésre kerül, azaz elveszíti elemeit. A C++ nyelvben ezt a `move()` függvénnyel oldhatjuk meg. Az alap C++ Binfás programunkhoz képest annyi a változtatás, hogy a program végét kiegészítjük néhány sorral, az alábbiakkal:

```

LZWBinFa binFa3 = std::move (binFa);

kiFile << "depth = " << binFa3.getMelyseg () << std::endl;
kiFile << "mean = " << binFa3.getAtlag () << std::endl;
kiFile << "var = " << binFa3.getSzoras () << std::endl;

```

Ha megfigyeljük a kódcsipetet, akkor a forráskódban látható, hogy tulajdonképpen copy-paste a `kiFile` kiíró kódsorok, csak a változó nevét változtattuk meg `binFa3`-ra, ugyanis fentebb deklaráltunk egy ugyanilyen nevű változót, amihez hozzárendeltük a `move()` függvényt. Paraméterként a `binFa`-t adtuk meg.

```
LZWBinFa (LZWBinFa && regi) {
 std::cout << "LZWBinFa move ctor" << std::endl;

 gyoker.ujEgyesGyermekek (regi.gyoker.egyesGyermekek());
 gyoker.ujNullasGyermekek (regi.gyoker.nullasGyermekek());

 regi.gyoker.ujEgyesGyermekek (nullptr);
 regi.gyoker.ujNullasGyermekek (nullptr);

}
```

Programunkat kiegészítettük még a fenti kódcsipettel is. Itt a `nullptr` kulcsszó a null pointert jelöli. Ez egy tetszőleges mutató típusra implicit konvertálódik. Erre azért van szükség, mert a 0 konstansnak két jelentése van: egyrészt 0 számot, másrészt pedig a null pointer.

## 7. fejezet

# Helló, Conway!

### 7.1. Hangyaszimulációk

Írj Qt C++-ban egy hangyaszimulációs programot, a forrásaidról utólag reverse engineering jelleggel készíts UML osztálydiagramot is!

Megoldás videó: <https://bhaxor.blog.hu/2018/10/10/myrmecologist>

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/tree/master/attention\\_raising/Source/conway/Myrmecologist](https://gitlab.com/davidhalasz/bhax/tree/master/attention_raising/Source/conway/Myrmecologist)

Ebben a fejezetben a hangyaszimulációs programot mutatom be. A program 6 fájlból áll. Az `ant.h` fájlban deklaráljuk a hangya felépítéséhez kapcsolódó változókat. Az `antwin.h` fájlban az `Antwin()` függvényben is szintén változókat deklarálunk és értékeket adunk meg. Itt az ablakkal és a hangyákkal kapcsolatos adatok vannak megadva (pl. ablak szélesség, magasság, hangyák száma, stb.). A `closeEvent()` az ablakeseményeket figyeli. Ha bezárjuk az ablakot, akkor meghívja a `finish()` függvényt, ahol a futtatás átvált `false` értékre. A `keyPressEvent()` függvény pedig a billentyűzet leütéseket figyeli. Jelen esetben azt mondjuk, hogyha az user által leütött billentyűzet azonos a P gombbal (`event->key() == Qt::Key_P`), akkor megállítjuk a programot. Ha a Q billentyűt vagy az esc-t nyomjuk le, akkor kilép a proglamból (`close()`).

```
#ifndef ANTWIN_H
#define ANTWIN_H

#include <QMainWindow>
#include <QPainter>
#include <QString>
#include <QCloseEvent>
#include "antthread.h"
#include "ant.h"

class AntWin : public QMainWindow
{
 Q_OBJECT

public:
 AntWin(int width = 100, int height = 75,
 int delay = 120, int numAnts = 100,
```



```
 int pheromone = 10, int nbhPheromon = 3,
 int evaporation = 2, int cellDef = 1,
 int min = 2, int max = 50,
 int cellAntMax = 4, QWidget *parent = 0);

AntThread* antThread;

void closeEvent (QCloseEvent *event) {

 antThread->finish();
 antThread->wait();
 event->accept();
}

void keyPressEvent (QKeyEvent *event)
{

 if (event->key() == Qt::Key_P) {
 antThread->pause();
 } else if (event->key() == Qt::Key_Q
 || event->key() == Qt::Key_Escape) {
 close();
 }

}

virtual ~AntWin();
void paintEvent(QPaintEvent*);

private:

 int ***grids;
 int **grid;
 int gridIdx;
 int cellWidth;
 int cellHeight;
 int width;
 int height;
 int max;
 int min;
 Ants* ants;

public slots :
 void step (const int &);

};

#endif
```

Az `antwin.cpp` fájlban belül a `setWindowTitle()` segítségével adunk nevet az ablaknak, ami a program elindítása után a fejlécben fog majd megjelenni. Az ablak méretét a header fájlból kérjük le. Ezután `for` ciklussal létrehozunk két négyzetrácsot, ezek egy `grids` nevű tömbben lesznek eltárolva. Az `ants` változó meghívja az `Ants()` függvényt, azaz létrehozuk a hangyákat. Az `antThread` pedig a fenti adatok alapján létrehozza az egész életteret.

[illegible]



```
);

 QPainter painter (j*cellWidth, i*cellHeight,
 cellWidth, cellHeight);

 }

 for (auto h: *ants) {
 QPainter painter (QPen (Qt::black, 1));

 painter.drawRect (h.x*cellWidth+1, h.y*cellHeight+1,
 cellWidth-2, cellHeight-2);

 }

 painter.end();
}

AntWin::~AntWin()
{
 delete antThread;

 for (int i=0; i<height; ++i) {
 delete[] grids[0][i];
 delete[] grids[1][i];
 }

 delete[] grids[0];
 delete[] grids[1];
 delete[] grids;

 delete ants;
}

void AntWin::step (const int &gridIdx)
{
 this->gridIdx = gridIdx;
 update();
}
```

Az `antthread.cpp`-n belül történik a hangyák kiírása és mozgatása. Ahhoz, hogy a hangyák mozgása véletlenszerűen történjen, a `qrand()` függvényt hívjuk segítségül, ahol paraméterként a másodpercenként változó időt adjuk meg.

```
#include "antthread.h"
#include <QDebug>
#include <cmath>
```

```
#include <QDateTime>

AntThread::AntThread (Ants* ants, int*** grids,
 int width, int height,
 int delay, int numAnts,
 int pheromone, int nbrPheromone,
 int evaporation,
 int min, int max, int cellAntMax)
{
 this->ants = ants;
 this->grids = grids;
 this->width = width;
 this->height = height;
 this->delay = delay;
 this->pheromone = pheromone;
 this->evaporation = evaporation;
 this->min = min;
 this->max = max;
 this->cellAntMax = cellAntMax;
 this->nbrPheromone = nbrPheromone;

 numAntsinCells = new int*[height];
 for (int i=0; i<height; ++i) {
 numAntsinCells[i] = new int [width];
 }

 for (int i=0; i<height; ++i)
 for (int j=0; j<width; ++j) {
 numAntsinCells[i][j] = 0;
 }

 qrand (QDateTime::currentMSecsSinceEpoch());

 Ant h {0, 0};
 for (int i {0}; i<numAnts; ++i) {

 h.y = height/2 + qrand() % 40-20;
 h.x = width/2 + qrand() % 40-20;

 ++numAntsinCells[h.y][h.x];

 ants->push_back (h);
 }

 gridIdx = 0;
}
```

## 7.2. Java életjáték

Írd meg Java-ban a John Horton Conway-féle életjátékot, valósítsa meg a sikló-kilövőt!

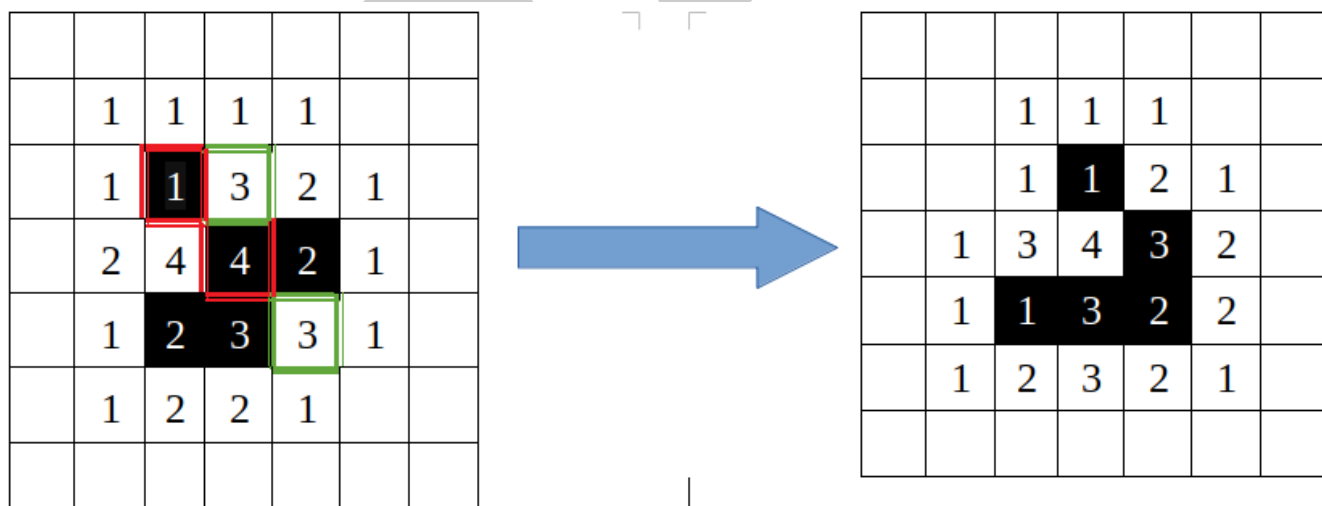
Megoldás videó:

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/blob/master/attention\\_raising/Source/conway/java/Sejtautomata.java](https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/conway/java/Sejtautomata.java)

John Horton Conway a 70-es évek elején valósította meg az általa elnevezett Élejtájékot, melynek lényege, hogy van egy négyzethálós élettér, és minden cellában egy sejt élhet. Minden sejtnek 8 szomszédja lehet: négy keresztben, négy pedig átlósan. A szabályok pedig a következők:

- Ha egy sejtnek kettő vagy három szomszédja van: életben marad.
- Ha egy sejtnek négy vagy több szomszédja van: meghal. Ugyanez vonatkozik a 2-nél kisebb szomszédokkal rendelkezőkre.
- Ha egy üres cellának pontosan három élő sejt van a szomszédjában, akkor ott új sejt születik.

Innen az élet elnevezés, ugyanis a szimuláció generációról generációra mutatja meg a sejtek alakulását. Hogy könnyebb legyen elképzelni a fenti szabályokat, készítettem egy ábrát, ami egy lépést (azaz egy generáció születését) mutat be. Az alábbi ábrán a fekete háttérrel rendelkezők az élő sejtek, a piros szegéllyel jelöltek jelentik a halálozást, mivel 2-nél kisebb és 3-nál több szomszédokkal rendelkeznek. Ezek a következő generációban tehát fehér háttérűek lesznek. Vannak még továbbá üres cellák, zöld szegéllyel, ezeknek pontosan 3 élő sejtrel rendelkező szomszédjuk van, így a következő lépésben itt fognak új sejtek születni.



7.1. ábra. A sejtek alakulása

Ezt a szimulációt fogjuk megvalósítani java-ban, melynek neve `Sejtautomata.java`. Először is létrehozunk a `Sejtautomata` osztályt, majd azon belül deklaráljuk a változókat és értékeket rendelünk hozzá. Például megadunk két boolean típusú változót, mellyel azt fogjuk meghatározni, hogy egy sejt élő vagy halott lesz-e. Továbbá a rács elkészítéséhez fontos adatokat is itt deklaráljuk.

```
public class Sejtautomata extends java.awt.Frame implements Runnable {

 public static final boolean ÉLŐ = true;
 public static final boolean HALOTT = false;
 protected boolean [][][] rácsok = new boolean [2][][];
 protected boolean [][] rács;
 protected int rácsIndex = 0;
 protected int cellaSzélesség = 20;
 protected int cellaMagasság = 20;
 protected int szélesség = 20;
 protected int magasság = 10;
 protected int várakozás = 1000;
 private java.awt.Robot robot;
 private boolean pillanatfelvétel = false;
 private static int pillanatfelvételSzámláló = 0;
 /**
 * Létrehoz egy <code>Sejtautomata</code> objektumot.
 *
 * @param szélesség a sejtter szélessége.
 * @param magasság a sejtter szélessége.
 */
 ...
}
```

Ezután létrehozuk a `Sejtautomata()` függvényt. Itt készítjük el a rácsokat a fenti adatok felhasználásával. Kezdetben a rács minden cellája halott lesz, ezért for ciklussal végigmegyünk először az egyes cellákon és `HALOTT` azaz `false` értékeket rendelünk hozzá. Meghívjuk a `siklóKilövő()` függvényt, melynek paraméterként a rácsot, a sor és az oszlop számát adjuk át. Lényege, hogy a sejtterbe ezzel helyezzük el a "sikló ágyút". Ez fog majd először megjelenni a programunk elindítása után, mely egy adott irányba fog elindulni. Az `addWindowListener` függvény figyel az ablakot, ha bezárjuk, akkor `windowClosing` függvény leállítja a programot is. Van még egy `addKeyListener` függvényünk is, amely a nevéből adódóan a billentyűzetleütéseket figyel. Itt megadjuk hogy, hogy ha a leütött billentyűzet megegyezik a programban megadottakkal, akkor végezze ez az utasításokat. Például 'K' leütése megfelel a sejtek méretét, 'N' billentyű lenyomásával pedig növeljük. Az 'S' billentyű pillanatfelvételt készít. Itt azért van felkiáltójel a pillanatfelvétel mögött, mert alapértelmezettként `false` van megadva, ezért ezzel `true` értéké alakítjuk át. A 'G' billentyűzettel megfelezzük az időt, azaz gyorsítjuk, végül pedig az 'L'-el fogjuk tudni lassítani, ha arra lenne szükségünk.

...

```
public Sejtautomata(int szélesség, int magasság) {
 this.szélesség = szélesség;
 this.magasság = magasság;

 rácsok[0] = new boolean[magasság][szélesség];
 rácsok[1] = new boolean[magasság][szélesség];
 rácsIndex = 0;
 rács = rácsok[rácsIndex];
}
```

```
for(int i=0; i<rács.length; ++i)
 for(int j=0; j<rács[0].length; ++j)
 rács[i][j] = HALOTT;

siklóKilövő(rács, 5, 60);

addWindowListener(new java.awt.event.WindowAdapter() {
 public void windowClosing(java.awt.event.WindowEvent e) {
 setVisible(false);
 System.exit(0);
 }
});

addKeyListener(new java.awt.event.KeyAdapter() {

 public void keyPressed(java.awt.event.KeyEvent e) {
 if(e.getKeyCode() == java.awt.event.KeyEvent.VK_K) {

 cellaSzélesség /= 2;
 cellaMagasság /= 2;
 setSize(Sejtautomata.this.szélesség*cellaSzélesség,
 Sejtautomata.this.magasság*cellaMagasság);
 validate();
 } else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_N) {

 cellaSzélesség *= 2;
 cellaMagasság *= 2;
 setSize(Sejtautomata.this.szélesség*cellaSzélesség,
 Sejtautomata.this.magasság*cellaMagasság);
 validate();
 } else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_S)
 pillanatfelvétel = !pillanatfelvétel;
 else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_G)
 várakozás /= 2;
 else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_L)
 várakozás *= 2;
 repaint();
 }
});

addMouseListener(new java.awt.event.MouseAdapter() {
 public void mousePressed(java.awt.event.MouseEvent m) {
 int x = m.getX()/cellaSzélesség;
 int y = m.getY()/cellaMagasság;
 rácsok[rácsIndex][y][x] = !rácsok[rácsIndex][y][x];
 repaint();
 }
});

addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {
```



```
 public void mouseDragged(java.awt.event.MouseEvent m) {
 int x = m.getX()/cellaSzélesség;
 int y = m.getY()/cellaMagasság;
 rácsok[rácsIndex][y][x] = ÉLŐ;
 repaint();
 }
 });

 cellaSzélesség = 10;
 cellaMagasság = 10;
 try {
 robot = new java.awt.Robot(
 java.awt.GraphicsEnvironment.
 getLocalGraphicsEnvironment().
 getDefaultScreenDevice());
 } catch (java.awt.AWTException e) {
 e.printStackTrace();
 }
 setTitle("Sejtautomata");
 setResizable(false);
 setSize(szélesség*cellaSzélesség,
 magasság*cellaMagasság);
 setVisible(true);

 new Thread(this).start();
}
...
```

Valahogy ki kellene rajzolni az élő sejteket is, ezért készítünk egy `paint()` függvényt. For ciklussal végigmenyünk a sorokon és az oszlopokon, ha az adott sor adott oszlopában van egy ÉLŐ sejt, akkor feketére színezzük, különben fehér lesz. Továbbá itt figyeljük azt is, hogy a pillanatfelvétel készítés aktiválva lett-e, ha igen, meghívjuk a pillanatfelvétel függvényt és készítünk egy képet továbbá visszaállítjuk `false` értékre, különben folyamatosan készítené a képeket, amit ugye nem szeretnénk, hogy megtörténjen.

```
...

public void paint(java.awt.Graphics g) {

 boolean [][] rács = rácsok[rácsIndex];

 for(int i=0; i<rács.length; ++i) {
 for(int j=0; j<rács[0].length; ++j) {

 if(rács[i][j] == ÉLŐ)
 g.setColor(java.awt.Color.BLACK);
 else
 g.setColor(java.awt.Color.WHITE);
 g.fillRect(j*cellaSzélesség, i*cellaMagasság,
 cellaSzélesség, cellaMagasság);
 }
 }
}
```

```
 g.setColor(java.awt.Color.LIGHT_GRAY);
 g.drawRect(j*cellaSzélesség, i*cellaMagasság,
 cellaSzélesség, cellaMagasság);
 }
}

if(pillanatfelvétel) {

 pillanatfelvétel = false;
 pillanatfelvétel(robot.createScreenCapture
 (new java.awt.Rectangle
 (getLocation().x, getLocation().y,
 szélesség*cellaSzélesség,
 magasság*cellaMagasság)));
}

...
}
```

A `szomszédokSzama()` függvényben fogjuk megszámolni a szomszédokat az adott élő sejt körül. Itt is két for ciklus lesz, a sorok és az oszlopok miatt, azonban -1-től 1-ig fogjuk megvizsgálni, és csak akkor ha aza adott sejt nincs benne ( $i==0$  és  $j==0$ ).

```
...

public int szomszédokSzama(boolean [][] rács,
 int sor, int oszlop, boolean állapot) {
 int állapotúSzomszéd = 0;

 for(int i=-1; i<2; ++i)
 for(int j=-1; j<2; ++j)
 if(!((i==0) && (j==0))) {

 int o = oszlop + j;
 if(o < 0)
 o = szélesség-1;
 else if(o >= szélesség)

 o = 0;
 int s = sor + i;
 if(s < 0)
 s = magasság-1;
 else if(s >= magasság)
 s = 0;

 if(rács[s][o] == állapot)
 ++állapotúSzomszéd;
 }
 return állapotúSzomszéd;
}
```

```
}
...
```

A bevezetőben ismertetett életjáték szabályokat az `időFejlődés()`-ben adjuk meg. If függvénnyel megvizsgáljuk, hogy az adott cella ÉLŐ-e? Ha igen, akkor a vizsgált élő cellának van-e 2 vagy 3 ÉLŐ szomszédja, ha ez teljesül, akkor az ezt követő rácsban is élő marad az adott pozícióban. Ellenkező esetben halott lesz. Ha a vizsgált cella nem élő akkor lépünk az else ágba. If függvénnyel itt azt vizsgáljuk meg, hogy van-e pontosan 3 szomszédja, ha igen, akkor ez is élő lesz, ellenkező esetben halott.

```
...

public void időFejlődés() {

 boolean [][] rácsElőtte = rácsok[rácsIndex];
 boolean [][] rácsUtána = rácsok[(rácsIndex+1)%2];

 for(int i=0; i<rácsElőtte.length; ++i) { // sorok
 for(int j=0; j<rácsElőtte[0].length; ++j) { // oszlopok

 int élők = szomszédokSzama(rácsElőtte, i, j, ÉLŐ);

 if(rácsElőtte[i][j] == ÉLŐ) {
 /* Élő élő marad, ha kettő vagy három élő
 szomszédja van, különben halott lesz. */
 if(élők==2 || élők==3)
 rácsUtána[i][j] = ÉLŐ;
 else
 rácsUtána[i][j] = HALOTT;
 } else {
 /* Halott halott marad, ha három élő
 szomszédja van, különben élő lesz. */
 if(élők==3)
 rácsUtána[i][j] = ÉLŐ;
 else
 rácsUtána[i][j] = HALOTT;
 }
 }
 }
 rácsIndex = (rácsIndex+1)%2;
}
...
```

## 7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás videó:

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/tree/master/attention\\_raising/Source/conway/c++](https://gitlab.com/davidhalasz/bhax/tree/master/attention_raising/Source/conway/c++)

A program felépítése nagyon hasonlít a Mandelbrot nagyítás fejezetnél megismert programhoz, és a futtatása is ugyanúgy történik (qmake és make parancsok használata). Itt is 5 különálló fájlból fog állni a programunk. A main.cpp-ben adjuk meg, hogy mekkora legyen az ablakunk mérete. Jelen esetünkben 100x75 pixel méretű lesz. w.show() függvénnyel hívjuk elő a az ablakot.

```
#include <QApplication>
#include "sejtablak.h"
#include <QDesktopWidget>

int main(int argc, char *argv[])
{
 QApplication a(argc, argv);
 SejtAblak w(100, 75);
 w.show();

 return a.exec();
}
```

A sejtablak.h fájlban megadjuk az ablakon belüli rácsokat, melynek mérete megegyezik a az ablakunk méretével. Továbbá az ELO és HALOTT boolean típusú változónk konstans változók lesznek, azaz ezt a későbbiekben a programon belül ezek értékeit nem fogjuk tudni megváltoztatni. A protected-en belül pedig további változókat deklarálunk.

```
#ifndef SEJTABLAK_H
#define SEJTABLAK_H

#include <QMainWindow>
#include <QPainter>
#include "sejtszal.h"

class SejtSzal;

class SejtAblak : public QMainWindow
{
 Q_OBJECT

public:
 SejtAblak(int szelesseg = 100, int magassag = 75, QWidget * ←
 parent = 0);

 ~SejtAblak();
 static const bool ELO = true;
 static const bool HALOTT = false;
 void vissza(int racsIndex);

protected:
 bool ***racsok;
```

```
bool **racs;
int racsIndex;
int cellaSzelesseg;
int cellaMagassag;
int szelesseg;
int magassag;
void paintEvent(QPaintEvent*);
void siklo(bool **racs, int x, int y);
void sikloKilovo(bool **racs, int x, int y);

private:
 SejtSzal* eletjatek;

};

#endif // SEJTABLAK_H
```

## sejtszal.h

```
#ifndef SEJTSZAL_H
#define SEJTSZAL_H

#include <QThread>
#include "sejtablak.h"

class SejtAblak;

class SejtSzal : public QThread
{
 Q_OBJECT

public:
 SejtSzal(bool ***racsok, int szelesseg, int magassag,
 int varakozas, SejtAblak *sejtAblak);
 ~SejtSzal();
 void run();

protected:
 bool ***racsok;
 int szelesseg, magassag;
 int racsIndex;
 int varakozas;
 void idoFejlodes();
 int szomszedokSzama(bool **racs,
 int sor, int oszlop, bool allapot);
 SejtAblak* sejtAblak;

};

#endif // SEJTSZAL_H
```

A `sejtablak.cpp` fájlban a `SejtAblak()` függvénnyel az ablakkal kapcsolatos adatokat adjuk meg. Ez 3 paramétert vár: szélesség, magasság és egy `QWidget` "parent" paraméteréhez tartozik. Ennek értéke 0. A `setWindowTitle()` azt jelenti, hogy a zárójelben megadott szöveg lesz majd az ablakunk fejlécében látható cím. A `setFixedSize()`-ban a szélességet és a magasságot megszorozzuk 6-al, majd ez alapján for ciklusokkal létrehozunk két rácsot. A `paintEvent`-en belül végiglépkedünk a sorokon és az oszlopokon és ha az adott cellában van egy ELO, akkor feketére színezzük a QT segítségével. Ellenkező esetben fehér lesz.

```
#include "sejtablak.h"

SejtAblak::SejtAblak(int szelesseg, int magassag, QWidget *parent)
: QMainWindow(parent)
{
 setWindowTitle("A John Horton Conway-féle életjáték");

 this->magassag = magassag;
 this->szelesseg = szelesseg;

 cellaSzelesseg = 6;
 cellaMagassag = 6;

 setFixedSize(QSize(szelesseg*cellaSzelesseg, magassag*cellaMagassag));

 racsok = new bool**[2];
 racsok[0] = new bool*[magassag];
 for(int i=0; i<magassag; ++i)
 racsok[0][i] = new bool [szelesseg];
 racsok[1] = new bool*[magassag];
 for(int i=0; i<magassag; ++i)
 racsok[1][i] = new bool [szelesseg];

 racsIndex = 0;
 racs = racsok[racsIndex];

 for(int i=0; i<magassag; ++i)
 for(int j=0; j<szelesseg; ++j)
 racs[i][j] = HALOTT;

 sikloKilovo(racs, 5, 60);

 eletjatek = new SejtSzal(racsok, szelesseg, magassag, 120, this);

 eletjatek->start();
}

void SejtAblak::paintEvent(QPaintEvent*) {
```

```
QPainter qpainter(this);

bool **racs = racsok[racsIndex];
for(int i=0; i<magassag; ++i) {
 for(int j=0; j<szelesseg; ++j) {
 if(racs[i][j] == ELO)
 qpainter.fillRect(j*cellaSzelesseg, i*cellaMagassag,
 cellaSzelesseg, cellaMagassag, Qt::black);
 else
 qpainter.fillRect(j*cellaSzelesseg, i*cellaMagassag,
 cellaSzelesseg, cellaMagassag, Qt::white);
 qpainter.setPen(QPen(Qt::gray, 1));

 qpainter.drawRect(j*cellaSzelesseg, i*cellaMagassag,
 cellaSzelesseg, cellaMagassag);
 }
}
qpainter.end();
}

SejtAblak::~SejtAblak()
{
 delete eletjatek;

 for(int i=0; i<magassag; ++i) {
 delete[] racsok[0][i];
 delete[] racsok[1][i];
 }

 delete[] racsok[0];
 delete[] racsok[1];
 delete[] racsok;
}

void SejtAblak::vissza(int racsIndex)
{
 this->racsIndex = racsIndex;
 update();
}

void SejtAblak::siklo(bool **racs, int x, int y) {
 racs[y+ 0][x+ 2] = ELO;
 racs[y+ 1][x+ 1] = ELO;
 racs[y+ 2][x+ 1] = ELO;
 racs[y+ 2][x+ 2] = ELO;
 racs[y+ 2][x+ 3] = ELO;
```

```
}

void SejtAblak::sikloKilovo(bool **racs, int x, int y) {

 racs[y+ 6][x+ 0] = ELO;
 racs[y+ 6][x+ 1] = ELO;
 racs[y+ 7][x+ 0] = ELO;
 racs[y+ 7][x+ 1] = ELO;

 ...
}
```

Most nézzük meg a `sejtszal.cpp` fájlt. Itt fogjuk megszámolni a szomszédokat, és megadjuk a szomszédok száma alapján, hogy hol lesz élő és halott cella. Ennek a felépítése megegyezik a java programunkban ismertetett módszerrel.

```
#include "sejtszal.h"

SejtSzal::SejtSzal(bool ***racsok, int szelesseg, int magassag, int ←
 varakozas, SejtAblak *sejtAblak)
{
 this->racsok = racsok;
 this->szelesseg = szelesseg;
 this->magassag = magassag;
 this->varakozas = varakozas;
 this->sejtAblak = sejtAblak;

 racsIndex = 0;
}

int SejtSzal::szomszedokSzama(bool **racs,
 int sor, int oszlop, bool allapot) {
 int allapotuSzomszed = 0;
 for(int i=-1; i<2; ++i)
 for(int j=-1; j<2; ++j)

 if(!((i==0) && (j==0))) {

 int o = oszlop + j;
 if(o < 0)
 o = szelesseg-1;
 else if(o >= szelesseg)
 o = 0;

 int s = sor + i;
 if(s < 0)
 s = magassag-1;
 else if(s >= magassag)
```



```
 s = 0;

 if(racs[s][o] == allapot)
 ++allapotuSzomszed;
 }
 return allapotuSzomszed;
}

void SejtSzal::idoFejlodes() {

 bool **racsElotte = racsok[racsIndex];
 bool **racsUtana = racsok[(racsIndex+1)%2];

 for(int i=0; i<magassag; ++i) { // sorok
 for(int j=0; j<szelesseg; ++j) { // oszlopok

 int elok = szomszedokSzama(racsElotte, i, j, SejtAblak::ELO);

 if(racsElotte[i][j] == SejtAblak::ELO) {

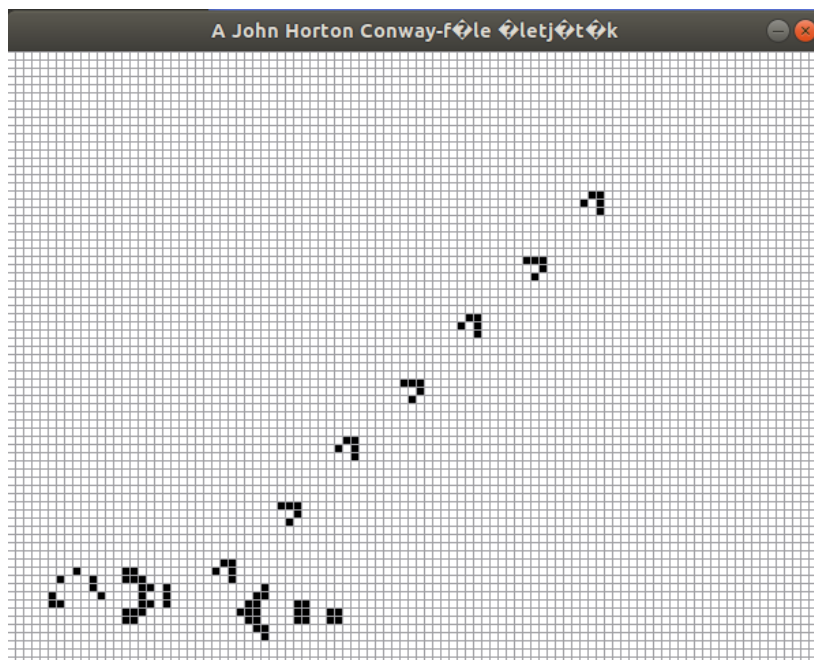
 if(elok==2 || elok==3)
 racsUtana[i][j] = SejtAblak::ELO;
 else
 racsUtana[i][j] = SejtAblak::HALOTT;
 } else {

 if(elok==3)
 racsUtana[i][j] = SejtAblak::ELO;
 else
 racsUtana[i][j] = SejtAblak::HALOTT;
 }
 }
 }
 racsIndex = (racsIndex+1)%2;
}

void SejtSzal::run()
{
 while(true) {
 QThread::msleep(varakozas);
 idoFejlodes();
 sejtAblak->vissza(racsIndex);
 }
}

SejtSzal::~SejtSzal()
{
}
```

A program futtatása a következő: a `qmake -project` létrehoz egy `.pro` végződésű fájlt, amibe bele kell írni, hogy `QT += widgets`. Majd a parancssorba beírjuk hogy `qmake *.pro`, ami létrehoz egy Makefile-t. `make` parancssal bináris fájlt hozunk létre, majd futtatjuk a szokásos `g++` programot. Ha mindent jól csináltunk, a következő képernyőt kell kapnunk:



7.2. ábra. A QT c++ program elindulása

## 7.4. BrainB Benchmark

Megoldás videó:

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/tree/master/attention\\_raising/Source/conway/brainb](https://gitlab.com/davidhalasz/bhax/tree/master/attention_raising/Source/conway/brainb)

A BrainB is egy Qt-s alkalmazás, ami OpenCV könyvtárt használ. A program célja a kiemelkedő e-sportolók korai felismerése illetve felkutatása, mérési módszerek alkalmazásával. A program lényege, hogy először egy doboz jelenik meg a képernyőn, középen egy kék körrel, ha az egeret minél tovább tartjuk a körön, annál több doboz jelenik meg a képernyőn és egyre gyorsabban is mozognak, egyre nehezebb lesz majd a körön tartani az egérmutatót. Most vizsgáljuk meg a mérés alapötletét, ami a `BrainBWin.cpp` fájlban található, azon belül a `BrainBWin::updateHeroes` függvényben. Először is kiszámítjuk az egér és a Samu Entropy doboza (karakter) közötti távolságot, ennek eredményét a `dist` változóban tároljuk el. Ezután megnézzük, hogy a távolság (`dist`) nagyobb-e, mint 121 pixel, ha az `if` feltétel teljesül, akkor a `nofLost` értékét megnöveljük eggyel. Ha a `nofLost` értéke több lesz, mint 12, azaz  $12 \cdot 100$  milliszekundum, akkor a feltétel teljesül, mert a játékos sikeresen tudta tartani több mint 1200 ms-on keresztül az adott karakteren az egérmutatót. Ekkor elmentjük az adatokat a `found2lost` verembe, aminek értéke bit per second (bps) lesz. Ha az user nem képes 1200ms-on keresztül tartani, akkor az `else` ágba lépünk, tehát a a játékos elvesztette a karaktert és ennek adatát is elmentjük a `lost2found` változóba.

```
void BrainBWin::updateHeroes (const QImage &image, const int & ←
 x, const int &y)
```

```

{

 if (start && !brainBThread->get_paused()) {
 int dist = (this->mouse_x - x) * (this->mouse_x - x) + ←
 (this->mouse_y - y) * (this->mouse_y - y);
 if (dist > 121) {
 ++nofLost;
 nofFound = 0;

 if (nofLost > 12) {
 if (state == found && firstLost) {
 found2lost.push_back (brainBThread-> ←
 get_bps());
 }
 firstLost = true;

 state = lost;
 nofLost = 0;
 brainBThread->decComp();
 }
 } else {
 ++nofFound;
 nofLost = 0;
 if (nofFound > 12) {

 if (state == lost && firstLost) {
 lost2found.push_back (brainBThread ←
 ->get_bps());
 }
 state = found;
 nofFound = 0;
 brainBThread->incComp();
 }
 }
 }
 pixmap = QPixmap::fromImage (image);
 update();
}

```

A fenti két számítást utána összeadjuk, elosztjuk 2-vel, majd 8-cal, végül pedig 1024-el osztjuk el és elmentjük egy text fájlba. Ez a BrainBWin.h fájlban van felépítve. Ebből most számunkra az int m1, int m2 és a double res változók az érdekesek. az integer típusú változók értékei a lost2found illetve a found2lost átlagával lesz egyenlő, ennek kiszámításához használjuk a mean() függvényt. A res változóba mentjük el a fentebb leírt képlettel kiszámított adatokat.

```

textStremam << "\n";
int m1 = m = mean (lost2found);
textStremam << "mean : " << m << "\n";
textStremam << "var : " << var (lost2found, m) << "\n" ←

```

```
 ;

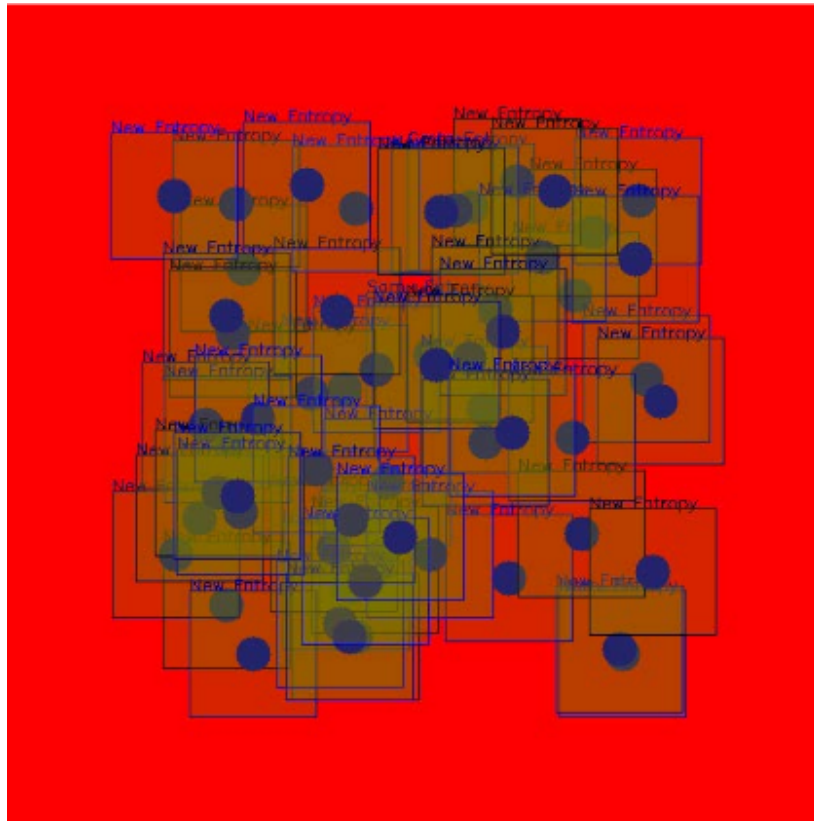
textStremam << "found2lost: " ;
for (int n : found2lost) {
 textStremam << n << ' ';
}
textStremam << "\n";
int m2 = m = mean (found2lost);
textStremam << "mean : " << m << "\n";
textStremam << "var : " << var (found2lost, m) << "\n" ←
 ;

if (m1 < m2) {
 textStremam << "mean(lost2found) < mean(found2lost)" << "\n ←
 ";
}

int min, sec;
millis2minsec (t, min, sec);
textStremam << "time : " << min << ":" << sec << "\n";

double res = ((((double) m1+ (double) m2) /2.0) /8.0) ←
 /1024.0;
textStremam << "U R about " << res << " Kilobytes\n";
```

A program futtatása után, ha az egeret képesek leszünk tartani egy adott kék ponton több, mint 1,2 másodpercig, akkor a program folyamatosan új dobozokat jelenít meg, és egyre gyorsabban is mozognak:



7.3. ábra. BrainB Benchmark

Ezeknek a dobozoknak a létrehozása és mozgatása a `BrainBThread.cpp` és a `BrainBThread.h` fájlokban vannak implementálva. A `BrainThread.cpp`-ban a a konstruktornak átadjuk paraméterként a szélességet és a magasságot. Itt a Hero lesz a karakter, azaz jelen esetünkben 5 karakter van megadva, ezek fognak tehát majd megjelenni először a képernyőn. Ezeket a heroes nevű verembe pakoljuk be a `push.back()` függvénnyel, melynek paraméterként adjuk át az általunk kreált Hero-k neveit egyenként.

```
BrainBThread::BrainBThread (int w, int h)
{
 dispShift = heroRectSize+heroRectSize/2;

 this->w = w - 3 * heroRectSize;
 this->h = h - 3 * heroRectSize;

 std::srand (std::time (0));

 Hero me (this->w / 2 + 200.0 * std::rand() / (RAND_MAX + 1.0) - 100,
 this->h / 2 + 200.0 * std::rand() / (RAND_MAX + 1.0) - 100, 255.0 * std::rand() / (RAND_MAX + 1.0), 9);

 Hero other1 (this->w / 2 + 200.0 * std::rand() / (RAND_MAX + 1.0) - 100,
 this->h / 2 + 200.0 * std::rand() / (RAND_MAX + 1.0) - 100, 255.0 * std::rand() / (RAND_MAX + 1.0), 9);
```

```

 5, "Norbi Entropy");
Hero other2 (this->w / 2 + 200.0 * std::rand() / (RAND_MAX + 1.0 ←
) - 100,
 this->h / 2 + 200.0 * std::rand() / (RAND_MAX + 1.0 ←
) - 100, 255.0 * std::rand() / (RAND_MAX + 1.0), ←
 3, "Greta Entropy");
Hero other4 (this->w / 2 + 200.0 * std::rand() / (RAND_MAX + 1.0 ←
) - 100,
 this->h / 2 + 200.0 * std::rand() / (RAND_MAX + 1.0 ←
) - 100, 255.0 * std::rand() / (RAND_MAX + 1.0), ←
 5, "Nandi Entropy");
Hero other5 (this->w / 2 + 200.0 * std::rand() / (RAND_MAX + 1.0 ←
) - 100,
 this->h / 2 + 200.0 * std::rand() / (RAND_MAX + 1.0 ←
) - 100, 255.0 * std::rand() / (RAND_MAX + 1.0), ←
 7, "Matyi Entropy");

heroes.push_back (me);
heroes.push_back (other1);
heroes.push_back (other2);
heroes.push_back (other4);
heroes.push_back (other5);
}

```

A dobozok mozgatásának kiszámítása a `BrainbThread.h` fájlban van megadva.

```

void move (int maxx, int maxy, int env) {

 int newx = x+ (((double) agility*1.0) * (double) (std::rand ←
) / (RAND_MAX+1.0))-agility/2);
 if (newx-env > 0 && newx+env < maxx) {
 x = newx;
 }
 int newy = y+ (((double) agility*1.0) * (double) (std::rand ←
) / (RAND_MAX+1.0))-agility/2);
 if (newy-env > 0 && newy+env < maxy) {
 y = newy;
 }
}

```

## 8. fejezet

# Helló, Schwarzenegger!

### 8.1. Szoftmax Py MNIST

Python

Felhasznált irodalom:

- [BIOINTELLIGENCE]
- [NEURALIS]

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/blob/master/attention\\_raising/Source/Schwarzenegger/-szoftmax.py](https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/Schwarzenegger/-szoftmax.py)



#### Megjegyzés

A feladatban tutoráltam Fürjes-Beke Pétert

---

Ebben a feladatban egy olyan programot vizsgálunk meg, amely egy adott 28x28 méretű kép alapján meg-tippeli, hogy hanyas szám látható rajta. Ezt egy mesterséges neurális hálók segítségével éri el. Ezek mű-ködése egy természetbeli folyamat modellezésén alapulnak: az emberi agyhoz tartozó neuronok és azok folyamatait utánozzák. A neurális hálózatok működése általában két fázisból áll. Az első a tanulási fázis, ami két további nagy csoportba sorolható: felügyelt és felügyelet nélküli tanulás. Felügyelt tanulás során valamilyen külső segítség, mint például betanító személy vesz részt a folyamatban, hogy a háló által adott kimenet és az elvárt érték közötti különbség alapján módosítsa a súlyokat. A felügyelet nélküli módszer esetén pedig a háló meglévő adatok vagy minták alapján dolgozik és tanul tovább. A második fázis az elő-hívási fázis, ahol az első szakaszban kapott információ-feldolgozó rendszert használjuk fel. Ez a folyamat az tanulási fázishoz képest gyorsan lefut.

Jelen példánkban a program futtatásához szükségünk lesz a Python magas szintű programozási nyelvre és a Tensorflow nyílt forráskódú Python csomagra. A Python alapjait 1989-ben tették le és az egyik legnépsze-rűbb programozási nyelv. Előnye, hogy más népszerű programnyelvekben (C, JAVA, stb.) megírt ugyanaz a program Pythonban sokkal rövidebb, ezért a fejlesztési idő is kevesebb. Felhasználása széleskörű és in-gyenes. Rengeteg feladat megoldásához használható fel: tudományos alkalmazások, webfejlesztés, pár-

---

vagy akár több sorból álló projektek, üzleti- és hálózati alkalmazások is. Ez a programnyelv rengeteg beépített kiegészítő csomaggal és szolgáltatással rendelkezik, mint például a már említett TensorFlow. Ez a Python csomag numerikus számításokat végez gráfok használatával. Ez nagy segítséget jelent a neurális hálózatok felhasználása során, ahol a gráf csomópontjai a neuronok lesznek, és a kapcsolatokat pedig a gráf élei reprezentálják, ezzel létrehozva egy rugalmas szerkezetet. A TensorFlow segítségével készített programokat akár mobiltelefonon is lehet futtatni, úgy hogy kihasználja a rendelkezésre álló processzorokat és grafikus kártyát. A TensorFlow csomagot a Google Brain csapata alkotta meg, kezdetben azzal a céllal, hogy kutassák a gépi tanulást és a neurális hálókat. Az elkészített program azonban más területen is felhasználhatóvá vált, ezért egy javított változattal, ahol már pontosabb neurális hálózatokat tudtak elkészíteni, létrejött a DistBelief szabadalmaztatott géptanuló rendszer. Ennek a programnak a használata annyira elterjedt a cégen belül, hogy a Google informatikusai egyszerűsítették, ezzel gyorsabbá téve a csomagot, végül ebből lett a TensorFlow.

Ahhoz, hogy Ubuntu rendszerünkön használni tudjuk a TensorFlow-t, először telepítenünk kell a gépünkre pip python alapú csomagkezelő rendszert, a python 3-ast és a matplotlib grafikonrajzoló és képmegjelenítőt. Ezeket az alábbi parancsok terminálba történő beírásával tehetjük meg:

```
sudo apt update
sudo apt install python3-dev python3-pip
sudo apt-get install python3-matplotlib
sudo pip3 install -U virtualenv
```

Ha ezzel megvagyunk, akkor a virtuális környezetben fogjuk feltelepíteni a TensorFlow csomagot:

```
virtualenv --system-site-packages -p python3
source ./venv/bin/activate
(venv) $ pip install --upgrade pip
(venv) $ pip install --upgrade tensorflow
(venv) $ python -c "import tensorflow as tf; tf.enable_eager_execution(); print(tf.reduce_sum(tf.random_normal([1000, 1000])))"
```

Ezután ebben a virtuális környezetben `cd` parancssal megkeressük a `softmax.py` fájl helyét és a python `softmax.py` beírásával történik a program elindítása. Most nézzük meg magát a programot. Először importáljuk a szükséges könyvtárakat az `import` segítségével:

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse

from tensorflow.examples.tutorials.mnist import input_data

import tensorflow as tf
old_v = tf.logging.get_verbosity()
tf.logging.set_verbosity(tf.logging.ERROR)

import matplotlib.pyplot

FLAGS = None
```



Következő lépésként a változókat deklaráljuk. Itt az  $x$  változónak azt adjuk meg, hogy a képek adatátalmánya hány dimenziós vektor legyen. Jelen esetben 784 dimenzó lesz, mert a képünk mérete  $28 \times 28$ , és ahhoz, hogy bármennyi képet olvashassunk be, meg kell adni, hogy "None". A  $W$  és a  $b$  változóban a `tf.Variable()` függvénnyel súlyokat és bias értékeket adunk meg. Ezek kezdeti értéke most nulla, azaz zeros lesz. Az  $y$  változóban lesz a modellünk által megjósolt adat, az  $y_$  pedig egy újabb placeholder lesz, amit majd a `cross_entropy` változóba implementáljuk. Itt fogjuk megmérni, hogy a jóslatunk mennyire elégtelen a valóságos helyzet leírására. Ahhoz, hogy csökkentsük a hibahatárt, a `tf.train.GradientDescentOptimizer` és a `minimize` függvényeket használjuk. Az elsővel adjuk meg a csökkentés mértékét, az utóbbival pedig, annak a változónak a nevét, amit csökkenteni szeretnénk. Ez a `train_step` változóban lesznek megadva.

```
def main(_):
 mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)

 # Create the model
 x = tf.placeholder(tf.float32, [None, 784])
 W = tf.Variable(tf.zeros([784, 10]))
 b = tf.Variable(tf.zeros([10]))
 y = tf.matmul(x, W) + b

 # Define loss and optimizer
 y_ = tf.placeholder(tf.float32, [None, 10])

 cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(↵
 labels = y_, logits = y))
 train_step = tf.train.GradientDescentOptimizer(0.5).minimize(↵
 cross_entropy)

 sess = tf.InteractiveSession()
```

A `for i in range(1000)` azt jelenti, hogy for ciklusban 1000-szer futtatjuk le, azaz a tanítás ennyiszer fog lefutni. Majd kiíratjuk, hogy a modellünk milyen értéket adott meg, közben a képet is látjuk, így összehasonlíthatjuk az adatokat. Ha bezárjuk a képet, látni fogunk újabb képet, aminek adatait szintén összehasonlíthatjuk.

```
Train
tf.initialize_all_variables().run(session=sess)
print("-- A halozat tanitasa")
for i in range(1000):
 batch_xs, batch_ys = mnist.train.next_batch(100)
 sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
 if i % 100 == 0:
 print(i/10, "%")
print("-----")

Test trained model
print("-- A halozat tesztelese")
```

```
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
print("-- Pontosság: ", sess.run(accuracy, feed_dict={x: mnist.test. ←
 images,
 y_: mnist.test.labels}))
print("-----")

print("-- A MNIST 42. tesztkepenek felismerese, mutatom a szamot, a ←
 tovabblepeshez csukd be az ablakat")

img = mnist.test.images[42]
image = img

matplotlib.pyplot.imshow(image.reshape(28, 28), cmap=matplotlib.pyplot.cm ←
 .binary)
matplotlib.pyplot.savefig("4.png")
matplotlib.pyplot.show()

classification = sess.run(tf.argmax(y, 1), feed_dict={x: [image]})

print("-- Ezt a halozat ennek ismeri fel: ", classification[0])
print("-----")

print("-- A MNIST 11. tesztkepenek felismerese, mutatom a szamot, a ←
 tovabblepeshez csukd be az ablakat")

img = mnist.test.images[11]
image = img
matplotlib.pyplot.imshow(image.reshape(28,28), cmap=matplotlib.pyplot.cm. ←
 binary)
matplotlib.pyplot.savefig("8.png")
matplotlib.pyplot.show()

classification = sess.run(tf.argmax(y, 1), feed_dict={x: [image]})

print("-- Ezt a halozat ennek ismeri fel: ", classification[0])
print("-----")

if __name__ == '__main__':
 parser = argparse.ArgumentParser()
 parser.add_argument('--data_dir', type=str, default='/tmp/tensorflow/ ←
 mnist/input_data',
 help='Directory for storing input data')
 FLAGS = parser.parse_args()
 tf.app.run()
```

Az alábbi képernyőmentés mutatja, mi történik a program futtatása után. Ezek szerint a hálózatunk tesztelése 0,9205 pontosságú lett és fel is ismerte a képen lévő számokat, hiba nélkül.

```
(venv) davee@davee-K501UB:~/Dokumentumok/bhax/attention_raising/Source/Schwarzenegger$ python szoftmax.py
Extracting /tmp/tensorflow/mnist/input_data/train-images-idx3-ubyte.gz
Extracting /tmp/tensorflow/mnist/input_data/train-labels-idx1-ubyte.gz
Extracting /tmp/tensorflow/mnist/input_data/t10k-images-idx3-ubyte.gz
Extracting /tmp/tensorflow/mnist/input_data/t10k-labels-idx1-ubyte.gz
2019-04-21 14:13:58.530077: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2 FMA
2019-04-21 14:13:58.533645: I tensorflow/core/platform/profile_utils/cpu_utils.cc:94] CPU Frequency: 2400000000 Hz
2019-04-21 14:13:58.534026: I tensorflow/compiler/xla/service/service.cc:150] XLA service 0x5618814fbd80 executing computations on platform Host. Devices:
2019-04-21 14:13:58.534048: I tensorflow/compiler/xla/service/service.cc:158] StreamExecutor device (0): <undefined>, <undefined>
-- A halozat tanitasa
0.0 %
10.0 %
20.0 %
30.0 %
40.0 %
50.0 %
60.0 %
70.0 %
80.0 %
90.0 %

-- A halozat tesztelese
-- Pontossag: 0.9205

-- A MNIST 42. tesztkepenek felismerese, mutatom a szamot, a tovabblepeshez csukd be az ablakat
-- Ezt a halozat ennek ismeri fel: 4

-- A MNIST 11. tesztkepenek felismerese, mutatom a szamot, a tovabblepeshez csukd be az ablakat
-- Ezt a halozat ennek ismeri fel: 6

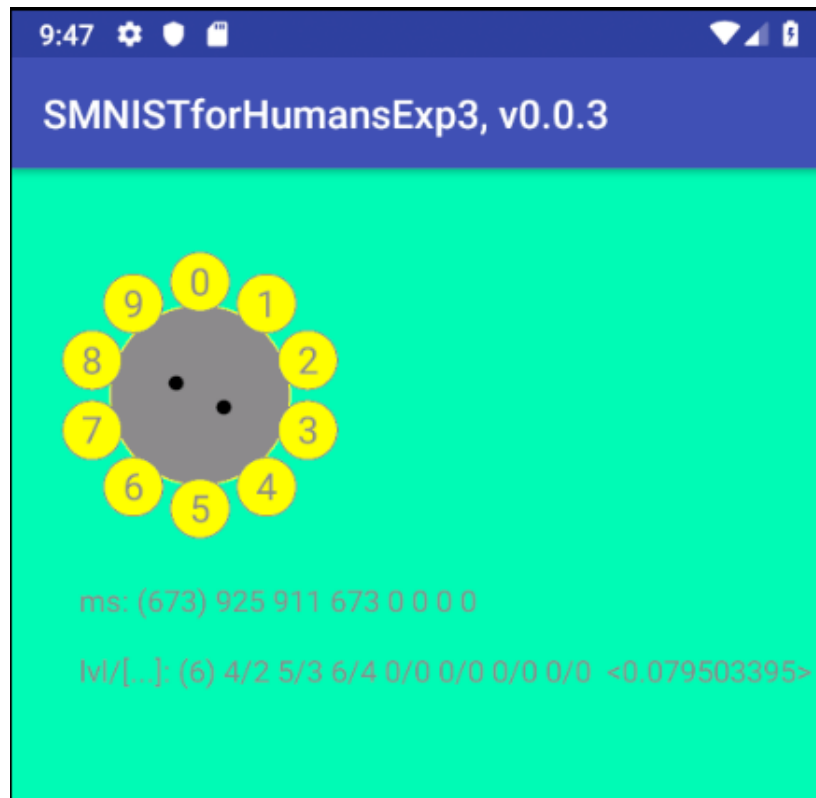
```

8.1. ábra. Hálózat tesztelése

## 8.2. Mély MNIST

Python

Ennél a feladatnál fel szeretném használni a passzolási lehetőséget, amit az MNIST program minimum level 6 szint elérésével lehetett kapni. Az általam elért eredményt az alábbi kép tartalmazza:



8.2. ábra. SMNIST Eredmény

### 8.3. Minecraft-MALMÖ

Megoldás videó: <https://youtu.be/bAPSu3Rndi8>

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/blob/master/attention\\_raising/Source/Schwarzenegger/tutorial\\_1.py](https://gitlab.com/davidhalasz/bhax/blob/master/attention_raising/Source/Schwarzenegger/tutorial_1.py)

A Minecraft Malmo program telepítéséhez szükségünk lesz egy pre-build csomagra, amit a <https://github.com/Microsoft/malmo/releases> oldalról tölthetjük le. A kicsomagolás után lépünk be a Minecraft mappába és futtassuk a launchClient.sh fájlt. Ez a folyamat egy kis időbe fog telni, de 95%-nál elindul a program és ennél tovább nem fog növekedni. Ha ezzel megvagyunk lépünk be a Python\_Examples mappába és futtassuk a tutorial\_1.py python fájlt.

```
cd Minecraft
./launchClient.sh

A python fájl futtatása
cd Python_Examples
python3 tutorial_1.py
```

A tutorial\_1.py fájlt tetszőlegesen szerkeszthetjük. Ha a programon belül megkeressük a #loop until mission ends kommentet és az alatta lévő while ciklusban ágenseket adunk meg a agent\_host.sendCom függvénnyel, akkor a tutorial\_1.py program újboli futtatása során amíg le nem jár a megadott idő, végrehajtja az általunk megadott utasításokat, jelen esetben azt, hogy mindig lépjen előre egyet, ugorjon és támadjon.

```
Loop until mission ends:
while world_state.is_mission_running:
 print(".", end="")

 agent_host.sendCommand("move 1")
 agent_host.sendCommand("jump 1")
 agent_host.sendCommand("attack 1")
 time.sleep(0.1)
 world_state = agent_host.getWorldState()
 for error in world_state.errors:
 print("Error:", error.text)

print()
print("Mission ended")
Mission has ended.
```

A Minecraft élővilágát (bionome) a missionXML részben tudjuk megadni. Ha szeretnénk saját magunk szerkeszteni, akkor használhatjuk a <https://www.minecraft101.net/superflat/> oldalt, amit majd be kell másolnunk a FlatWorldGenerator sorba. Továbbá ennél a kódcsipetnél adjuk meg az időt is, jelen esetben az időlimit 360000 ms lesz. Ahhoz, hogy mindez meg legyen jelenítve a programban, meg kell adnunk hogy: `my_mission = MalmöPython.MissionSpec(missionXML, True)`.

```
missionXML='''<?xml version="1.0" encoding="UTF-8" standalone="no" ↵
?>
<Mission xmlns="http://ProjectMalmo.microsoft.com" xmlns:xsi=" ↵
http://www.w3.org/2001/XMLSchema-instance">

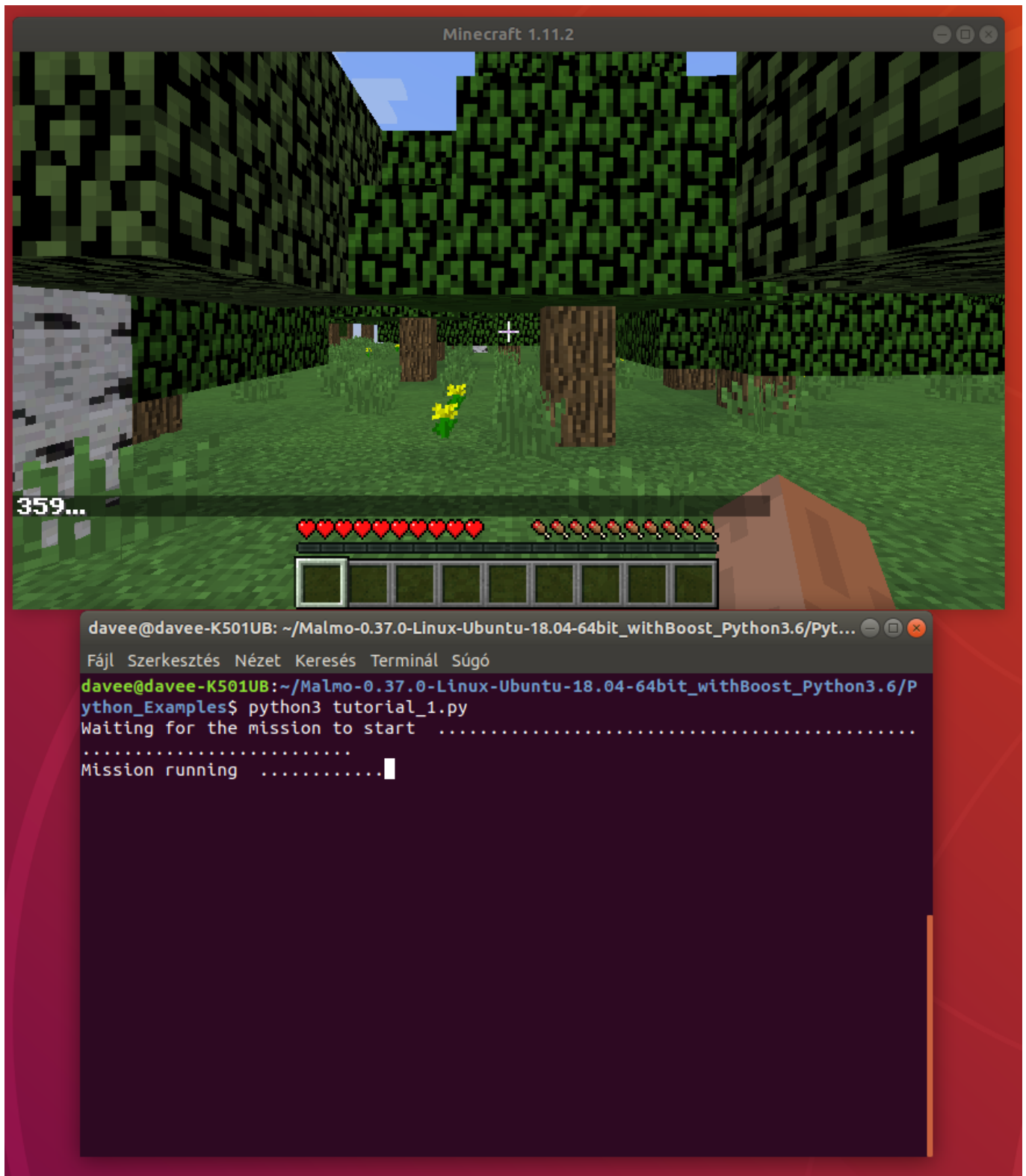
<About>
<Summary>gitlab.com/whoisZORZ</Summary>
</About>

<ServerSection>
<ServerHandlers>
<FlatWorldGenerator generatorString="3;1*minecraft: ↵
bedrock,7*minecraft:dirt,1*minecraft:grass;4;village, ↵
decoration,lake"/>
<DrawingDecorator>
<DrawSphere x="-27" y="70" z="0" radius="30" type="air"/>
</DrawingDecorator>
<ServerQuitFromTimeUp timeLimitMs="360000"/>
<ServerQuitWhenAnyAgentFinishes/>
</ServerHandlers>
</ServerSection>

<AgentSection mode="Survival">
<Name>ZORZBot</Name>
<AgentStart/>
<AgentHandlers>
<ObservationFromFullStats/>
```

```
 <ContinuousMovementCommands turnSpeedDegs="180"/>
 </AgentHandlers>
</AgentSection>
</Mission>'''

my_mission = MalmoPython.MissionSpec(missionXML, True)
```



8.3. ábra. Minecraft

## 9. fejezet

# Helló, Chaitin!

### 9.1. Iteratív és rekurzív faktoriális Lisp-ben

Megoldás videó: <https://youtu.be/z6NJE2a1zIA>

A LISP 1950-es évek második felében jött létre. Ez egy olyan interpreteres, interaktív nyelvi rendszer, amelyet beépített függvények alkotnak. Az évek során számos nyelvjárással gazdagodott. A legelterjedtebb változatai a Common Lisp és a Scheme nyelvek. Ebben a feladatban most ennek a nyelvnek a segítségével írunk egy programot, ami kiírja egy adott szám faktoriális értékét. A matematikában az  $n$  szám faktoriálisának nevezzük az első  $n$  pozitív egész szám szorzatát. A programozási nyelvben ez a függvény megoldható iteratív és rekurzív módon is. A feladat megoldásához szükségünk lesz a GNU képszerkesztő programra, amit az Ubuntu szoftverletöltő központból tölthető le. Miután elindítottuk a programot, az ablak felső menüsorából válasszuk ki az Szűrők -> Script-Fu -> Konzol menüpontot. Most vizsgáljuk meg először iteratíván a programot. Ez a program azért iteratív, mert egy számlálót helyezünk el és ciklusonként folyamatosan megismétljük a folyamatot, egészen addig, amíg el nem éri az adott  $n$  számot. A nyelvben nincs különbség kifejezések és utasítások között, így mindent kifejezés formájában írunk le. Ha megfigyeljük az alábbi kódot, akkor szembetűnő lehet számunkra a rengeteg zárójel. Ez azért van, mert a LISP-ben egy listát zárójelekkel határolunk. A define segítségével elnevezzük a függvényt factorial-nak és a mellette lévő  $n$  pedig majd a program futtatása során egy általunk megadott szám lesz. Másik szokatlan dolog még, hogy a számtani műveletek megadása során prefix jelölést használunk, azaz jelen esetben a  $>$  jel mindig az első és utána jönnek a számok. Ha bemásoltuk a programot a Script-fu-ba és beírjuk még, hogy (factorial 4), akkor megkapjuk az eredményt.

```
(define (factorial n)
 (define (iter product counter)
 (if (> counter n)
 product
 (iter (* counter product) (+ counter 1))))
 (iter 1 1))
```

Most vizsgáljuk meg rekurzívan. Egy függvényt akkor nevezünk rekurzívnak, ha a függvény törzsében újra meghívjuk magát a függvényt. A rekurzív faktoriális megoldásához írjuk be, hogy: (define (fakt n) (if (< n 1) 1 (\* n(fakt(- n 1 ))))) Jelen esetben a függvényhívás neve (fakt n) lesz. Ezután if-el megvizsgáljuk, hogy a beírt szám nagyobb-e 1-nél. Ha igen, akkor írassuk ki az egyest,



ellenkező esetben végezzük el rekurzívan a faktoriális számítást. Így tehát ha beírjuk, hogy `(fakt 4)` akkor megkapjuk eredményül a 24-et.

```
> (define (fakt n) (if (< n 1) 1 (* n(fakt(- n 1)))))
fakt
> (fakt 4)
24
```

9.1. ábra. Króm effekt

## 9.2. Gimp Scheme Script-fu: króm effekt

Írj olyan script-fu kiterjesztést a GIMP programhoz, amely megvalósítja a króm effektet egy bemenő szövegre!

Megoldás videó: [https://youtu.be/OKdAkI\\_c7Sc](https://youtu.be/OKdAkI_c7Sc)

Megoldás forrása: [https://gitlab.com/nbatfai/bhax/tree/master/attention\\_raising/GIMP\\_Lisp/Chrome](https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/GIMP_Lisp/Chrome)

Ebben a feladatban olyan script-fu kiterjesztésről lesz szó, amivel a szövegre egy króm effektet nyomunk. Ahhoz, hogy használni tudjuk a `.scm` végződésű fájlokat a GIMP programban, először át kell másolnunk a GIMP saját scripts mappájába. Ha ezzel megvagyunk, akkor már csak el kell indítanunk a GNU programot, és a menüsorból válasszuk ki a Fájl/létrehozás/BHAX menüpontokat. Itt lesznek majd felsorolva az általunk átmásolt szkriptek. Ha valamiért mégse működne a szkript, vagy hibaüzenetet dob ki, akkor az azért lehet mert nem a megfelelő GIMP programverziót használjuk. Nekünk mindenképpen a GIMP 2.10-es verzióra van szükségünk, ezt egyszerűen a `snap install gimp` parancssor terminálba történő beírásával telepíthetjük.

Mi most nézzük meg a `Chrome3_border2` szkriptet. Az elindítás után megjelenik egy ablak, ahol megadhatjuk a szöveget, a króm effektet, színét és méretét. Ezután a fájl/exportálás menüponttal lementhetjük az elkészült képet:



A program felépítését általában 3 részre lehet osztani: A fő függvény, ami a munkát végzi, az alap beállítások megadása és végül hogy hol legyen elérhető a kiegészítő modul a programon belül. A script elemek definiálása során a script-fu-bhax-chrome-border nevet adjuk, melynek paraméterei a sorba a következő: text (szöveg), font (betűtípus), fontsize (betűméret), width (hosszúság), height (magasság), new width (új hosszúság), color (szín), gradient (átmenet), border-size (szegély mérete). Itt deklaráljuk tehát a lisp programozási nyelvvel az adatokat. A GIMP saját könyvtárait használjuk arra, hogy az képezelő adatokat megadhassuk. Ez a függvényhívás mindig gimp szóval kezdődik, mint például: gimp-text-get-extents-fontname

```
//SZKRIPT ELEMÉK DEFINIÁLÁSA
```

```
(define (color-curve)
 (let* (
 (tomb (cons-array 8 'byte))
)
 (aset tomb 0 0)
 (aset tomb 1 0)
 (aset tomb 2 50)
 (aset tomb 3 190)
 (aset tomb 4 110)
 (aset tomb 5 20)
 (aset tomb 6 200)
 (aset tomb 7 190)
 tomb)
)

; (color-curve)

(define (elem x lista)
 (if (= x 1) (car lista) (elem (- x 1) (cdr lista)))
)

(define (text-wh text font fontsize)
 (let*
 (
 (text-width 1)
 (text-height 1)
)

 (set! text-width (car (gimp-text-get-extents-fontname text fontsize ↵
 PIXELS font)))
 (set! text-height (elem 2 (gimp-text-get-extents-fontname text ↵
 fontsize PIXELS font)))

 (list text-width text-height)
)
)

; (text-width "alma" "Sans" 100)
```

```
(define (script-fu-bhax-chrome-border text font fontsize width height new- ↵
 width color gradient border-size)
(let*
 (
 (text-width (car (text-wh text font fontsize)))
 (text-height (elem 2 (text-wh text font fontsize)))
 (image (car (gimp-image-new width (+ height (/ text-height 2)) 0)))
 (layer (car (gimp-layer-new image width (+ height (/ text-height 2) ↵
) RGB-IMAGE "bg" 100 LAYER-MODE-NORMAL-LEGACY)))
 (textfs)
 (layer2)
)

 (gimp-image-insert-layer image layer 0 0)

 (gimp-image-select-rectangle image CHANNEL-OP-ADD 0 (/ text-height 2) ↵
 width height)
 (gimp-context-set-foreground '(255 255 255))
 (gimp-drawable-edit-fill layer FILL-FOREGROUND)

 (gimp-image-select-rectangle image CHANNEL-OP-REPLACE border-size (+ (/ ↵
 text-height 2) border-size) (- width (* border-size 2)) (- height ↵
 (* border-size 2)))
 (gimp-context-set-foreground '(0 0 0))
 (gimp-drawable-edit-fill layer FILL-FOREGROUND)

 (gimp-image-select-rectangle image CHANNEL-OP-REPLACE (* border-size 3) ↵
 0 text-width text-height)
 (gimp-drawable-edit-fill layer FILL-FOREGROUND)

 (gimp-selection-none image)

;step 1
(gimp-context-set-foreground '(255 255 255))

(set! textfs (car (gimp-text-layer-new image text font fontsize PIXELS) ↵
))
(gimp-image-insert-layer image textfs 0 0)
(gimp-layer-set-offsets textfs (* border-size 3) 0)

(set! layer (car (gimp-image-merge-down image textfs CLIP-TO-BOTTOM- ↵
 LAYER)))

;step 2
(plug-in-gauss-iir RUN-INTERACTIVE image layer 25 TRUE TRUE)

;step 3
(gimp-drawable-levels layer HISTOGRAM-VALUE .18 .38 TRUE 1 0 1 TRUE)
```

```

;step 4
(plug-in-gauss-iir RUN-INTERACTIVE image layer 2 TRUE TRUE)

;step 5
(gimp-image-select-color image CHANNEL-OP-REPLACE layer '(0 0 0))
(gimp-selection-invert image)

;step 6
(set! layer2 (car (gimp-layer-new image width (+ height (/ text-height 2)) RGB-IMAGE "2" 100 LAYER-MODE-NORMAL-LEGACY)))
(gimp-image-insert-layer image layer2 0 0)

;step 7
(gimp-context-set-gradient gradient)
(gimp-edit-blend layer2 BLEND-CUSTOM LAYER-MODE-NORMAL-LEGACY GRADIENT-
 LINEAR 100 0 REPEAT-NONE
 FALSE TRUE 5 .1 TRUE width 0 width (+ height (/ text-height 2)))

;step 8
(plug-in-bump-map RUN-NONINTERACTIVE image layer2 layer 120 25 7 5 5 0
 0 TRUE FALSE 2)

;step 9
(gimp-curves-spline layer2 HISTOGRAM-VALUE 8 (color-curve))

(gimp-image-scale image new-width (/ (* new-width (+ height (/ text-
 height 2))) width))

(gimp-display-new image)
(gimp-image-clean-all image)
)
)

```

A `script-fu-bhax-chrome-border`-nek megadunk egy alap beállítást, azaz paramétereit, például a szöveg legyen most **"PROGRAMOZÁS"**, a betűstílus Sans, a betűméret 160px és így tovább.

```

;(script-fu-bhax-chrome-border "PROGRAMOZÁS" "Sans" 160 1920 1080 400 '(255
 0 0) "Crown molding" 7)

```

Ahhoz, hogy a GIMP porgramon belül elő is tudjuk hívni a scriptet, regisztrálnunk kell a modult. Ehhez a `script-fu-menu-register` függvényt hívjuk segítségül. 2 paramétert kell átadni neki: az egyik a script neve, amit már fentebb definiáltunk és hogy melyik menüpontból érhesük majd el a programon belül.

```

//SCRIPT HELYÉNEK MEGADÁSA
(script-fu-menu-register "script-fu-bhax-chrome-border"
 "<Image>/File/Create/BHAX"
)

```

### 9.3. Gimp Scheme Script-fu: név mandala

Írj olyan script-fu kiterjesztést a GIMP programhoz, amely név-mandalát készít a bemenő szövegből!

Megoldás videó: [https://bhaxor.blog.hu/2019/01/10/a\\_gimp\\_lisp\\_hackelese\\_a\\_scheme\\_programozasi\\_nyelv](https://bhaxor.blog.hu/2019/01/10/a_gimp_lisp_hackelese_a_scheme_programozasi_nyelv)

Megoldás forrása: [https://gitlab.com/nbatfai/bhax/tree/master/attention\\_raising/GIMP\\_Lisp/Mandala](https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/GIMP_Lisp/Mandala)

Ebben a feladatban olyan programot vizsgálunk meg, amelynek segítségével egy mandalát rajzolunk a Gimp-ben.

A `let*` egy beépített kulcsszó, ennek segítségével lehet lokális változókat megadni. Például az alábbi kódcsipetben definiáltuk a `text-width` változót, ahol majd a szöveg méretét adjuk meg.

```
(define (elem x lista)

 (if (= x 1) (car lista) (elem (- x 1) (cdr lista)))

)

(define (text-width text font fontsize)
(let*
 (
 (text-width 1)
)
 (set! text-width (car (gimp-text-get-extents-fontname text fontsize ↵
 PIXELS font)))

 text-width
)
)

(define (text-wh text font fontsize)
(let*
 (
 (text-width 1)
 (text-height 1)
)
 ;;;
 (set! text-width (car (gimp-text-get-extents-fontname text fontsize ↵
 PIXELS font)))
 ;;; ved ki a lista 2. elemét
 (set! text-height (elem 2 (gimp-text-get-extents-fontname text ↵
 fontsize PIXELS font)))
 ;;;

 (list text-width text-height)
)
)
```

A fő függvénynek a `script-fu-bhax-mandala` nevet adtuk, aminek 8 paramétere van. Itt is lokális változókat adunk meg. Az első változónk neve `image` lesz és ebben a kódsorban a `car` azt mondja, hogy az utána következő listában a kiveszi az első elemet. Fontos, hogy itt az első elem nem a `gimp-image-new` lesz, mert ez egy függvényhívás, ezért az utána következő elemet veszi ki.

```
(define (script-fu-bhax-mandala text text2 font fontsize width height color ←
 gradient)
 (let*
 (
 (image (car (gimp-image-new width height 0)))
 (layer (car (gimp-layer-new image width height RGB-IMAGE "bg" 100 ←
 LAYER-MODE-NORMAL-LEGACY)))
 (textfs)
 (text-layer)
 (text-width (text-width text font fontsize))
 ;;;
 (text2-width (car (text-wh text2 font fontsize)))
 (text2-height (elem 2 (text-wh text2 font fontsize)))
 ;;;
 (textfs-width)
 (textfs-height)
 (gradient-layer)
)

 (gimp-image-insert-layer image layer 0 0)

 (gimp-context-set-foreground '(0 255 0))
 (gimp-drawable-fill layer FILL-FOREGROUND)
 (gimp-image-undo-disable image)

 (gimp-context-set-foreground color)

 (set! textfs (car (gimp-text-layer-new image text font fontsize PIXELS) ←
))
 (gimp-image-insert-layer image textfs 0 -1)
 (gimp-layer-set-offsets textfs (- (/ width 2) (/ text-width 2)) (/ ←
 height 2))
 (gimp-layer-resize-to-image-size textfs)

 (set! text-layer (car (gimp-layer-new-from-drawable textfs image)))
 (gimp-image-insert-layer image text-layer 0 -1)
 (gimp-item-transform-rotate-simple text-layer ROTATE-180 TRUE 0 0)
 (set! textfs (car (gimp-image-merge-down image text-layer CLIP-TO-BOTTOM ←
 -LAYER)))

 (set! text-layer (car (gimp-layer-new-from-drawable textfs image)))
 (gimp-image-insert-layer image text-layer 0 -1)
 (gimp-item-transform-rotate text-layer (/ *pi* 2) TRUE 0 0)
 (set! textfs (car (gimp-image-merge-down image text-layer CLIP-TO-BOTTOM ←
 -LAYER)))
```

```
(set! text-layer (car (gimp-layer-new-from-drawable textfs image)))
(gimp-image-insert-layer image text-layer 0 -1)
(gimp-item-transform-rotate text-layer (/ *pi* 4) TRUE 0 0)
(set! textfs (car(gimp-image-merge-down image text-layer CLIP-TO-BOTTOM ↔
-LAYER)))

(set! text-layer (car (gimp-layer-new-from-drawable textfs image)))
(gimp-image-insert-layer image text-layer 0 -1)
(gimp-item-transform-rotate text-layer (/ *pi* 6) TRUE 0 0)
(set! textfs (car(gimp-image-merge-down image text-layer CLIP-TO-BOTTOM ↔
-LAYER)))

(plug-in-autocrop-layer RUN-NONINTERACTIVE image textfs)
(set! textfs-width (+ (car(gimp-drawable-width textfs)) 100))
(set! textfs-height (+ (car(gimp-drawable-height textfs)) 100))

(gimp-layer-resize-to-image-size textfs)

(gimp-image-select-ellipse image CHANNEL-OP-REPLACE (- (- (/ width 2) ↔
(/ textfs-width 2)) 18)
(- (- (/ height 2) (/ textfs-height 2)) 18) (+ textfs-width 36) (+ ↔
textfs-height 36))
(plug-in-sel2path RUN-NONINTERACTIVE image textfs)

(gimp-context-set-brush-size 22)
(gimp-edit-stroke textfs)

(set! textfs-width (- textfs-width 70))
(set! textfs-height (- textfs-height 70))

(gimp-image-select-ellipse image CHANNEL-OP-REPLACE (- (- (/ width 2) ↔
(/ textfs-width 2)) 18)
(- (- (/ height 2) (/ textfs-height 2)) 18) (+ textfs-width 36) (+ ↔
textfs-height 36))
(plug-in-sel2path RUN-NONINTERACTIVE image textfs)

(gimp-context-set-brush-size 8)
(gimp-edit-stroke textfs)

(set! gradient-layer (car (gimp-layer-new image width height RGB-IMAGE ↔
"gradient" 100 LAYER-MODE-NORMAL-LEGACY)))

(gimp-image-insert-layer image gradient-layer 0 -1)
(gimp-image-select-item image CHANNEL-OP-REPLACE textfs)
(gimp-context-set-gradient gradient)
(gimp-edit-blend gradient-layer BLEND-CUSTOM LAYER-MODE-NORMAL-LEGACY ↔
GRADIENT-RADIAL 100 0
REPEAT-TRIANGULAR FALSE TRUE 5 .1 TRUE (/ width 2) (/ height 2) (+ (+ (/ ↔
width 2) (/ textfs-width 2)) 8) (/ height 2))
```

```
(plug-in-sel2path RUN-NONINTERACTIVE image textfs)

 (set! textfs (car (gimp-text-layer-new image text2 font fontsize PIXELS ↵
)))
 (gimp-image-insert-layer image textfs 0 -1)
 (gimp-message (number->string text2-height))
 (gimp-layer-set-offsets textfs (- (/ width 2) (/ text2-width 2)) (- (/ ↵
 height 2) (/ text2-height 2)))

 ;(gimp-selection-none image)
 ;(gimp-image-flatten image)

 (gimp-display-new image)
 (gimp-image-clean-all image)
)
)
```

Ahhoz, hogy a kiegészítő modul alapbeállításokkal rendelkezzen, a `script-fu-register`-ben adjuk meg. Például az első paraméter, azaz a Text most "Programozás" lesz, a Font "Sans" és így tovább. A menübe történő implementálás itt is ugyanúgy történik, ahogy az előző említettem.

```
(script-fu-register "script-fu-bhax-mandala"
 "Mandala9"
 "Creates a mandala from a text box."
 "Norbert Bátfai"
 "Copyright 2019, Norbert Bátfai"
 "January 9, 2019"
 ""
 SF-STRING "Text" "Programozás"
 SF-STRING "Text2" "BHAX"
 SF-FONT "Font" "Sans"
 SF-ADJUSTMENT "Font size" '(100 1 1000 1 10 0 1)
 SF-VALUE "Width" "1000"
 SF-VALUE "Height" "1000"
 SF-COLOR "Color" '(255 0 0)
 SF-GRADIENT "Gradient" "Deep Sea"
)
(script-fu-menu-register "script-fu-bhax-mandala"
 "<Image>/File/Create/BHAX"
)
```





9.2. ábra. Mandala

## 10. fejezet

# Helló, Gutenberg!

### 10.1. Programozási alapfogalmak

[?]

1. ALAPFOGALMAK A programozási nyelveknek három szintjét különböztethetjük meg: gépi nyelv, assembly szintű nyelv és magas szintű nyelv. A magas szintű nyelven megírt programot forrásprogramnak nevezzük. Ezek összeállítására vonatkozó formai szabályok összességét pedig szintaktikai szabályoknak. A tartalmi, értelmezési, jelentésbeli szabályok alkotják a szemantikai szabályokat. Ez a két szabály együttese határozza meg a magas szintű programozási nyelvet. Ahhoz, hogy az adott processzor végre tudja hajtani a programokat, szükség lesz fordító programokra vagy interpreterre, amik lefordítják a processzor saját gépi kódú nyelvére. Az a különbség a fordítóprogramok és az interpreterek között, hogy az utóbbi nem készít tárgyprogramot, hanem utasításokként értelmezi a forrásprogramot és rögtön végre is hajtja azokat. Egyébként mindkét program végez lexikális (forrásszöveg feldarabolása lexikális egységekre), szintaktikai (Teljesülnek-e az adott nyelv szintaktikai szabályai?) és szemantikai elemzés. Ahhoz, hogy egy programot helyesen írjunk meg, át kell tanulmányozni az adott programnyelvek szabványait (hivatkozási nyelv). A legnagyobb probléma a programozási nyelvekkel kapcsolatban, az a hordozhatóság, ami az egyik magas szintű programnyelvből egy másik programnyelvre történő implementációja, ugyanis ez a technika jelenleg nem megoldott. A programnyelvek osztályozása háromféleképpen történik: imperatív nyelvek, deklaratív nyelvek és máselví (egyéb) nyelvek. Az imperatív nyelvekhez tartoznak az algoritmikus nyelvek, a program utasítások sorozata. Legfőbb programozói eszköz a változó, amihez értékeket rendelhetünk, manipulálhatjuk az adatokat, tehát a tár közvetlen elérése. Ez a nyelv szorosan kötődik a Neumann-architektúrához. Két alcsoportja van: Eljárásorientált- és objektumorientált nyelvek. Ezzel szemben a deklaratív nyelvek nem algoritmikus nyelvek, nem kötődnek szorosan a Neumann architektúrához. A programozó csak a problémát adja meg, és nincs lehetősége memóriaműveletekre. Alcsoportjai a funkcionális- és logikai nyelvek. A másnyelvű(egyéb) nyelvekhez pedig azok tartoznak, amiket nem lehet besorolni a fenti két nyelv valamelyikébe.

#### 2. Adattípusok

Az adattípus a programozási nyelvekben egy absztrakt programozási eszköz. A típuslétrehozás úgy történik, hogy megadjuk a tartományát, a műveleteit és a reprezentációját. Vannak esetek, amikor a saját típust a beépített és a már korábban definiált saját típusok segítségével adjuk meg. Nem minden programozási nyelv ismeri ezt az eszközt, ezért ennek megfelelően elkülöníthetünk típusos és nem típusos nyelveket. Egy adattípus három dolgot határoz meg: tartomány, műveletek, reprezentáció. A tartomány tartalmazza azokat

az elemeket, amelyeket az adott programozási eszköz felvehet értékként. Egyszerű típus lehet például a C nyelvben int típus, ami az egész számokra vonatkozik. Ott van továbbá a char, azaz a karakteres típus, amelynek elemei karakterek. De létezik még logikai, felsorolásos, sorszámozott típus stb. Az összetett típusokhoz tartoznak például a tömbök, amik lehetnek statikus vagy homogén összetett típus. attól függően, hogy a tömb elemeiben milyen típusú értékek fordulnak elő. Továbbá ide tartozik még a rekord típusú programozási eszköz is. Fontos még a mutató típus, amelyek tartományának elemei lényegében tárcímek, tehát az adott eszköz a tár adott területét címzi. A nevesített konstans olyan programozási eszköz, amelynek három komponense van: név, típus, érték. A C programban a nevesített konstans így kell elképzelni: #define név literál. Lényege, hogy előre definiáljuk a konstansok értékeit és a programban bárhol hivatkozhatunk rá a konstans nevének használatával, így betöltjük az ott lévő adatokat. Továbbá, ha meg akarjuk változtatni az adatot, akkor elég csak a deklarációs részben megváltoztatni, így az egész programban érvényes lesz a megváltoztatott érték. A változónak négy komponense van: név, attribútumok, cím, érték. A név egy azonosító, ahogy a nevesített konstansoknál is van. Az attribútumok pedig a változók futás közbeni viselkedését határozzák meg, például ha típust rendelünk hozzá (int, char stb). A cím a tárnak azt a részét határozza meg, ahol a változó értéke elhelyezkedik. Az értéket a típussal összhangban kell megadni. A C nyelv típusrendszere lehet aritmetikai és származtatott típusok. Az aritmetikai típusok tartományának elemeivel aritmetikai műveletek végezhetők:

- egész típus: int szam = 5
- karakter típus: char szoveg = 'valami'
- felsorolás: enum szinek { VOROS=11, NARANCS=9, SARGA=7, ZOLD=5, KEK=3, IBOLYA=3};
- valós (float, double, long double)

Származtatott típusok lehetnek:

- tömb: int tomb[5]
- mutató: int \*mut
- függvény: fgv()
- struktúra: STRUCT [struktúrátípus\_név] {mező\_deklarációk} [változólista];
- union: UNION [uniontípus\_név] {mező\_deklarációk} [változólista];

### 3. Kifejezések

A kifejezések szintaktikai eszközök. Lényege, hogy ha a programban már ismert értékek segítségével új értékeket határozunk meg. Formálisan operandusokból, operátorokból és kerek zárójelekből állnak. A kifejezésnek három alakja lehet: pre-, in- és postfix. A prefix esetben az operátor az operandusok előtt áll (\* 3 5). Infix-nél az operátor középen (3 \* 5). Postfix-nél pedig az operandusok mögött áll az operátor (3 5 \*). A C egy kifejezésorientált nyelv. Itt a kifejezéseket rekurzívan építjük fel, tehát a kifejezések egyszerű kifejezésekből épülnek fel, zárójelek és operátorok használatával, majd ezt az egészet lexikális egységek zárják le. Az aritmetikai típusoknál a típuskényszerítés elvét vallja, ami azt jelenti, hogy egy két operandusú operátornak különböző típusú operandusai lehetnek, de a műveletek csak azonos típusok között végezhetők

el. Bizonyos esetekben konverzió van, tehát a nyelv megvizsgálja, hogy milyen típuskombinációk megengedettek.

#### 4. Utasítások

Az utasítások olyan program egységek, amelyekkel megadhatjuk az algoritmusok egyes lépéseit, továbbá a fordítóprogramok ezek segítségével generálja a tárgykódot. Két csoportból áll: deklarációs és végrehajtható utasítás. A deklarációs utasításnál nincs tárgykód, nem kerülnek lefordításra. Ez csak a fordítóprogramoktól kérnek valamilyen szolgáltatás, vagy olyan információkat ad át a fordítóprogramnak, amik szükségesek a tárgykód generálásánál. A végrehajtható utasításokból készül el a tárgykód a fordítóprogram által. Ezek tovább csoportosíthatók:

- értékadó utasítás: változó értékének beállítása a program futása során.
- üres utasítás: üres gépi utasítás
- ugró utasítás: feltétel nélküli vezérlés átadó utasítás során egy adott címkével ellátott végrehajtható utasításra adhatjuk át a vezérlést. Használt alakja: GOTO címke
- elágaztató utasítás:

Kétirányú elágaztató utasítás: ebben az esetben a program két tevékenység közül választ és a választott tevékenységet végrehajtja (vagy ellenkezőleg). A feltételes utasítás alakja: IF feltétel THEN tevékenység [ ELSE tevékenység ] Ha nem szerepel az ELSE-ág, akkor hosszú alakról van szó, ellenkező esetben rövid utasítás. Működése úgy zajlik, hogy előbb kiértékelődik a feltétel. Ha ez igaz, akkor az ott lévő tevékenység végrehajtódik. Ha a feltétel hamis és van else ág, akkor az ott lévő feltétel is kiértékelődik. Szót kell még ejteni a „csellengő ELSE” problémáról is, ami akkor fordul elő, amikor egy rövid if utasításba be van ágyazva egy hosszú if utasítás is:

```
IF ... THEN
 IF ... THEN
 ELSE ...
```

A kérdés az, hogy vajon melyik IF feltételhez tartozik az ELSE-ág? A probléma egyszerűen kiküszöbölhető, ha egy hosszú utasításba ágyazzuk be a hosszú utasítást, úgy hogy a külső ELSE-ágba üres utasítás szerepel. De némely programnál nincs szükség erre, ugyanis az implementáció többsége azt mondja, hogy az értelmezés belülről kifelé történik, tehát minden egyes ELSE-ágot belülről kifelé párosítjuk az adott IF utasításokhoz.

Többirányú elágaztató utasítás: Itt a program egymást kölcsönösen kizáró akárhány tevékenység közül egyet választ. C esetében úgy néz ki a dolog, hogy először kiértékelődik a kifejezés, ami egy integer típusú számnak kell lennie. Ez összehasonlításra kerül a CASE ágak értékeivel. Ha van egyezés akkor az adott ponton végrehajtódik a tevékenység. Ha nincs egyezés, akkor a DEFAULT ággal lépünk ki:

```
SWITCH (kifejezés) {
 CASE egész_konstans_kifejezés : [tevékenység]
 [CASE egész_konstans_kifejezés : [tevékenység]]...
 [DEFAULT: tevékenység]
};
```

- ciklusszervező utasítás: Ez az utasítás lehetővé teszi, hogy egy bizonyos tevékenységet akárhányszor megismételjünk. Felépítése: fej, mag, vég. Működés szerint 2 típusa van: az egyik az üres ciklus, amikor a mag egyszer sem fut le, a másik pedig a végtelen ciklus, amikor az ismétlődés soha nem áll le. Kezdőfeltételes ciklus esetében a feltétel a fejben jelenik meg. Tehát Először kiértékelődik a feltétel, Ha igaz, végrehajthó a ciklusmag, majd újra kiértékelődik a feltétel egészen addig, amíg hamis nem lesz. Ilyen a WHILE(feltétel) végrehajthó\_utasítás. Végfeltételes ciklus esetében a feltétel a végben van, de vannak nyelvek amelyekben a fej tartalmazza azt. Tehát először végrehajthó a mag és csak utána vizsgálja meg a feltételt. DO végrehajthó\_utasítás WHILE(feltétel); Előírt lépésszámú ciklus: Itt a fejben van a feltétel. Tartalmaz egy változót (ciklusváltozó) és a változó által felvett értékekre fut le a ciklus. FOR([kifejezés1]; [kifejezés2]; [kifejezés3]) végrehajthó\_utasítás
- hívó utasítás
- vezérlésátadó utasítás
- I/O utasítás
- egyéb utasítások

## 5. Programok szerkezete

Az alprogramok olyan programozási eszközök, amelyeket akkor használunk ha egy programon belül ugyanaz a programrész többször megismétlődik, így nem kell minden esetben újra megírni a programot, hanem elég csak hivatkoznunk az adott alprogramra. A formális alprogram felépítése a következő: fej vagy specifikáció, törzs vagy implementáció és vég. Ez a programozási eszköz négy komponensből áll: név, formális paraméter lista, törzs, környezet. A név lesz az azonosító, tehát mindig ennek használatával tudunk majd hivatkozni. A formális paraméter listában azonosítók szerepelnek, melyek aktuális értékei a hívás helyén szerepelnek. A formális paraméter lista általában a név mellett zárójelben szerepel. Dönthetünk úgy is, hogy nem adunk meg paramétert, mert az adott alprogramnak nincs szüksége rá. Ekkor ez egy paraméter nélküli alprogram lesz. Ez a két komponens az alprogramban mindig a fejben szerepelnek. A törzsben deklarációs és végrehajthó utasításokat adunk meg. Az alprogram környezete a globális változók együttese. Globális nevek alatt azt értjük, hogy az alprogramon kívül neveket deklaráltunk, amik elérhetőek az alprogram számára. Van ugyanis lokális nevek is, amik csak az alprogramban érhető el és a külső környezet számára el van rejtve. Az alprogramnak két típusa van: eljárás és függvény. Az eljárás általában a paraméterek vagy a környezetének megváltoztatását, vagy a törzsben elhelyezett utasítások végrehajtását jelenti. Egy eljárás akkor nevezhető szabályosnak, ha a program futtatása közben elérjük a végét és külön utasítással befejeztetjük. Míg a függvények általában valamilyen értéket adnak vissza. Egy függvényt meghívni csak kifejezésben lehet, ennek alakja: függvénynév(aktuális\_paraméter\_lista). A blokk egy olyan programegység, amely csak egy másik programegységben helyezkedhet el, tehát külső szinten nem. Ezt a programegységet csak az eljárásorientált nyelvek egy része ismeri. Szerepe a nevek hatáskörének elhatárolása. A blokknak nincs paramétere, de lehetnek a törzsében végrehajthó vagy deklarációs utasítások és úgy lehet elkezdni, hogy szekvenciálisan kerül rá a vezetés vagy egy GOTO-utasítással ráugrunk a kezdetére.

## 6. Paraméterek

Paraméterkiértékelés során először mindig a formális paraméter lista lesz az elsődleges, majd ezután rendeljük hozzá az aktuális paramétereket. Annak módszerét, hogy melyik formális paraméterhez, melyik

aktuális paraméter fog tartozni, megkülönböztethetjük sorrendi kötés vagy név szerinti kötés szerint. Sorrendi kötésnél, mint ahogy a nevében is benn van, felsorolás sorrendjében rendelődnek hozzá. A név szerinti kötésnél pedig, az aktuális paraméter listában megadjuk a formális paraméter lista nevét és mellette pedig valamilyen szintaktikával az aktuális paramétert. A paraméter átadás is különböző módon történhet (érték-, cím-, eredmény-, érték-eredmény-, név-, szöveg szerint).

## 7. Blokk és hatáskör

A blokkok olyan programegységek, amelyek csak egy másik programegység belsejében helyezkedhetnek el. Formálisan 3 részből áll: kezdet, törzs és vég. A kezdetet és a véget egy-egy speciális karaktersorozat vagy alapszó jelzi és a törzsben helyezkednek el a deklarációs és végrehajtható utasítások. Fontos azonban, hogy a blokknak nem lehetnek paraméterei és aktivizálni úgy lehet szekvenciális vezérléssel vagy GOTO utasítással lehet. A blokkok szerepe a nevek hatáskörének eltárolásában van.

A hatáskör a nevekhez kapcsolódó fogalom. Ezzel hivatkozunk az adott programra, annak nevének használatával. Az eljárásorientált nyelvekben a programegységekhez, fordítási egységekhez kapcsolódik. Egy név hatáskörének megállapítását hatáskörkezelésnek hívjuk, aminek két típusa van: Statikus és dinamikus. A statikus hatáskörkezelés esetén a fordító egy szabad nevet keres, ha talál, akkor kilép a tartalmazó programegységbe és megnézi hogy ez a név lokális-e ott. Ha nem talál szabad nevet, akkor vagy a programozónak kell deklarálni egyet vagy automatikusan történik a deklaráció az egyes programnyelvekben. A dinamikus hatáskörkezelést pedig a futtató rendszer végzi, ami ha talál egy szabad nevet a programegységben, akkor megkeresi a lokális nevet is, ha azt nem találja meg, akkor futási hiba keletkezik.

## 10.2. Programozás bevezetés

### [KERNIGHANRITCHIE]

#### Vezérlési szerkezetek

A C nyelvben, ha a kifejezéseket pontosvessző követi őket, akkor utasítássá válnak. A pontosvessző ebben az esetben csak utasításlezárást jelent. A kapcsos zárójelekkel pedig az utasításokat egyetlen összetett blokkba foghatunk össze, de a blokkot lezáró kapcsos zárójelnél nincs pontosvessző.

Az if-else utasítással azt fejezzük ki, hogy a program hogyan döntsön egy adott folyamat során. Tehát először jön az if (kifejezés), ami ha teljesül akkor a kapcsos zárójelben lévő utasítást hajtja végre. Ellenkező esetben az else ág hajtódik végre, ez utóbbi azonban elhagyható, ha nem szeretnénk hogy a program csináljon valamit. Egy példa az if-else utasításra:

```
int a = 1;

if (a == 1) {
 a = 3;
} else {
 a = 4;
}
```

Az else-if utasítás annyiban különbözik az előzőhöz képest, hogy itt nem csak 2, hanem több döntési lehetőség adunk meg. A többszörös elágazás először if-el kezdődik, majd annyi else if ágot adunk, amennyit szeretnénk, végül else elágazással fejezzük be. Ilyenkor a program végig megvizsgálja, hogy hol teljesül a feltétel és igaz esetén végrehajtja azt. Formálisan így néz ki:

```
if (kifejezés)
 utasítás
else if (kifejezés)
 utasítás
else if (kifejezés)
 utasítás
else
 utasítás
```

A switch utasítás is többirányú, itt megvizsgálja, hogy egy adott kifejezés értéke megegyezik-e valamelyik case állandó értékével, ha van ilyen, akkor végrehajtja az utasítást, ellenkező esetben a default-ra ugrik, vagy ha nincs default, akkor nem történik semmi. Formális felépítése:

```
switch (kifejezés) {
 case 'állandó kifejezés': utasítás
 break;
 case 'állandó kifejezés':
 case 'állandó kifejezés': utasítás
 break;
 case 'állandó kifejezés':
 case 'állandó kifejezés':
 default: utasítás
 break;
}
```

A case állandó kifejezés értékeinek különböznie kell egymástól, továbbá minden utasítás végén egy break áll, aminek hatására azonnal kilép a switch utasításból.

A while ciklus során a gép először kiértékeli a kifejezést, aminek ha értéke nem nulla, akkor végrehajtja az utasítást. Ez a folyamat addig ismétlődik, amíg a kifejezés nulla nem lesz.

```
while (kifejezés)
 utasítás
```

For ciklus során is ugyanez történik, de itt 3 kifejezés szerepel. Általában a kif1 és a kif3 értékadás vagy függvényhívás, a kif2 pedig relációs kifejezés. Ha elhagyjuk valamelyik kifejezést, akkor a for ciklussal végtelen ciklust érünk el.

```
for (kif1; kif2; kif3)
 utasítás
```

A do-while utasítás során előbb mindig végrehajtja az utasítást és csak utána értékeli ki a kifejezést. Ez is addig fut le, amíg hamis nem lesz.

A goto utasítással megoldhatóak az olyan problémák, amikor mélyen egymásba ágyazott ciklusokból szeretnénk kilépni. Ilyenkor az egyik ciklusba betesszük a goto utasítást, ami egy másik helyre fog majd mutatni, ott folytatna a folyamatot. Ezt az utasítást nagyon ritkán használjuk.

```
for (. . .)
 for (. . .) {
 . . .
```

```
 if (zavar)
 goto hiba;
 . . .
 }
 hiba:
 számold fel a zavart
```

## 9. kivételkezelés

A kivételkezelési eszközrendszerrel végezhetjük el a megszakítások kezelését. A kivételek okozzák a megszakításokat és a kivételkezelést a program végzi, ha a kivétel bekövetkezik. A Java-ban a kivételkezelés Adaszerű elveket vall. Először is a program működése közben módszerek hívódnak meg, amely során ha bekövetkezik egy esemény, akkor egy kivétel-objektum jön létre. Ez a kivétel-objektum két részből áll: kivétel-osztályok és azok kivétel-példányai. Ha ez az objektum létrejön, akkor a módszer eldobja a kivételt, ami a Java virtuális gép hatáskörébe kerül át, majd az utasítás befejezése után bekövetkezik a kivételkezelés. A Java virtuális gép (JVM) feladata lesz a megfelelő kivételkezelő megkeresése, aminek feltétele, hogy a kivételkezelő típusa megegyezzen a kivétel típusával, továbbá a kivételkezelő őse a kivétel típusának. A Javában a kivételkezelő tulajdonképpen egy blokk, ami tetszőleges kódrészlethez köthető. A kivételeknek pedig két nagy csoportja van:

- **Ellenőrzött kivétel:** egy metódusban felléphet és a programozónak mindig specifikálnia kell, vagy ell kell kapnia őket. Mindig ezt javasolt használni. Használata mindig a módszer fejének végén történik a `THROWS kivételnév_lista` utasítás segítségével. Itt lesznek felsorolva az ellenőrzött, de nem kezelt kivételek. Az objektumok eldobása a `THROW` utasítás segítségével történik, de csak a `java.lang` csomagban definiált `Throwable` obektumaira vonatkoznak ezek.
- **Nem ellenőrzött kivétel:** akkor van szó erről, amikor egy ellenőrzés vagy kellemetlen vagy túl nagy kódrészletet eredményezne, esetleg a programozó nem tud mit kezdeni a kivételekkel. Ilyenkor az ellenőrzés elengedésre kerül.

## 10.3. Programozás

[BMECPP]

### A C és C++ nyelv közötti különbségek

Ha egy függvénynek nem adunk meg paramétert, akkor tetszőleges számú paraméterrel hívhatjuk meg. A C++ nyelvben meg kell adnunk a `void`-ot, aminek jelentése pontosan az, hogy nincs a függvénynek paramétere:

```
// C üres paraméter
void f() {
 //az F függvény törzse
}

//C++ üres paraméter
void f(...) {
 //Az f függvény törzse
}
```



A függvénynevek azonosítása is eltérően működik a két nyelvben. Amíg a C-nél maga a neve az azonosítója, így nem lehet két azonos nevű függvény, addig a C++ nyelvben a függvényeket a nevük és argumentumlistájuk együttesen azonosítja, így fordulhatnak elő azonos nevű függvények. A fordító ezt úgy oldja meg, hogy a függvényneveket kiegészíti a az argumentumtípus elő- vagy utótagjával, így a linker szintjén különböző nevekké jelennek meg. Ezt nevezzük névelferdítésnek. Ahhoz, hogy a C és a C++ közötti együttműködés, emiatt ne legyen problémás, használnunk kell az `extern "C"` deklarációt. Ez lehetővé teszi, hogy a már C fordítóval lefordított függvényeket felhasználhassuk a C++ programunkban is.

```
//Példa C++ nyelvben azonos azonosítónevű, de eltérő ↔
argumentumú függvények
void PrintTime(int hour, int minute) {
 ...
}

int PrintTime(int hour, int minute) {
 ...
}
```

C++ nyelvben van lehetőség alapértelmezett értékeket megadni a függvények argumentumában. Szerepe az, hogy függvényhívásnál megfelelő elemszámú/típusú paramétereket adjunk át. Fontos különbség még, hogy a C++ nyelvben van referenciatípus szerinti paraméterátadás is. Ez feleslegessé teszi a pointereket a cím szerinti paraméterátadásnál:

```
void f(int& i) {
 i = i + 2;
}

int main(void) {
 int a = 0;
 f(a);
 print("%\n", a);
}
```

A fenti programban látható, hogy a referencianeve elé egy `&` jellel deklaráltuk a referenciát.

## Objektumok és osztályok

Az objektumorientált szemlélet akkor alakult ki, amikor a programok egyre bonyolultabbak, átláthatatlannabbak lettek és ennek megoldására találták ki az objektum orientált programozást. Ennek lényege, hogy a rendszer funkcióit egymással együttműködő objektumok valósítják meg. Tehát mindig van egy osztályunk és az osztályon belül különböző objektumok lehetnek. Ezek összessége jelenti az egységbe zárást. Például tagfüggvények esetében, megadhatjuk őket osztálydefinícióban vagy pedig struktúra-definícióban. Ez utóbbira egy példa:

```
struct Point {
 unsigned int x;

 int setX(unsigned int value);
};

int Point::setX(unsigned int value) {
 if(value <= MAX_X) {
```

```
 x = value;
 return 0;
 }
 return -1;
}
```

Itt tehát egy hatókör operátort használunk, azaz dupla kettőspontot. Az osztály nevét és a hatóköroperátort a függvény neve elé írva tudatjuk a fordítóval, hogy a Point osztályon belül a setX függvényről van szó.

### C++ sablonok

Sablonnak nevezzük azokat az osztálysablonokat és függvénysablonokat, amikor bizonyos elemeket a függvény/osztály definiálásakor nem adjuk meg, hanem paraméterként kezelünk. A függvénysablont úgy kell elképzelni, hogy például ha egy számsorozat legnagyobb értékét akarjuk kiírni, akkor elég előhívni a max() függvényt, aminek csak a paramétereket kell átadni. A C++ esetében a függvénysablon implementációja a következőképpen történik:

```
template <class T> inline T max(T lhs, T rhs) {
 return lhs > rhs ? lhs : rhs;
}
```

Tehát először mindig a template kulcsszóval kezdődik, majd ezután < > között felsoroljuk a sablonparamétereket vesszővel elválasztva. Jelen esetünkben egy sablonparaméter szerepel, aminek a T nevet adtuk. Az inline-t itt azért használjuk, mert a függvénytörzsünk rövid. Az elkészült függvénysablont ezután így tudjuk felhasználni: `int n = max(3, 5)`. Fontos, hogy nem lehet az egyik paraméter double, a másik pedig int típusú, mert a kód ebben az esetben nem fordul le.

### Típuskonverziók

A C++ típuskonverziói hasonlítanak a C nyelvre, de azrét vannak különbségek: sokkal biztonságosabbak, új objektumorientált elemek tartoznak a típuskonverziókhoz és a referenciatípusra is vonatkoznak a konverziós szabályok.

**A beépített típusok közötti típuskonverziók** esetén a C nyelvben létezik enum és int közötti konverzió, míg C++ esetén ki kell írunk a típuskonverziót. A pointerekhez hasonlóan a referenciák is egy memóriaterületre hivatkoznak, így ha nem szeretnénk megváltoztatni az átadott értéket, de lemásolni sem, ezért a referenciát konstansnak deklaráljuk, ezzel elérve a referencia szerinti átadását másolás nélkül.

**felhasználói típusok konverziója** kétféle lehet: független típusok közötti és öröklési hierarchia mentén. **A független típusok közötti** konverzió két lehetőséget kínál. Egyik eset az, amikor egy másik típusról a mi osztályunk típusára konvertálunk, ezt nevezzük konverziós konstruktornak. A másik pedig a konverziós operátor, amikor az osztályunkról konvertálunk egy másik típusra.

**Az öröklési hierarchia mentén** történő konverzió során a leszármazottról a szülőre történik a folyamat. Ekkor mindig a másoló konstruktor fog meghívódni. Mivel másolókonstruktor mindig van, ezért ez a típusú konverzió automatikusan végrehajtódik. A szülőről a leszármazotttra nincs ilyen jellegű konverzió, de ha mégis szükségünk lenne rá, akkor a leszármazottban konverziós konstruktort kell írunk.

**A C++ típuskonverziós operátorai** lehetnek: `static_cast` (statikus típuskonverzió), `const_cast` (konstans típuskonverzió), `dynamic_cast` (dinamikus típuskonverzió), `reinterpret_cast` (újraértelmező típuskonverzió). Ezek közül a leggyakrabban használt konverzió a statikus. A konstans típuskonverzióra jellemző, hogy

egyedül ez képes konstans típust nem konstanssá tenni, illetve volatíe típust nem azzá. A dinamikus típuskonverzió az öröklési hierarchián lefelé történő konverziókhoz használjuk. Ez futási időben ellenőrzi, hogy végrehajtható-e a konverzió, ezért egy virtuális függvény kell tartalmaznia. Az újraértelmező típuskonverzió pedig az implementáció konverziók esetén alkalmazható. Általában a pointer típusát változtatjuk meg ebben az esetben.

### Kivételkezelés

A C++ nyelvben a kivételkezelés nemcsak hiba, hanem bármilyen kivételes eset esetén használható. Lényege, hogy ha hibát detektálunk valahol, akkor a program futása a kivételkezelő ágon folytatódik tovább. Az alábbi kódrészletben ha a program futtatása során, a felhasználó 0-at ad meg, akkor hibát dobunk. Ehhez egy try-catch blokkot használunk. A try blokkba írjuk a normál kódot és a catch ágba a hibakezelő kódot. Ha helyes számot adtunk meg, akkor a try blokkban valamennyi utasítás lefut, kiíratjuk a szöveget, majd a catch blokkot átlépve folytatódik tovább a folyamat. Ha a védett blokkban hibát találunk, akkor a Throw kulcsszóval kivételt dobunk, ami jelen esetben egy const char\* típusú szöveg. Ezután egyből a catch ágba kerül, itt fontos megjegyezni, hogy több catch ág is lehet. Annak eldöntése, hogy melyik lesz aktiválva, a catch-nak átadott paraméter típusától függ. Ha a paraméter kompatibilis a kapott kivétellel, akkor azt mondjuk, hogy elkapja a dobott kivételt. Ezután lefut a catch törzse, majd innen folytatódik tovább a folyamat. Ha egy kivételt nem kapunk el, akkor az egy kezeletlen kivétel. Ebben az esetben az alapértelmezett könyvtári terminate függvény hívódik meg.

```
#include <iostream>
using namespace std;

int main() {
 try {
 double d;
 cout << "Adj meg egy nem nulla számot: ";
 cin >> d;
 if (d == 0)
 throw "A beírt szám nem lehet nulla.";
 cout << "A szám reciproka: " << 1/d << endl;
 }
 catch (const char* exc) {
 cout << "Hiba! A hibaüzenet: " << exc << endl;
 }
 cout << "Done." << endl;
}
```

A hívási verem visszacsévézésének nevezzük azt a folyamatot, amikor egy kivétel dobásakor annak elkapásáig a függvények hívási láncában felfelé haladva az egyes függvények lokális változói felszabadulnak. Példa:

```
int main() {
 try {
 f1();
 }
 catch (const char* errorText) {
 cerr << errorText << endl;
 }
}
```

```
 }

 void f1() {
 Fifo fifo;
 f2();
 ...
 }

 void f2() {
 int i = 1;
 throw "error1";
 }
```

Először tehát az f2() függvény kivételt dob, ezért az f2()-ben definiált i lokális változó felszabadul. Az f1()-ben lefoglalt Fifo fifo objektum is felszabadul és meghívódik a destruktora, majd végül lefut a main függvényben lévő catch blokk.

## **III. rész**

### **Második felvonás**

DRAFT

**Bátf41 Haxor Stream**

A feladatokkal kapcsolatos élő adásokat sugároz a <https://www.twitch.tv/nbatfai> csatorna, melynek permanens archívuma a <https://www.youtube.com/c/nbatfai> csatornán található.

---

DRAFT

---

## 11. fejezet

# Helló, Arroway!

### 11.1. OO szemlélet

A módosított polártranszformációs normális generátor beprogramozása Java nyelven. Mutassunk rá, hogy a mi természetes saját megoldásunk (az algoritmus egyszerre két normálist állít elő, kell egy példánytag, amely a nem visszaadottat tárolja és egy logikai tag, hogy van-e tárolt vagy futtatni kell az algoritmust) és az OpenJDK, Oracle JDK-ban a Sun által adott OO szervezés ua.!

Megoldás forrása: [PolarGenerator.java](#)

Ebben a feladatban a polártranszformációs generátort nézzük meg Java verzióban, melynek célja, hogy egyszerre két random számot állítson elő, amik közül az egyiket eltárolja, hogy a következő hívásnál visszaadja ennek értékét. Tehát lényegében minden páratlanodik meghíváskor az előző eltárolt adatot kapjuk vissza, majd a következő lépésnél újra generálunk két véletlenszámot. Annak megállapítására, hogy van-e az előző lépésből eltárolt érték, az `boolean` típusú változót használunk, azaz ha van tárolt értékünk, akkor a `nincsTarolt` hamis lesz, ellenkező esetben igaz.

```
public class PolarGenerator {
 boolean nincsTarolt = true;
 double tarolt;

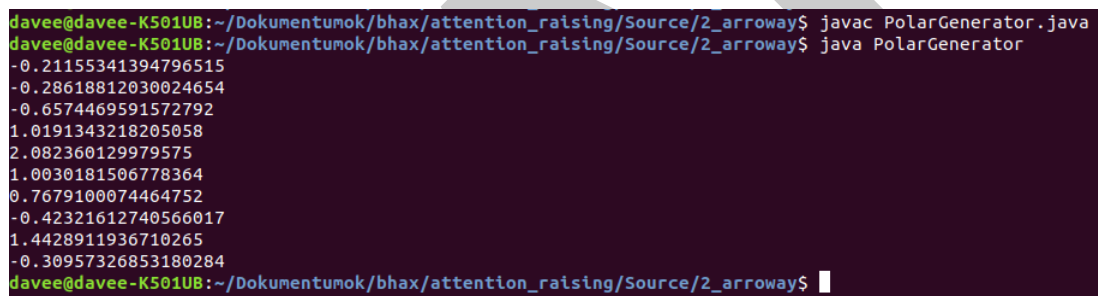
 public PolarGenerator() {
 nincsTarolt = true;
 }

 public double kovetkezo() {
 if (nincsTarolt) {
 double u1, u2, v1, v2, w;
 do {
 u1 = Math.random();
 u2 = Math.random();
 v1 = 2 * u1 - 1;
 v2 = 2 * u2 - 1;
 w = v1 * v1 + v2 * v2;
 } while (w > 1);
 double r = Math.sqrt((-2 * Math.log(w)) / w);
```

```
 tarolt = r * v2;
 nincsTarolt = !nincsTarolt;
 return r * v1;
 } else {
 nincsTarolt = !nincsTarolt;
 return tarolt;
 }
}

public static void main(String[] args) {
 PolarGenerator g = new PolarGenerator();
 for (int i = 0; i < 10; ++i) {
 System.out.println(g.kovetkezo());
 }
}
}
```

A program futtatása és annak eredménye az alábbi képen látható:



```
davee@davee-K501UB:~/Dokumentumok/bhax/attention_raising/Source/2_arroway$ javac PolarGenerator.java
davee@davee-K501UB:~/Dokumentumok/bhax/attention_raising/Source/2_arroway$ java PolarGenerator
-0.21155341394796515
-0.28618812030024654
-0.6574469591572792
1.0191343218205058
2.082360129979575
1.0030181506778364
0.7679100074464752
-0.42321612740566017
1.4428911936710265
-0.30957326853180284
davee@davee-K501UB:~/Dokumentumok/bhax/attention_raising/Source/2_arroway$
```

11.1. ábra. A polárgenerátor eredménye

Ha későbbiekben szükségünk lenne egy hasonló programra, akkor használhatjuk az OpenJDK Random.java fájlban megtalálható NetxGaussian() metódust. Ennek felépítése hasonló a mi programunkhoz:



```
private double nextNextGaussian;
private boolean haveNextNextGaussian = false;

synchronized public double nextGaussian() {
 // See Knuth, ACP, Section 3.4.1 Algorithm C.
 if (haveNextNextGaussian) {
 haveNextNextGaussian = false;
 return nextNextGaussian;
 } else {
 double v1, v2, s;
 do {
 v1 = 2 * nextDouble() - 1; // between -1 and 1
 v2 = 2 * nextDouble() - 1; // between -1 and 1
 s = v1 * v1 + v2 * v2;
 } while (s >= 1 || s == 0);
 double multiplier = StrictMath.sqrt(-2 * StrictMath.log(s)/s);
 nextNextGaussian = v2 * multiplier;
 haveNextNextGaussian = true;
 return v1 * multiplier;
 }
}
```

11.2. ábra. Java.random

Látható, hogy szerepel egy `synchronized` kulcsszó is a metódus előtt. Ezzel egy szál lock-olhat vagy unlock-olhat egy objektumot. Egyszerre csak egy szál hajtódik végre és a többi szál blokkolt állapotba kerül és addig vár, amíg sorra nem kerül. A másik különbség, ami még megfigyelhető a mi programunkhoz képest, hogy JDK-ban `StrictMath` osztályt használják `Math` helyett. A különbség a két osztály között, hogy a `Math` esetében nem biztosított, hogy minden Java implementációban ugyanazt az eredményt kapjuk. Ha nem fontos, hogy minden lehetséges implementációban bitről bitre megegyezzenek a számítások, akkor maradjunk a `Math` osztálynál, mert ennek működése gyorsabb.

## 11.2. Homokozó

Írjuk át az első védelési programot (LZW binfa) C++ nyelvről Java nyelvre, ugyanúgy működjön! Mutasunk rá, hogy gyakorlatilag a pointereket és referenciákat kell kiirtani és minden máris működik (erre utal a feladat neve, hogy Java-ban minden referencia, nincs választás, hogy mondjuk egy attribútum pointer, referencia vagy tagként tartalmazott legyen). Miután már áttettük Java nyelvre, tegyük be egy Java Servletbe és a böngészőből GET-es kéréssel (például a böngésző címsorából) kapja meg azt a mintát, amelynek kiszámolja az LZW binfáját!

Első Megoldás forrása: [LZWBinFa.java](#)

Második Megoldás forrása: [LZWJavaServlet.java](#)

A programunk először az osztályok importálásával kezdődik. A `PrintWriter` segítségével írjuk ki a szövegeket, a `BufferedReader` az osztályok bufferelt olvasáshoz használható. A `FileReader` osztályra értelemszerűen a fájlok beolvasásához lesz majd szükségünk. Kelleni fog még a hibakereső osztályok, azaz az `IOException` és a `FileNotFoundException`.

```
import java.io.PrintWriter;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.FileNotFoundException;
```

A program további részével kapcsolatban nem célom minden egyes kódcsipet működését elmagyarázni, mert megegyezik a C++ nyelvben megírt binfa feladatban leírtakkal. Helyette csak a lényeges különbségeket emelem ki. A leginkább szembevető változás a Java-ban, hogy itt minden objektum egy referencia, kivéve a primitív típusok. Itt nincsenek mutatók, így azok törlésre kerülnek a programunkból, továbbá az operátor túlterhelést is el kell hagynunk.

```
class LZWBinFa {

 public LZWBinFa() {
 gyoker = new Csomopont('/');
 fa = gyoker;
 }

 public void operator(char b) {
 if (b == '0') {
 if (fa.nullasGyermek() == null) {
 Csomopont uj = new Csomopont('0');
 fa.ujNullasGyermek(uj);
 fa = gyoker;
 }
 else {
 fa = fa.nullasGyermek ();
 }
 } else {
 if (fa.egyenesGyermek() == null) {
 Csomopont uj = new Csomopont('1');
 fa.ujEgyenesGyermek (uj);
 fa = gyoker;
 }
 else {
 fa = fa.egyenesGyermek();
 }
 }
 }

 public void kiir() {
 melyseg = 0;
 kiir (gyoker, new java.io.PrintWriter(System.out));
 }

 public void kiir (java.io.PrintWriter os) {
```

```
melyseg = 0;
kiir (gyoker, os);
}

class Csomopont {
 public Csomopont (char betu) {
 this.betu = betu;
 balNulla= null;
 jobbEgy = null;
 };

 public Csomopont nullasGyermek() {
 return balNulla;
 }

 public Csomopont egyesGyermek() {
 return jobbEgy;
 }

 public void ujNullasGyermek (Csomopont gy) {
 balNulla = gy;
 }

 public void ujEgyesGyermek (Csomopont gy) {
 jobbEgy = gy;
 }

 public char getBetu() {
 return betu;
 }

 private char betu;
 private Csomopont balNulla = null;
 private Csomopont jobbEgy = null;
};

private Csomopont fa = null;
private int melyseg, atlagosszeg, atlagdb;
private double szorasosszeg;

public void kiir (Csomopont elem, java.io.PrintWriter os) {
 if (elem != null) {
 ++melyseg;
 kiir (elem.egyesGyermek(), os);
 for (int i = 0; i < melyseg; ++i) {
 os.print ("---");
 }
 os.print(elem.getBetu() + "(" + (melyseg - 1) + ")\n");
 kiir (elem.nullasGyermek(), os);
 --melyseg;
 }
}
```

```
 }
}

protected Csomopont gyoker;
protected int maxMelyseg;
protected double atlag, szoras;

public int getMelyseg() {
 melyseg = maxMelyseg = 0;
 rmelyseg (gyoker);
 return maxMelyseg - 1;
}

public double getAtlag() {
 melyseg = atlagosszeg = atlagdb = 0;
 ratlag (gyoker);
 atlag = ((double) atlagosszeg) / atlagdb;
 return atlag;
}

public double getSzas() {
 atlag = getAtlag ();
 szorasosszeg = 0.0;
 melyseg = atlagdb = 0;

 rszas (gyoker);

 if (atlagdb - 1 > 0) {
 szoras = Math.sqrt (szorasosszeg / (atlagdb - 1));
 } else {
 szoras = Math.sqrt (szorasosszeg);
 }
 return szoras;
}

public void rmelyseg (Csomopont elem) {
 if (elem != null) {
 ++melyseg;
 if (melyseg > maxMelyseg) {
 maxMelyseg = melyseg;
 }
 rmelyseg (elem.egyGyermeke ());
 rmelyseg (elem.nullasGyermeke ());
 --melyseg;
 }
}

public void ratlag (Csomopont elem) {
 if (elem != null) {
```

```
 ++melyseg;
 ratlag (elem.egyenesGyermek ());
 ratlag (elem.nullasGyermek ());
 --melyseg;
 if (elem.egyenesGyermek () == null && elem.nullasGyermek () == null) {
 ++atlagdb;
 atlagosszeg += melyseg;
 }
 }
}

public void rszoras (Csomopont elem) {
 if (elem != null) {
 ++melyseg;
 rszoras (elem.egyenesGyermek ());
 rszoras (elem.nullasGyermek ());
 --melyseg;
 if (elem.egyenesGyermek () == null && elem.nullasGyermek () == null) {
 ++atlagdb;
 szorasosszeg += ((melyseg - atlag) * (melyseg - atlag));
 }
 }
}

public static void usage() {
 System.out.println("Usage: lzwtree in_file -o out_file");
}
```

Eddig a kódrészletig tehát az látható, hogy még a destruktorkat is kivettük a programunkból, ugyanis a C++ nyelvben volt egy szabadít függvényünk, mely a memória felszabadítására szolgált. Java-ban ez automatikusan történik a Garbage Collector-nak köszönhetően. Amikor egy új objektumot hozunk létre, akkor a referencia a deklaráló blokk végén felszámolódik és a new-val létrehozott objektumot a szemétyűjtő felszámolhatja ha már a későbbiekben nem használjuk. Ennek az automatikus szemétyűjtésnek és a mutatók elhagyásának köszönhetően szerintem sokkal letisztultabb lett a programunk a C++ verzióhoz képest.

```
public static void main (String args[]) throws FileNotFoundException,
IOException {
 if (args.length < 3) {
 usage();
 return;
 }

 String inFile = args[0];
```

```
if (!"-o".equals(args[1])) {
 System.out.println(args[1]);
 usage ();
 return;
}

java.io.FileInputStream beFile = new java.io.FileInputStream(new ↵
 java.io.File(inFile));

PrintWriter kiFile = new PrintWriter(args[2]);

byte[] b = new byte[1];

LZWBinFa binFa = new LZWBinFa();

while (beFile.read(b) != -1) {
 if (b[0] == 0x0a) {
 break;
 }
}

boolean kommentben = false;

while (beFile.read(b) != -1) {

 if (b[0] == 0x3e) {
 kommentben = true;
 continue;
 }

 if (b[0] == 0x0a) {
 kommentben = false;
 continue;
 }

 if (kommentben) {
 continue;
 }

 if (b[0] == 0x4e){
 continue;
 }

 for (int i = 0; i < 8; ++i)
 {
 if ((b[0] & 0x80) != 0) {
 binFa.operator('1');
 } else {
 binFa.operator('0');
 }
 }
}
```

```
 b[0] <= 1;
 }

}

binFa.kiir(kiFile);

kiFile.println("depth = " + binFa.getMelyseg() + "\n");
kiFile.println("mean = " + binFa.getAtlag() + "\n");
kiFile.println("var = " + binFa.getSzoras() + "\n");

kiFile.close ();
beFile.close ();
}
};
```

A main függvénynél fontos kiemelni a kivételkezelést, ami a Java-nál alapnak számít a használata. Jelen esetben ezt a main függvény fejlécében adtuk meg (throws `FileNotFoundException`, `IOException`), ami automatikusan a `java.io.FileInputStream` osztályhoz kapcsolódik, ami a fájlkezelést felügyeli. Ha futtatjuk a programot, akkor egy `b.txt` fájlba lesz elmentve az eredmény, ahogy az alábbi képen látható:



### 11.3. ábra. Java Binfá

Most pedig nézzük meg a Java Servlet verziót. Ehhez előbb fel kell telepítanunk a Tomcat szerveret, ami a <http://tomcat.apache.org/> oldalról lehet letölteni. Kicsomagolás után lépünk be a `bin` mappába, majd a terminálban a `sudo ./catalina.sh start;` parancssorral tudjuk elindítani. Ha a böngészőbe beírjuk, hogy `localhost:8080` és bejön az Apache Tomcat oldal, akkor mindent jól csináltunk eddig. A lefordított java class-okat a `/webapps/ROOT/WEB-INF/classes` mappába kell bemásolni. és az `/webapps/ROOT/WEB-INF/web.xml` fájlba is be kell írunk a következőket:

```
<servlet>
 <servlet-name>LZWJavaServlet</servlet-name>
 <servlet-class>LZWJavaServlet</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>LZWJavaServlet</servlet-name>
 <url-pattern>/LZWJavaServlet</url-pattern>
</servlet-mapping>
```



Mivel most a böngészőben szeretnénk egyszerű módon megjeleníteni a binfát, ezért most kitöröljük a fájlkezeléssel kapcsolatos részeket a programunkból, helyette `CreateStringBuffer()` osztályt fogjuk használni, mely egy string értéket vár (bemenet). Ez lényegében majd a böngésző címsorában kap fontos szerepet, amikor egy szöveget adunk meg bemenetként.

```
public void CreateStringBuffer(String bemenet)
{
 for(int i = 0; i < bemenet.length(); ++i){
 int c = bemenet.charAt(i);
 if(c == 0x0a){
 break;
 }
 }
 boolean kommentben = false;

 for(int i = 0; i < bemenet.length(); ++i)
 {
 int c = bemenet.charAt(i);

 if (c == 0x3e) {
 kommentben = true;
 continue;
 }
 if (c == 0x0a) {
 kommentben = false;
 continue;
 }
 if (kommentben) {
 continue;
 }
 if (c == 0x4e)
 {
 continue;
 }

 for (int j = 0; j < 8; ++j) {

 if ((c & 0x80) == 128)
 {
 hozzarendel('1');
 } else
 {
 hozzarendel('0');
 }
 c <<= 1;
 }
 }

 kiir();
}
```

```
kimenet.append("depth = " + getMelyseg() + "
\n");
kimenet.append("mean = " + getAtlag() + "
\n");
kimenet.append("var = " + getSzoras() + "
\n");
}
```

A programunk utolsó metódusa egy `doGet` függvény, amivel GET kérést küldhetünk a szervernek, ami a már fentebb említett bemenethez kapcsolódik. Ezután már csak azt kell megadnunk, hogy nézzon ki a weboldalunk. Aki fejlesztett már weboldalt, annak ismerős lehet a HTML, mivel lényegében itt is ezt használjuk, csak itt a HTML kódotkat a Java `out.println` függvényével íratjuk ki.

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
 throws ServletException, IOException {

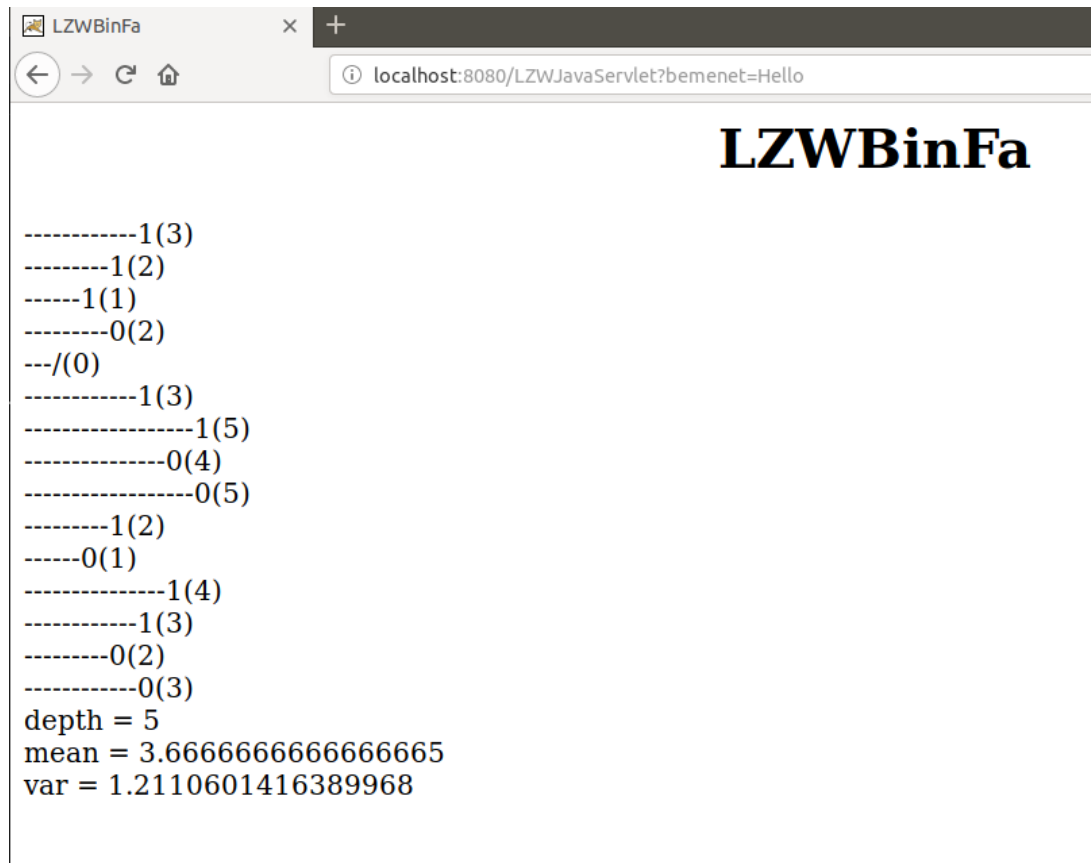
 // Set response content type
 response.setContentType("text/html");

 if(request.getParameter("bemenet") == null){
 return;
 }
 else
 CreateStringBuffer(request.getParameter("bemenet"));

 PrintWriter out = response.getWriter();
 String title = "LZWBinFa";
 String docType =
 "<!doctype html public \"-//w3c//dtd html 4.0 \" + \"transitional// ↵
 en\">\n";

 out.println(docType +
 "<html>\n" +
 "<head><title>" + title + "</title></head>\n" +
 "<body>\n" +
 "<h1 align = \"center\">" + title + "</h1>\n" +
 kimenet +
 "</body>" +
 "</html>"
);
}
```

Ha ezzel megvagyunk, beírhatjuk a böngészőbe, hogy: `http://localhost:8080/LZWJavaServlet?be`, ahol a szöveg szó tetszőleges lehet. A kapott eredmény pedig így néz ki:



11.4. ábra. LZWJavaServlet.class megjelenítése

### 11.3. „Gagyi”

Az ismert formális 2 „while ( $x \leq t \ \&\& \ x \geq t \ \&\& \ t \neq x$ );” tesztkérdéstípusra adj a szokásosnál (miszerint  $x$ ,  $t$  az egyik esetben az objektum által hordozott érték, a másikban meg az objektum referenciája) „mélyebb” választ, írd Java példaprogramot mely egyszer végtelen ciklus, más  $x$ ,  $t$  értékekkel meg nem! A példát építsd a JDK Integer.java forrására 3, hogy a 128-nál inkluzív objektum példányokat poolozza!

Első Megoldás forrása: [Gagyi1.java](#)

Második Megoldás forrása: [Gagyi2.java](#)

A feladat lényege, hogy megmutassuk, hogy két-két azonos szám megadásával miért viselkedik eltérő módon ugyanaz a while ciklus. Az alábbi programban  $x = -128$  és  $t = -128$  lesz:

```
public class Gagyi2 {
 public static void main (String[] args) {
 Integer x = -128;
 Integer t = -128;

 System.out.println(x);
 System.out.println(t);
 }
}
```

```
 while (x <= t && x >= t && t !=x);
 }
}
```

```
davee@davee-K501UB:~/Dokumentumok/bhax/attention_raising/Source/2_arroway/Gagy1$ javac Gagyi2.java
davee@davee-K501UB:~/Dokumentumok/bhax/attention_raising/Source/2_arroway/Gagy1$ java Gagyi2
-128
-128
davee@davee-K501UB:~/Dokumentumok/bhax/attention_raising/Source/2_arroway/Gagy1$
```

11.5. ábra. -128-as érték

A program elindítása után csak egyszer fut le ciklus, hiszen a while ciklusban megadott utolsó feltétel nem teljesül:  $-128 \neq -128$ . De most nézzük meg, hogy mi történik akkor, ha  $x$  és  $t$  értéke  $-129$  lesz:

```
public class Gagyi2 {
 public static void main (String[]args) {
 Integer x = -129;
 Integer t = -129;

 System.out.println(x);
 System.out.println(t);

 while (x <= t && x >= t && t !=x);
 }
}
```

Ilyenkor arra számítanánk, hogy az előző példához hasonlóan csak egyszer fut le, de itt most végtelen ciklust kapunk:

```
davee@davee-K501UB:~/Dokumentumok/bhax/attention_raising/Source/2_arroway/Gagy1$ javac Gagyi1.java
davee@davee-K501UB:~/Dokumentumok/bhax/attention_raising/Source/2_arroway/Gagy1$ java Gagyi1
-129
-129
-
```

11.6. ábra. -129-as érték

Ez úgy lehetséges, hogy a Java JDK Integer osztályában  $-128$ -tól  $127$ -ig van alaphól megadva az érték, tehát amíg ezen belül adunk meg értéket a programunknak, akkor ugyanarra a memóriacímre hivatkoznak. Ellenkező esetben, két különböző címen tárolja el az objektumunkat. Ez utóbbi az oka annak, hogy a második programban végtelen ciklust kapunk, mert  $-129 \neq -129$  feltétel ebben az esetben már igaz lesz az eltérő memóriacímük miatt.

## 11.4. Yoda

Írjunk olyan Java programot, ami `java.lang.NullPointerException`-el leáll, ha nem követjük a Yoda conditions-t!  
[https://en.wikipedia.org/wiki/Yoda\\_conditions](https://en.wikipedia.org/wiki/Yoda_conditions)

Megoldás forrása:

A Yoda conditions egy programozási stílus, ahol egy feltétel két részét felcseréljük. Tehát a konstans kerül a feltétel bal oldalára, míg a változó a jobb oldalra. E stílus nevét a Star Wars filmből ismert Yoda-tól kapta, aki angolul beszél ugyan, de nem a nyelv hagyományos szintaxisával. Ez azért hasznos, mert a Yoda stílusban megírt programunkban a futtatás során hibát kapunk, ha valamit nem jól csináltunk, míg a hagyományos verzióban lefut ugyan, de nem fogunk kapni visszajelzést a hibáról. Az alábbi szemlélteti a hagyományos és a Yoda conditions használatát:

Megoldás forrása: [Yoda.java](#)

```
//Hagyományos, ahogy eddig programoztunk
if(midiChlorian == 2000) { /* ... */ }

//Yoda conditions
if(2000 == midiChlorian) { /* ... */ }
```

Ez alapján már könnyen megoldhatjuk a feladatot, aminek megoldása:

```
class Yoda {

 public static void main (String[] args) {

 String force = null;

 if(force.equals("Light side")) {
 System.out.println("Yoda");
 }

 }

}
```

Mivel nem követtük a Yoda conditions-t, ezért egy `java.lang.NullPointerException` hibaüzenet fog megjelenni a terminálban. De ha kicseréljük az `if` sort erre: `if ("Light side".equals(force))`, akkor már nem kapunk error-t.

## 11.5. Kódolás from scratch

Induljunk ki ebből a tudományos közleményből: <http://crd-legacy.lbl.gov/~dhbailey/dhbpapers/bbp-alg.pdf> és csak ezt tanulmányozva írjuk meg Java nyelven a BBP algoritmus megvalósítását! Ha megakadsz, de csak végső esetben: [https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apbs02.html#pi\\_jegyei](https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apbs02.html#pi_jegyei) (mert ha csak lemásolod, akkor pont az a fejlesztői élmény marad ki, melyet szeretném, ha átélnél).

Megoldás forrása: [BBP.java](#)

A BBP algoritmust Bailey, Borwei és Plouffe 1995-ben publikálta, mellyel kiszámítható a Pí tetszőleges számjegye hexadecimális számrendszerben az előző számjegyek ismerete nélkül. Nézzük meg a java programunkat. Először kiszámítjuk ezt:  $\{16^d \text{ Pi}\} = \{4 * \{16^d \text{ S1}\} - 2 * \{16^d \text{ S4}\} - \{16^d \text{ S5}\} - \{16^d \text{ S6}\}\}$ . Itt a {} jel a törtet jelöli.

```
public class PiBBP {

 public PiBBP(int d) {

 double d16Pi = 0.0d;

 double d16S1t = d16Sj(d, 1);
 double d16S4t = d16Sj(d, 4);
 double d16S5t = d16Sj(d, 5);
 double d16S6t = d16Sj(d, 6);

 d16Pi = 4.0d*d16S1t - 2.0d*d16S4t - d16S5t - d16S6t;

 d16Pi = d16Pi - StrictMath.floor(d16Pi);

 StringBuffer sb = new StringBuffer();

 Character hexaJegyek[] = {'A', 'B', 'C', 'D', 'E', 'F'};

 while(d16Pi != 0.0d) {

 int jegy = (int)StrictMath.floor(16.0d*d16Pi);

 if(jegy<10)
 sb.append(jegy);
 else
 sb.append(hexaJegyek[jegy-10]);

 d16Pi = (16.0d*d16Pi) - StrictMath.floor(16.0d*d16Pi);
 }

 d16PiHexaJegyek = sb.toString();
 }
}
```

Itt a  $\{16^d \text{ Sj}\}$  részletet számítjuk ki, ahol a d paraméter segítségével a +1 hexa jegytől számítjuk és a j pedig az Sj indexe lesz.

```
1
 public double d16Sj(int d, int j) {

 double d16Sj = 0.0d;

 for(int k=0; k<=d; ++k)
 d16Sj += (double)n16modk(d-k, 8*k + j) / (double)(8*k + j);
 }
}
```

```
 return d16Sj - StrictMath.floor(d16Sj);
 }

 public long n16modk(int n, int k) {

 int t = 1;
 while(t <= n)
 t *= 2;

 long r = 1;

 while(true) {

 if(n >= t) {
 r = (16*r) % k;
 n = n - t;
 }

 t = t/2;

 if(t < 1)
 break;

 r = (r*r) % k;

 }

 return r;
 }

 public String toString() {

 return d16PiHexaJegyek;
 }

 public static void main(String args[]) {
 System.out.print(new PiBBP(1000000));
 }
}
```

## 12. fejezet

# Helló, Liskov!

### 12.1. Liskov helyettesítés sértése

Írjunk olyan OO, leforduló Java és C++ kódcsipetet, amely megsérti a Liskov elvet! Mutassunk rá a megoldásra: jobb OO tervezés.

Felhasznált irodalom: [PROTECHNIKA]

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/tree/master/attention\\_raising/Source/Liskov\\_2/liskov](https://gitlab.com/davidhalasz/bhax/tree/master/attention_raising/Source/Liskov_2/liskov)

A Liskov féle behellyettesítési elv (LSP) egy olyan programozási technológia, amely során a program viselkedése nem változik meg attól, hogy az ő osztály egy példánya helyett a jövőben valamelyik gyermek osztályának példányát használom. Eredeti angol megfogalmazása: „If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T”. Az LSP tehát akkor érvényesül, ha az az altípus egyes műveleteinek

- argumentumai bővebb típusra cserélhetőek
- visszatérési értéke szűkebb típusú lehet
- kivételkezelése nem dobhat más típust, mint az őstípus megfelelő műveletei, kivéve, ha a kivétel típusát szűkítettük

Egy programban érvényesül még az LSP, ha teljesül még:

- az altípusban nem erősebbek az előfeltételek, mint az őstípusban,
- az altípusban az utófeltételek nem gyengébbek, mint az őstípusban,
- az őstípus invariánsainak az altípusban is invariánsnak kell maradniuk,
- az altípus új műveletei nem módosíthatják, az őstípusból is elérhető állapotot, csak az újonnan bevezetett részeket.

Példaként nézzünk meg egy Java kódcsipetet, ami megsérti a Liskov elvet. Bevezetésként egy olyan programról lesz szó, ahol az őosztály a Madár lesz, amihez két gyermek tartozik: a sas és a pingvin. A téma pedig a repülési képesség lesz.



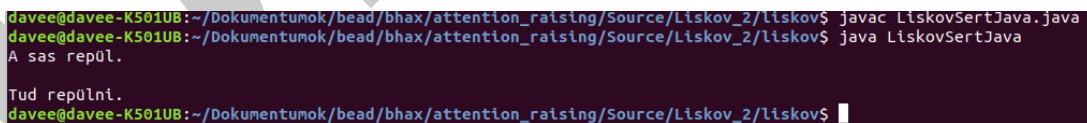
```
// ez a T az LSP-ben
class Madar {
 public void repul() {
 System.out.println("Tud repülni.\n");
 }
}

// itt jönnek az LSP-s S osztályok
class Sas extends Madar {
 public void repul() {
 System.out.println("A sas repül.\n");
 }
}

class Pingvin extends Madar {
}

class LiskovSertJava {
 public static void repulok(Madar r) {
 r.repul();
 }
 public static void main(String[] args) {
 Madar sas = new Sas();
 Madar pingvin = new Pingvin();

 repulok(sas);
 repulok(pingvin); // sérül az LSP, mert a repulok() függvény ←
 röptetné a Pingvint, ami ugye lehetetlen.
 }
}
```



```
davee@davee-K501UB:~/Dokumentumok/bead/bhax/attention_raising/Source/Liskov_2/liskov$ javac LiskovSertJava.java
davee@davee-K501UB:~/Dokumentumok/bead/bhax/attention_raising/Source/Liskov_2/liskov$ java LiskovSertJava
A sas repül.
Tud repülni.
davee@davee-K501UB:~/Dokumentumok/bead/bhax/attention_raising/Source/Liskov_2/liskov$
```

12.1. ábra. Az LSP megsértése

A fenti képen láthatjuk, hogy a sas tud repülni, de emellett kiírja, hogy az adott osztályhoz tartozó pingvin is "Tud repülni.". Ez pedig téves, ami az LSP sérülésének eredménye. Ez azért lehetséges, mert minden gyermek a `repulok()` metódust is örökli a `Madar` őszosztályból. Ahhoz, hogy ne sértsük a LSP-t, át kell írunk a programunkat úgy, hogy az őszosztály gyermekei maradjanak madarak, de ne minden madár tudjon repülni, ehhez pedig használnunk kell egy interface deklarációt, aminek a neve jelen esetben a `Madar`,

ami egy absztrakt típusú lesz. Továbbá lesznek még TudRepulni és NemTudRepulni public típusú interfészek is, amik már más csomagokból is elérhetővé válnak. Ezeket az interfészeket a `implements` kulcsszóval implementálhatjuk. Egy osztály tetszőleges számú interfészt implementálhat, ezáltal többszörös öröklődéshez hasonló dolog jöhet létre. A Sas osztályba implementáljuk a TudRepulni interfészt, majd a kapcsos zárójelek közé tesszük be az előző programunkból ismert `repul()` metódust. A Pingvin osztály értelemszerűen, megkapja a NemTudRepulni interfészt. Itt most nem adunk meg semmilyen tulajdonságot.

```
interface Madar {

}

interface TudRepulni extends Madar {
 public void repul();
}

interface NemTudRepulni extends Madar {

}

class Sas implements TudRepulni {
 public void repul() {
 System.out.println("A sas repül.");
 }
}

class Pingvin implements NemTudRepulni {
 public void repul() {
 System.out.println("A pingvin nem repül.");
 }
}

class LiskovJava {
 public static void repulok(TudRepulni r) {
 r.repul();
 }

 public static void main(String[] args) {
 Sas sasR = new Sas();
 Pingvin pingvinR = new Pingvin();

 repulok(sasR);
 //repulok(pingvinR);
 }
}
```

```
davee@davee-K501UB:~/Dokumentumok/bhax/attention_raising/Source/2_liskov$ javac LiskovJava.java
davee@davee-K501UB:~/Dokumentumok/bhax/attention_raising/Source/2_liskov$ java LiskovJava
A sas repül.
davee@davee-K501UB:~/Dokumentumok/bhax/attention_raising/Source/2_liskov$
```

## 12.2. ábra. Az LSP betartása

A program futtatása után már pontosan azt az eredményt kapjuk, amit vártunk. Ha a programuk végén kivesszük a kommentjelet a repulok (pingvinR) előtt, akkor a fordítás során hibaüzenetet fogunk kapni, mert a pingvin nem örökölheti a repülési képességet az LSP miatt. Most pedig ugyanezt nézzük meg c++ verzióban is. Először kezdjük a Liskov elv megsértésével:

```
#include <iostream>

// ez a T az LSP-ben
class Madar {
public:
 virtual void repul() {
 std::cout << "Tud repülni.\n";
 };
};

// ez a két osztály alkotja a "P programot" az LPS-ben
class Program {
public:
 void fgv (Madar &madar) {
 madar.repul();
 }
};

// itt jönnek az LSP-s S osztályok
class Sas : public Madar
{};

class Pingvin : public Madar // ezt úgy is lehet/kell olvasni, hogy a ←
 pingvin tud repülni
{};

int main (int argc, char **argv)
{
 Program program;
 Madar madar;
 program.fgv (madar);

 Sas sas;
 program.fgv (sas);

 Pingvin pingvin;
 program.fgv (pingvin); // sérül az LSP, mert a P::fgv röptetné a ←
 Pingvint, ami ugye lehetetlen.
```

```
}
```

## 12.2. Szülő-gyerek

Írjunk Szülő-gyerek Java és C++ osztálydefiníciót, amelyben demonstrálni tudjuk, hogy az ősön keresztül csak az ős üzenetei küldhetők!

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/tree/master/attention\\_raising/Source/Liskov\\_2/szulo](https://gitlab.com/davidhalasz/bhax/tree/master/attention_raising/Source/Liskov_2/szulo)

Ebben a feladatban olyan programot mutatok be, amivel demonstrálni lehet, hogy az ős osztály csak a azt tudja elérni a gyermekén, ami már az ősosztályban definiáltunk.

```
class Teacher {

 protected String tName;

 public void setName(String name) {
 tName = name;
 }
}

class Student extends Teacher {

 public String getName() {
 return tName;
 }
}

class School {
 public static void main (String args[]) {

 Teacher t = new Student();
 t.setName("Tanár");

 Student s = new Student();
 s.setName("Diák");

 System.out.println(s.getName() + " " + t.getName());
 }
}
```

Ez a program tehát hibát fog jelezni a `Teacher t = new Student();` miatt, mert a program nem engedi, hogy az altípust szupertípusként használjuk.

```
davee@davee-K501U8:~/Dokumentumok/bhax/attention_raising/Source/2_liskov/szulo_gyerek$ javac School.java
School.java:26: error: cannot find symbol
 System.out.println(s.getName() + " " + t.getName());
 ^
 symbol: method getName()
 location: variable t of type Teacher
1 error
davee@davee-K501U8:~/Dokumentumok/bhax/attention_raising/Source/2_liskov/szulo_gyerek$
```

### 12.3. ábra. A School.java program fordítása

És most ugyanez c++ nyelven, ahol a fordítás során a következő hibaüzenetet kapjuk: `class Teacher' has no member named getName'`;

```
#include<iostream>
#include <string>

class Teacher {
 public:

 std::string t_name;

 Teacher(std::string name) {
 t_name = name;
 }

 std::string teachName() {
 return t_name;
 }
};

class Student : public Teacher {
 public:
 Student(std::string s_name): Teacher(s_name) {
 t_name = s_name;
 }

 std::string getName() {
 return t_name;
 }
};

int main (int argc, char **argv) {
 std::string stName = "Thomas";

 Teacher* t = new student(stName);
 Student* s = new Student(stName);

 std::cout << s->getName() << t->getName() << std::endl;
}
```

## 12.3. Anti OO

A BBP algoritmussal 4 a Pi hexadecimális kifejtésének a 0. pozíciótól számított  $10^6$ ,  $10^7$ ,  $10^8$  darab jegyét határozzuk meg C, C++, Java és C# nyelveken és vessük össze a futási időket!

Ebben a feladatban az előző fejezetben megismert BBP algoritmussal fogunk dolgozni. A különböző nyelven megírt programokat most nem célozom bemásolni és ismertetni, mert lényeges különbségek nincsenek benne és itt most inkább a kapott eredmények az érdekesek számunkra. Mindegyik programban a for ciklus fejlécében a `d` változó értékét kell változtatni, úgy, hogy az 1-es után hat/hét/nyolc darab számjegy álljon. A programok fordítása majd futtatása az alábbiak alapján történik:

### C

```
g++ pi_bbp_bench.c -o pi_bbp_bench -lm
./pi_bbp_bench
```

### C++

```
g++ BBP_test.cpp -o bbpTest
./bbpTest
```

### Java

```
javac PiBBPBench.java
java PiBBPBench
```

### C#

```
mcs PiBBPBench.cs
mono PiBBPBench.exe
```

A programokat egy Asus K501UB laptopon teszteltem, amiben egy Intel Core i5 6200U processzor és 12GB SDRAM található benne. A kapott eredményeket egy táblázatban foglaltam össze, amiben az figyelhető meg, hogy a legrosszabbul a C++ verzió teljesített, míg a legjobb eredményt a Java nyelven megírt programmal kaptuk meg.

	C	C++	Java	C#
$10^6$	2.111283	2.44803	1.883	1.922333
$10^7$	24.401839	27.7696	22.144	22.595576
$10^8$	282.93285	313.902	248.324	256.861381

12.1. táblázat. BBP algoritmus eredménye

## 12.4. Hello, Android!

Élesszük fel az SMNIST for Humans projektet! <https://gitlab.com/nbatfai/smnist/tree/master/forHumans/-/SMNISTforHumansExp3/app/src/main> Apró módosításokat eszközölj benne, pl. színvilág.

Megoldás forrása: [https://gitlab.com/davidhalasz/bhax/tree/master/attention\\_raising/Source/Liskov\\_2/SMNIST](https://gitlab.com/davidhalasz/bhax/tree/master/attention_raising/Source/Liskov_2/SMNIST)

Az SMNIST egy olyan java nyelven írt program, mely megvizsgálja, hogy milyen gyorsan hány pontot tudunk megszámolni. Minél több pontot érünk el, annál gyorsabban váltakoznak a pontok. Ha meg szeretnénk változtatni a feladatban szereplő programunk színvilágát, akkor azt az SMNISTSurfaceView.java programban tehetjük meg.

```
int[] bgColor =
{
 android.graphics.Color.rgb(11, 180, 250),
 android.graphics.Color.rgb(11, 250, 180)
};
```

Jelenleg kék és zöld között váltakoznak a színek. A piros és a sárga színhez a következő módosításokra lesz szükségünk:

```
int[] bgColor =
{
 android.graphics.Color.rgb(252, 2, 1),
 android.graphics.Color.rgb(252, 253, 1)
};
```

## 12.5. Ciklomatikus komplexitás

Számoljuk ki valamelyik programunk függvényeinek ciklomatikus komplexitását! Lásd a fogalom tekintetében a [https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2\\_2.pdf](https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_2.pdf) (77-79 főlíát)!

Felhasznált irodalom: [MERES]

Megoldás forrása:

A ciklomatikus komplexitás a gráfelméleten alapul és a program vezérlésének bonyolultságát méri. Képlete:  $M = E - N + 2P$  ahol az  $E$  a gráf éleinek száma, az  $N$  a gráfban lévő csúcsok száma és a  $P$  pedig az összefüggő komponensek száma. Lényegében a forráskódban az elágazásokból felépülő vezérfolyam-gráf pontjai és a köztük lévő élek alapján számítható. A program eredménye egy egész szám lesz és minél alacsonyabb az érték, annál kevésbé bonyolult a vizsgált program.

Ciklomatikus komplexitás	Kockázati szint
1-10	egyszerű program, kevés kockázat
11-20	Mérsékelten bonyolult program, mérsékelt kockázat
21-50	Bonyolult program, magas kockázat
50+	Nem tesztelhető program, nagyon magas kockázat

12.2. táblázat. Ciklomatikus komplexitás

A ciklomatus komplexitásnak vannak hátrányai is. Mivel főleg a szoftver döntési komplexitását méri, ezért az olyan adatorientált programoknál, amiben nincsenek feltételes utasítások, akkor alacsony értéket fog kapni, de ettől még elég komplex és nehezen érthető marad a program. Továbbá a nem beágyazott és a mélyen beágyazott ciklusok ugyanolyan súllyal lesznek számolva, annak ellenére, hogy az utóbbit nehezebb megérteni.

Most pedig lássuk, hogyan is kell lefuttatni a tesztet. Először fel kell telepítenünk a maven-t, ami a `sudo apt install maven` paranccsal ehetünk meg. Hozzunk létre egy maven projektet a

```
mvn archetype:generate -DgroupId=com.monolith -DartifactId=LZW -DinteractiveMode=false
```

parancs megadásával. Ez létrehoz egy projektet egy mappában, ezt nyissuk meg a Visual Studio Code-ban. Az `src/main/java/com/BBP` mappába másoljuk be a vizsgálni kívánt java fájlokat. Én most a `LZWBinFA` és a `PiBBPBench` programokat tettem be. Találni fogunk még egy `pom.xml` fájlt is, ebbe kell beillesztenünk a pluginokat, ami jelen esetben így néz ki:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.BBP</groupId>
 <artifactId>BBP</artifactId>
 <packaging>jar</packaging>
 <version>1.0-SNAPSHOT</version>
 <name>BBP</name>
 <url>http://maven.apache.org</url>

 <build>
 <plugins>
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-site-plugin</artifactId>
 <version>3.8.2</version>
 </plugin>
 </plugins>
 </build>

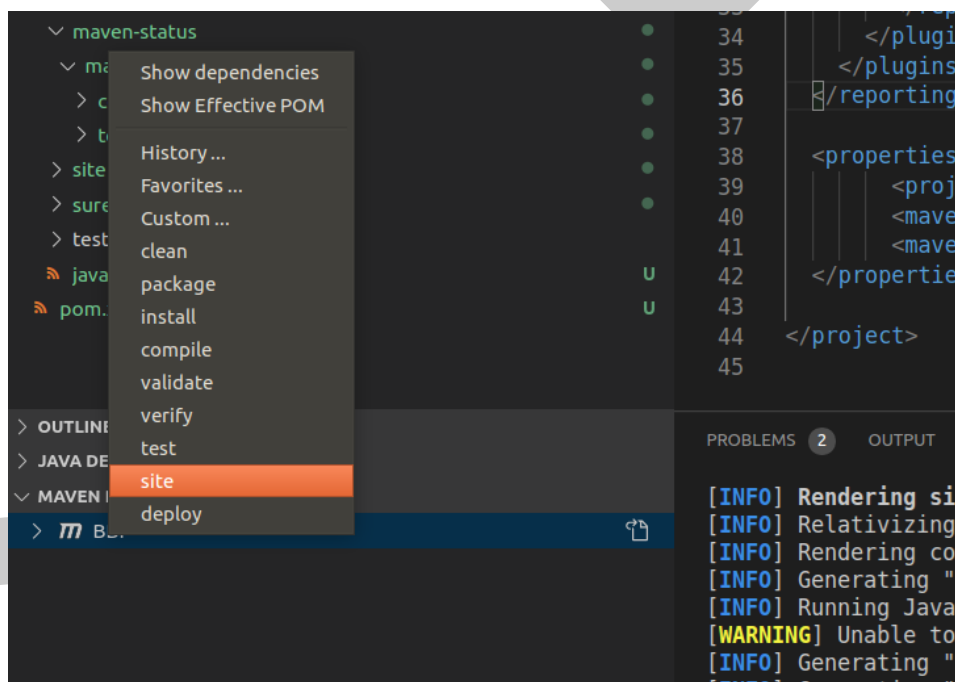
 <reporting>
 <plugins>
 <plugin>
 <groupId>org.codehaus.mojo</groupId>
 <artifactId>javancss-maven-plugin</artifactId>
 <version>2.1</version>
 <reportSets>
 <reportSet>
 <reports>
 <report>report</report>
 </reports>
 </reportSet>
 </reportSets>
 </plugin>
 </plugins>
 </reporting>
</project>
```



```
 </reportSets>
 </plugin>
 </plugins>
 </reporting>

 <properties>
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 <maven.compiler.source>12</maven.compiler.source>
 <maven.compiler.target>12</maven.compiler.target>
 </properties>
</project>
```

Ha ezzel megvagyunk már csak futtatni kell, amit egyszerűen tehetünk meg a Visual Studio Code-ban, úgy hogy az Explorer oldalsávban látni fogjuk a MAVEN PROJECTS menüt, amin belül lesz a BBP. Itt érhetjük el jobb klikkel a site funkcióval a futtatást.



12.4. ábra. Program futtatása

Először feltelepíti a szükséges plugint, majd a site mappába teszi be a javanccss.html fájlt. Ezt egyszerűen futtathatjuk a böngészőnkben. Az eredmények táblázatokban lesznek betéve. Nekünk most a Methods táblázat az érdekes. Az LZWBinFa CCN értéke 14, ami mérsékelten bonyolult értékű program, míg a PiBBPBench pedig 2-es értéket kapott, ami egész jó érték.

## Methods

Methods

[ [package](#) ] [ [object](#) ] [ [method](#) ] [ [explanation](#) ]

**TOP 30 Methods containing the most NCSS.**

Methods	NCSS	CCN	Javadocs
LZWBInFa.main(String[])	40	14	0
PIBBPBench.main(String[])	20	2	1
LZWBInFa.operator(char)	15	4	0
PIBBPBench.n16modk(int,int)	14	5	1
LZWBInFa.getSzoras()	10	2	0
LZWBInFa.kiir(Csomopont,java.io.PrintWriter)	9	3	0
LZWBInFa.ratlalag(Csomopont)	9	4	0
LZWBInFa.rszozas(Csomopont)	9	4	0
LZWBInFa.melyseg(Csomopont)	8	3	0
LZWBInFa.getAtlag()	5	1	0

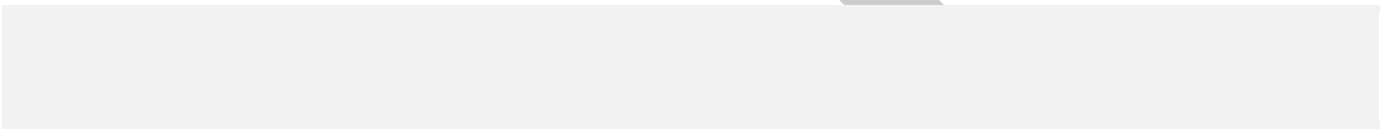
12.5. ábra. Ciklusos komplexitás eredménye

## 13. fejezet

# Helló, Mandelbrot!

### 13.1.

Megoldás forrása:



DRY

## 14. fejezet

# Helló, Berners-Lee!

### 14.1. Nyékyné Gaizler Judit: Java 2 Útikalauz programozóknak 5.0

A Java egy teljesen objektumorientált nyelv, mely jelölésrendszerében hasonlít a C++ nyelvhez, azzal a különbséggel, hogy a Java nyelvet teljesen az alapjaitól építették fel, míg a C++ a C nyelv továbbfejlesztett változata. Ebben a nyelvben nincsenek mutatók és a program osztályokból épül fel. Az osztályok pedig adattagokból és metódusokból állnak. A programozó rendelkezésére pedig rengeteg előre megírt osztály áll, amely a Java fejlesztői környezet részét alkotják. Ezekkel az előre megírt osztályokkal rengeteg időt spórolhatunk meg és ez a program-újrafelhasználás az objektum-orientált programozás egyik alapelve. A Java egyik előnye még a hordozhatóság, ami más operációs rendszereken történő gond nélküli futtatását teszi lehetővé. A platformfüggetlenség eléréséhez a fordítóprogram Java bájtkódra fordítja le a forráskódot, amit a Java virtuális gép (JVM) tovább fordít az adott gép kódjára. A fordítás a parancssorban a `javac` utasítással történik, majd az így kapott fájlt a `java` utasítással futtathatjuk.

A Javában a változók értékét teljes egészében a változó deklarált típusa határozza meg. Ha a változót primitív típusúnak deklaráljuk (logikai, `char`, `byte`, `short`, `int`, `long`, `float` vagy `double`), akkor mindig megad egy értéket. A tömbök, osztályok és az interfészek pedig mindig referenciatípusúak lesznek. Itt nincsenek globális változók és függvények, helyette minden adat és művelet valamilyen objektumhoz tartozik. Itt nincsenek mutatók és objektumok, ez utóbbi a dinamikus tárterületen jönnek létre és hivatkozásokon keresztül érhetjük el őket. A C++ esetében bármilyen típus (primitív vagy összetett) érték vagy referencia alapján határozható meg. Ebben a nyelvben mutatókat és hivatkozásokat is használhatunk.

A vezérlés a C++ nyelvhez hasonlóan történik itt is. A legszembetűnőbb különbség az, hogy a Java-ban nincs `goto` utasítás. Pontosabban maga a `goto` kulcsszó ismert és le is van foglalta, de nincs jelentése, így használata esetén hibát fogunk kapni. Ez nagyban hozzájárul a biztonságosabb programok írásához. Helyette használhatjuk a `break` utasítást ciklusok elhagyására, vagy `continue` utasítással folytathatjuk azt. Fontos még megjegyezni, hogy a C++ nyelvvel ellentétben a Java-ban nem deklarálhatunk változót az `if` utasítás fejében.

A memóriakezelés a Java egyik nagy erőssége, mert nekünk csak létre kell hoznunk az objektumokat és amikor már nincs rájuk szükségük, egyszerűen csak a szemétyűjtő rendszerre hagyjuk a lefoglalt memória automatikus felszabadítását. Ez kétféle módon történhet: egyik a `new` operátor használatával, azaz amikor új objektumot hozunk létre, vagy pedig `finalize` metódus használatával, ami akkor szabadít fel egy objektumot, mielőtt az adott tárterület újrafelhasználásra kerülne. Itt nincs lehetőség az adott objektum memóriaigény lekérdezésre, mivel a Java-ban minden a virtuális gépen történik, ezért ennek kiszámítására

sose kapnánk pontos választ. Ezzel szemben a C++ nyelvben manuálisan tetszőleges memóriablokkokat foglalhatunk le és pointerok használatával közvetlenül történik.

A C++ standard könyvtára korlátozott számú alap- és általános célú összetevőket tartalmaz. A Java lényegesen nagyobb szabványos könyvtárral rendelkezik és amit a `java.lang` csomag tartalmaz. Mivel ez a csomag olyan alapvető típusokat definiál, amik a programok futtatásához szükség van, ezért minden Java programba automatikusan importálódik. Az alcsomagokat és az abban lévő osztályokat, metódusokat pont operátorral érhetjük el. Például a

```
java.lang.System.out.println("Helló Világ!");
```

esetében a `println` segítségével írjuk ki a szöveget, ami az `out` objektum egy metódusa. Az `out` a `System` osztály egy adattagja, ami a `java.lang` csomag egy osztálya.

Mint minden objektum-orientált nyelv, így a C++ és a Java is támogatja az öröklődést, de mindkét esetben eltérő módon történik. A C++ támogatja a többszörös öröklődést, ahol egy osztály tetszőleges számú osztálytól öröközhet. Ezzel szemben a Javában nincsen többszörös öröklődés, ugyanis minden osztálynak csak egyetlen őse lehet. Egy osztály egy már meglévőből történő származtatása az `extends` kulcsóval történhet és nem tesz különbséget a `protected`, `privát` és `publikus` öröklődés között. Interfészek esetén is az `extends` kulcsszót használjuk és itt már támogatja a Java a többszörös öröklődést is. Ezek implementálásához az `implements` kulcsszót kell használni.

A Java-ban is van kivételkezelés, ahol a `try-catch-throw` mellett `finally` kifejezést is használjuk. Csak `Throwable` osztályú kivételeket lehet dobni. A gyári könyvtárakban sok a `Throwable`-tól öröklő osztály szerepel: az `Error` és az `Exception` osztályok leszármazottai.

## 14.2. Forstner Bertalan, Ekler Péter, Kelényi Imre: Bevezetés a mobilprogramozásba. Gyors prototípus fejlesztés Python és Java nyelven

Eddigi tanulmányaim során még nem foglalkoztam Python-nal, ezért csak annyit tudtam róla, hogy könnyen megtanulható nyelv és ráadásul kevesebb időbe telik egy program felépítése a C/C++ vagy Java nyelvhez képest, így kíváncsi voltam erre a könyvre. Ebből a könyvből megtudtam, hogy a Python is egy magas szintű objektumorientált nyelv. Teljesen platformfüggetlen és nincs szükség fordításra, elegendő csak futtatni az alkalmazást. Objektumorientált nyelvként rengeteg előre megírt modult hasznosíthatunk újra. Annak ellenére, hogy a Pythonnal felépített programok sokkal tömörebbek, mint például egy C program, még így is könnyen olvashatóak. Ezekkel a tulajdonságokkal rengeteg időt spórolhatunk meg.

Ami szokatlan volt számomra, hogy a Python nyelvben nem használunk sorvégi `;"` jelet, ugyanis itt minden egyes sor egy utasítás. Az állításokat pedig kapcsos zárójel helyett behúzásokkal jelezzük. Fontos, hogy a szkript első utasítása nem lehet behúzott és mindenhol egységesen kell kezelni a tabot.

Típusok és változók tekintetében a következőket tudtam meg: minden adat objektum és ennek típusa határozza meg, hogy milyen műveleteket végezhetünk az adatokon. Nem kell megadnunk a változók típusát, mert a rendszer futási időben automatikusan kitalálja azt. Adattípusok lehetnek a számok, sztringek, ennesek, listák és szótárak. Az ennesek objektumok gyűjteményei, vesszővel elválasztva és zárójelben írva. A listát szögletes zárójelek közé írjuk, szintén vesszővel elválasztva. A szótárt pedig kapcsos zárójelek között vesszővel elválasztva írjuk. Ez utóbbinál kulcsokkal azonosított elemeket használunk, ami

engem a Vue.js-re emlékeztet. Változók megadása a többi nyelvhez hasonlóan "=" jel használatával történik. Java-hoz hasonlóan itt is automatikus garbage collector működik, azaz a memória felszabadítással nem kell foglalkoznunk.

Ciklusok és elágazások során nem használunk kapcsos zárójeleket, például az if sor után kettőspontot írunk, majd egy új sorban, tabulátort használva adjuk meg az utasítást:

```
if szam < 0:
 print 'kisebb, mint 0'
elif szam > 0:
 print 'Nagyobb mint 0'
else:
 print 'Pontosan 0'
```

A Pythonban is megtalálható a kivételkezelés, ami a try kulcsszóval kezdődik, ami után szerelepnie kell a kódblokknak, hiba esetén az exceptre ugrik a vezérlés. Beíráhatunk még egy else ágat is.

```
try:
 utasítások
except [kifejezés]:
 utasítások
[else:
 utasítások]
```

Függvényeket pedig a def kulcsszóval adhatunk meg. A függvények az eddigi nyelvekhez hasonlóan rendelkezhetnek paraméterekkel. Ugyanígy megadhatunk osztályokat is, amelyek példányai az objektumok, sőt az osztályok örökölhetnek más osztályoktól is. A Python másik nagy előnye még továbbá, hogy szabványos modulok állnak rendelkezésünkre. Ezeknek a moduloknak a célja a fejlesztés megkönnyítése, például felhasználói felületeket, hálózatokat, vagy a kamerát stb. kezelhetjük.

## **IV. rész**

### **Irodalomjegyzék**

### 14.3. Általános

[MARX] Marx, György, *Gyorsuló idő*, Typotex , 2005.

### 14.4. C

[KERNIGHANRITCHIE] Kernighan, Brian W. És Ritchie, Dennis M., *A C programozási nyelv*, Bp., Műszaki, 1993.

### 14.5. C++

[BMECPP] Benedek, Zoltán És Levendovszky, Tihamér, *Szoftverfejlesztés C++ nyelven*, Bp., Szak Kiadó, 2013.

### 14.6. Lisp

[METAMATH] Chaitin, Gregory, *META MATH! The Quest for Omega*, [http://arxiv.org/PS\\_cache/math/pdf/0404/0404335v7.pdf](http://arxiv.org/PS_cache/math/pdf/0404/0404335v7.pdf) , 2004.

### 14.7. Bio Intelligence

[BIOINTELLIGENCE] Dr. Buzáné dr. Kis Piroska, *Ismerkedés a TensorFlowrendszerrel*, [http://biointelligence.hu/pdf/tf\\_bkp.pdf?fbclid=IwAR3PBRoiMwwI8QtFXBMnhmvP-NHK7ee5BROJ\\_qk0pzMqqlwlrkwtNx10wpA](http://biointelligence.hu/pdf/tf_bkp.pdf?fbclid=IwAR3PBRoiMwwI8QtFXBMnhmvP-NHK7ee5BROJ_qk0pzMqqlwlrkwtNx10wpA) .

### 14.8. Neurális hálózatok

[NEURALIS] Altrichter Márta, Horváth Gábor, Pataki Béla, Strausz György, Takács Gábor, És Valyon József, *Neurális hálózatok*, [https://www.tankonyvtar.hu/hu/tartalom/tamop425/0026\\_neuralis\\_4\\_4/adatok.html](https://www.tankonyvtar.hu/hu/tartalom/tamop425/0026_neuralis_4_4/adatok.html) , 2006.

### 14.9. Programozás Technika

[PROTECHNIKA] Kúspér Gábor És Radványi Tibor, *Programozás Technika*, [https://www.tankonyvtar.hu/hu/tartalom/tamop425/0038\\_informatika\\_Projektlabor/ch01s03.html](https://www.tankonyvtar.hu/hu/tartalom/tamop425/0038_informatika_Projektlabor/ch01s03.html) .



## 14.10. A Szoftver mérése

[MERES] Dr. Mileff Péter, *A Szoftver mérése*, [https://users.iit.uni-miskolc.hu/~mileff/intszvr/-ISZRM\\_4.pdf](https://users.iit.uni-miskolc.hu/~mileff/intszvr/-ISZRM_4.pdf) .

Köszönet illeti a NEMESPOR, <https://groups.google.com/forum/#!forum/nemespor>, az UDPROG tanulószoba, <https://www.facebook.com/groups/udprog>, a DEAC-Hackers előszoba, <https://www.facebook.com/groups/DEACHackers> (illetve egyéb alkalmi szerveződésű szakmai csoportok) tagjait inspiráló érdeklődésükért és hasznos észrevételeikért.

Ezen túl kiemelt köszönet illeti az említett UDPROG közösséget, mely a Debreceni Egyetem reguláris programozás oktatása tartalmi szervezését támogatja. Sok példa eleve ebben a közösségben született, vagy itt került említésre és adott esetekben szerepet kapott, mint oktatási példa.